QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications[⋆]

Baochun Li and Klara Nahrstedt

Department of Computer Science
University of Illinois at Urbana-Champaign
{b-li,klara}@cs.uiuc.edu
http://cairo.cs.uiuc.edu

Abstract. It is widely accepted that in order to deliver the best Quality-of-Service (QoS), applications need to be adaptive to the fluctuating computing and communication environments. The middleware layer may assist by controlling the behavior of the applications so that they adapt and reconfigure themselves. In this paper, we present *QualProbes*, a set of middleware *QoS Probing and Profiling* services to discover such relationships at run-time. Our approach focuses on meeting the requirements of the *critical performance criterion* in the application. Such criterion may be affected by changes in more than one application-specific QoS parameters, and these parameters have diversely different resource usage patterns. *QualProbes* services are able to precisely capture the effects made to the critical performance criterion when resource availability varies, and thus enable more effective control of the application to adapt to resource variations. Our case study with *OmniTrack*, an omni-directional visual tracking application, provides solid proof that QualProbes significantly enhance our capabilities to satisfy the critical performance criterion, the *tracking precision*, while controlling the adaptation process of the application.

1 Introduction

Recent research advances in Quality-of-Service (QoS) and resource management have brought forth numerous solutions to support QoS-aware applications, so that their demands for both end system and network resources are met. Two major categories of such solutions have evolved. First, *reservation-based systems* employ various resource reservation and admission control mechanisms to enforce the delivery of requested QoS to the applications. Such enforcement may be deterministic or statistical, depending on the policies involved for resource reservation. One drawback of this approach is that many reservation mechanisms demand major overhaul in the design of prevalent operating systems in use today, such as Windows NT, or networking protocols, such as TCP/IP. In

[⋆] 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.

contrast, *adaptation-based systems* operate based on best-effort environments, and attempt to adapt themselves or the applications for the purpose of providing the *best possible* QoS under available resource conditions, and of achieving the most graceful quality degradation in case of scarce resources.

It is advantageous to implement such *adaptation-based systems* in the middleware level, 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 systems over reservation-based systems. Indeed, notable examples of adaptation-based systems, such as the *QuO* [1] and *Da CaPo++* [2], implement adaptation-based services in the middleware. 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 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. However, it lacks an easy way to manifest the relationship between application-specific adaptation choices and the actual changes in resource demands, caused by reconfiguring an adaptive application. We take the latter alternative in our approach.

Since the primary objective common to all adaptation-based approaches is to provide the *best possible* QoS with the current resource availability in a swiftly changing environment, the problem comes to the proper choice of a certain *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 paper, 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*, an omni-directional visual tracking application, the *tracking precision*[1] is the most critical QoS parameter in the tracking application. The *critical performance criterion*, therefore, is to keep the tracking precision accurate and stable.

In this paper, we present *QualProbes*, a set of middleware QoS probing and profiling services, that are uniquely designed to address the following problems:

[1] The *tracking precision* is a quantitative measurement of the collective performance of all concurrently running tracking algorithms, also referred to as *trackers*.

(1) How do changes in non-critical application QoS parameters relate to the critical QoS parameter, and thus the critical performance criterion? (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 translate to appropriate control actions activated by the middleware, 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 environment 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.

The rest of the paper is organized as follows. Section 2 briefly introduces the design and architecture of *Agilos* (**Agil**e **QoS**), a middleware control architecture that actively controls the application's adaptation behavior. The *QualProbes* services are introduced and serve as critical core components in the *Agilos* architecture. Section 3 presents our theoretical and practical solutions to the above problems, forming the basis of *QualProbes*. Section 4 shows a detailed experimental analysis of the control effectiveness from the middleware, with and without the assistance of *QualProbes*. We use *OmniTrack*, our omni-directional visual tracking application, as an example of complex applications. Section 5 discusses related work and Section 6 concludes the paper.

2 *Agilos* Middleware: A Background Introduction

The ultimate objective of *Agilos*, our middleware control architecture, is to control the adaptation process within the application so that it is steered towards the satisfaction of application-specific *critical performance criterion*. In order to accomplish the objective, the core middleware components of *Agilos* consist of application-neutral *Adaptors* and application-aware *Configurators*, which reflect a two-level hierarchy of middleware control. 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 is illustrated as in Figure 1.

Though the Adaptors and Configurators form the basis of the *Agilos* architecture, three additional components are necessary to complete the design and to achieve the desired functionality. First, the *Negotiator* is responsible for all communications among *Agilos* middleware components on different end systems. Second, the *Observer* is responsible for monitoring resource availability and inspecting any application-specific parameters. Third, *QualProbes* provide QoS probing and profiling services so that application-specific mappings between the two adaptation levels can be derived. This paper focuses on the algorithm design of the *QualProbes* services.

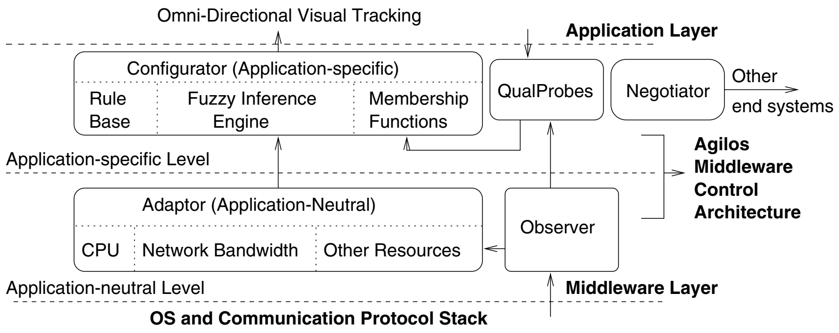

Omni-Directional Visual Tracking
Application Layer
Configurator (Application-specific)
Rule Base
Fuzzy Inference Engine
Membership Functions
QualProbes
Negotiator
Other end systems
Application-specific Level
Agilos Middleware Control Architecture
Adaptor (Application-Neutral)
CPU
Network Bandwidth
Other Resources
Observer
Application-neutral Level
Middleware Layer
OS and Communication Protocol Stack

Fig. 1. The Hierarchical Design of the *Agilos* Architecture

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 replacing the "fuel" of the Configurator. As detailed in previous work [3], the Configurator is designed as a rule-based fuzzy control system. As illustrated, the Configurator can be partitioned into three parts: the *Fuzzy Inference Engine*, *Membership Functions* and *Rule Base*. While *fuzzy inference engine* is application-neutral, the "fuel", namely the *rule base* and *membership functions* of associated linguistic variables, are application-specific. Such model guarantees that discrete adaptation choices and a wide variety of resource/application QoS mappings can be addressed easily with a replacement of the 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 *cpu_demand* and *throughput_demand*, and examples of values are *below_average* or *very_low*. 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_demand** *is very_high and* **throughput_demand** *is below_average then* **configuration** *is 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.

3 *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*.

3.1 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.

The Application Model In all subsequent discussions about application QoS parameters and resource types, we assume a *Task Flow Model* for distributed applications. A complex distributed application can be modeled as several *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 can be used to illustrate such a 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

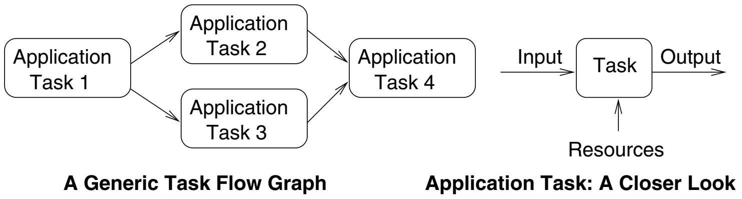


Fig. 2. Illustration of The Task Flow Model

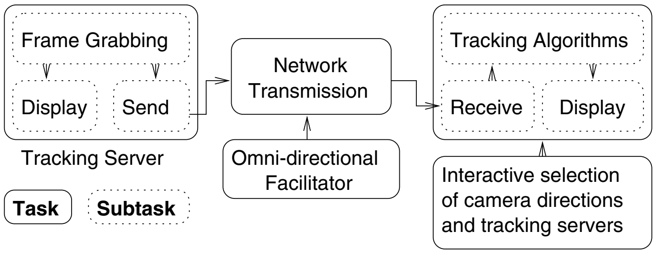


Fig. 3. The Task Flow Model of *OmniTrack*

they may be partitioned as *subtasks*. As an example, the Task Flow Model of *OmniTrack* is shown in Figure 3.

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 [2]. 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 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. An illustration of our directed dependency tree model and an real-world example with *OmniTrack* is given in Figure 4.

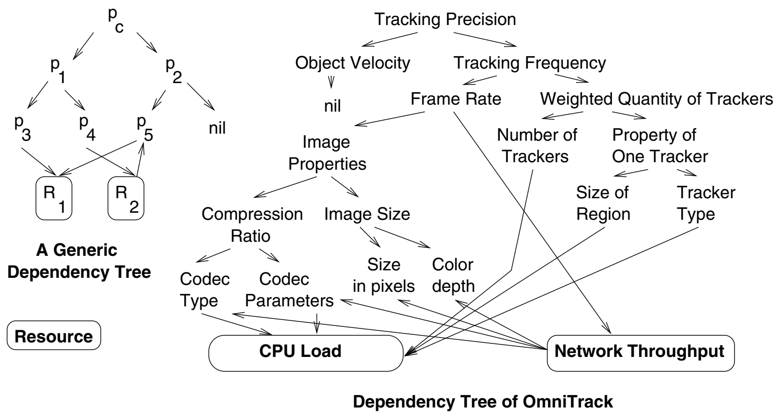


Fig. 4. The Dependency Graph for Application QoS Parameters

Characterizing the Relationship between Dependent Nodes Once we have established the dependency tree of QoS parameters for an application [3], the relationship between dependent nodes needs to be characterized appropriately. We assume that for $\forall i, \forall t$, there exists $\{p_i\}_{min}$ and $\{p_i\}_{max}$ such that $\{p_i\}_{min} \leq p_i(t) \leq \{p_i\}_{max}$, any values beyond 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:

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

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

$$\begin{aligned} \Delta p_i &= f_{i,x,y}(\Delta p_x, \Delta p_y) \\ p_k &= \{p_k\}_{min} + \Delta p_k, \text{ with } k \in \{i, x, y\} \\ 0 \leq \Delta p_k &\leq \{p_i\}_{max} - \{p_i\}_{min} \end{aligned} \tag{1}$$

where Δp_k is a normalized value of p_i. 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 $\Delta p_i = f_{i,x}(\Delta p_x)$. 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 5 visually shows the above characterization.

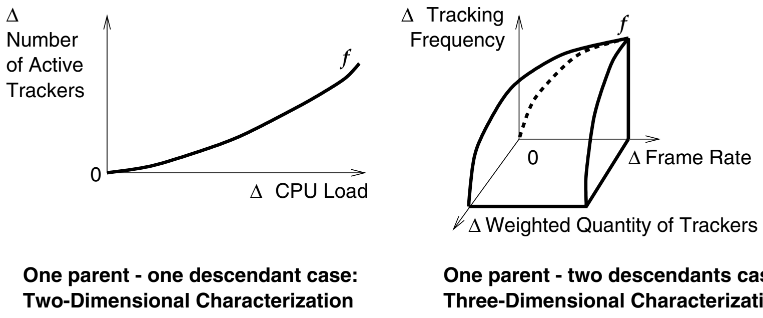


Fig. 5. Characterization of Dependencies among QoS Parameters

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 4, we have

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

and

$$\Delta R_2 = f^r_{r_2,5}(\Delta p_5) \tag{3}$$

which characterizes the relationship between p_c and resources R_1 and R_2.

3.2 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. The QualProbes services kernel 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. Implementation-wise, we utilize the CORBA Property Service. 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.
2. *Tunable.* They should be either directly or indirectly tunable from outside of the application. Since the application exports interfaces to the middleware Configurator for such tuning and reconfiguration, QualProbes services only need to reuse these interfaces to control the QoS parameters in the application.

Having ready "read/write" access to the application QoS parameters, QualProbes services execute a *QoS Profiling* algorithm in their kernel. The algorithm traverses the dependency tree from leaves up to the root, and attempts to discover the function 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 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.

As an concrete example, Figure 7 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.

3.3 Towards Better Middleware Control

The design of QualProbes services in previous sections addresses the problem of discovering relationships between the critical performance criterion 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 with actual membership functions and inference rules in the Configurator.

The Inference Rules Based on our extensive experiences with the real-world application *OmniTrack*, 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

```
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 observed $R_i$
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 observed $p_i$
    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 observed $p_i$

tune($p_i$, value)
    if $p_i$ is directly tunable via exported interface
        call application exported interface to set $p_i$ = value
    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;
```

Fig. 6. QualProbes Services Kernel Algorithm

adaptation choices and best optimize 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.

***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_demand* **is** *very_high* **and** *throughput_demand* **is** *very_low* **then** *configuration* **is** *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 to demand more CPU resources. This yields a high *cpu_demand* value. Similarly, when the network is congested and there are very low bandwidth available, the network

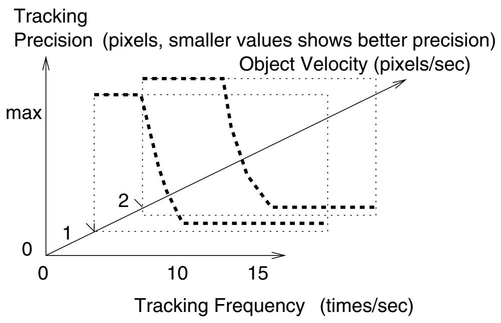


Fig. 7. QualProbes Services: An Example

bandwidth adaptor suggests that the application demand less network bandwidth, thus yielding a low value in *throughput_demand*. Second, the inference rule decides that if *cpu_demand* is high and *throughput_demand* 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 *very_high* and *very_low* decide the activation timing of such reconfiguration choice.

The question is: How "high" is *very_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 by the profiles obtained from QualProbes services. For example, *very_high* can be defined as higher than 60%, while *very_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 7. One of the corresponding inference rule is:

if *tracking_frequency* **is** *low* **and** *object_velocity* **is** *medium* **then** *configuration* **is** *remove_tracker*

As illustrated by Figure 7, 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 *remove_tracker* 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.

4 Case Study: OmniTrack

4.1 *OmniTrack*: An Introduction

As a case study, we have developed *OmniTrack*, a distributed omni-directional visual tracking system, using tracking algorithms in the *XVision* [4] project. *OmniTrack* is a flexible, multi-threaded and client-server based application, which adopts complex tracking capabilities in multiple dimensions, 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 8.

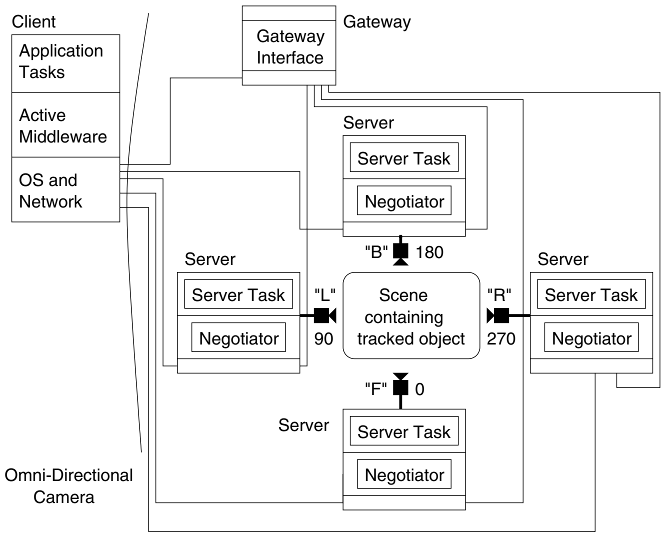


Fig. 8. *OmniTrack*: A Distributed Omni-Directional Visual Tracking System

OmniTrack is implemented in Windows NT, deployed under the control of Agilos middleware. *OmniTrack* exports a *control interface* which is clearly defined in IDL. All control commands made by the Agilos middleware is carried out through such a control interface via CORBA. This ensures that Agilos middleware architecture is generic and not bound to any specific applications. 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.

4.2 Experiments with OmniTrack

We have carried out a series of experiments with *OmniTrack*. In our experimental setting, while the basic inference rules in the Configurator are hand-tuned, we have been successful in applying the threshold values extracted from the profiles discovered by QualProbes services. Without QualProbes, it has been very difficult to specify appropriate membership functions to complete the definitions for the "fuel" of Configurator, let alone to put the Configurator in active service. With QualProbes services enabled and QoS profiles generated, such tasks have been straightforward. We feel that with QualProbes services, we are able to "see through" the internal behavior of the *OmniTrack* application. Such transparency has provided us with unparalleled assistance in our understanding of *OmniTrack*, as well as its control optimally. The following preliminary results are obtained in two different experimental scenarios.

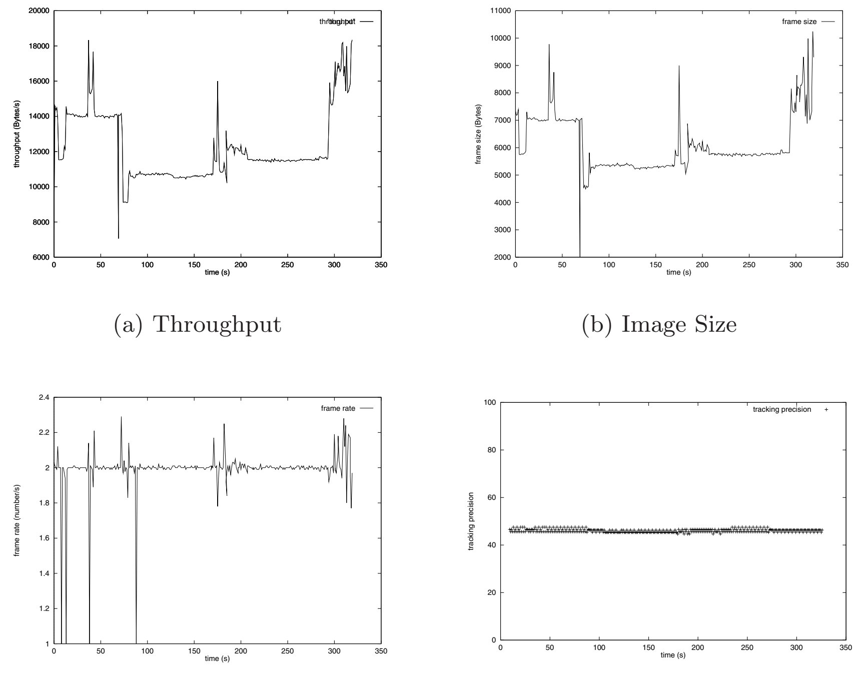
(1) An animated video sequence is streamed from the server to the client using Motion-JPEG compression. The animated sequence is 320*240 pixel frame size video sequence. Within this scenario, we illustrate basic adaptation possibilities by adapting the image size. We measure the tracking precision and show that the tracking precision remains stable with fluctuating bandwidth availability.

(2) Live video is streamed from the active server to the client in a omnidirectional setting. The content of the live video is captured by the digital camera and an image grabber. We use 320*240 pixel frame size for the default initial properties of the live video. Within this scenario, we illustrate both throughput-related and CPU-related adaptation in action simultaneously, such as compression and dropping trackers. We finally measure the tracking precision and show that it remains stable with fluctuating CPU availability.

4.3 Experimental Results

Scenario 1 In Figure 9, we illustrate basic adaptations by adapting the image size on a Motion-JPEG compressed video stream. We show from the results that, despite the fluctuating network bandwidth availability, the tracking precision remains stable under the control of Agilos middleware.

Scenario 2 Figure 10 and Table 1 show the experimental results. With respect to parameter-tuning adaptations, Figure 10(b) shows the result of Adaptors and Tuners by changing image size during the fluctuation of network bandwidth shown in Figure 10(a). With respect to reconfiguration alternatives, Figures 10(c), 10(d) and Table 1 show the Configurator in action. In this experiment, Figure 10(c) shows the CPU load fluctuation, while Table 1 shows the control actions generated by the Configurator at various time instants, and executed by the application. Figure 10(d) shows the actually measured tracking precision. The first tracker tracks a more important object, so if a `drop_tracker` event is signaled, later trackers should be dropped. We note that the tracking precision stays stable in a small range, which shows that the adaptation efforts are



(a) Throughput (b) Image Size

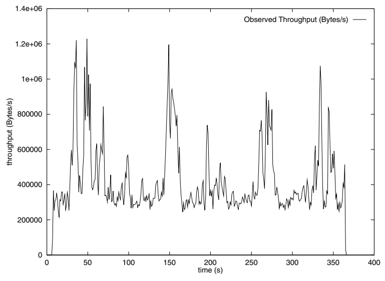
(c) Frame Rate (d) Tracking Precision

Fig. 9. Scenario 1

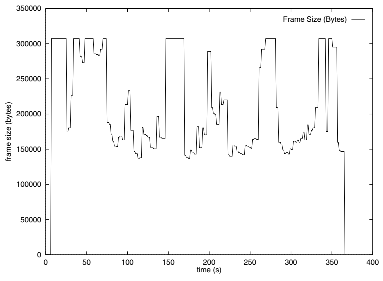
successful to lock the trackers on the objects, before they are dropped for more important trackers.

5 Related Work

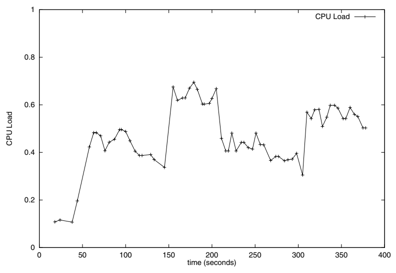
It has been widely recognized that many QoS-constrained distributed applications need to be adaptive in heterogeneous environments. Recent research work on resource management mechanisms at the systems level expressed much interests in studying various kinds of adaptive capabilities. Particularly, in wireless networking and mobile computing research, because of resource scarcity and bursty channel errors in wireless links, QoS adaptations are necessary in many occasions. For instance, in the work represented by [5][6], a series of adaptive resource management mechanisms were proposed that applies to the unique characteristics of a mobile environment, including 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 lost in network revenue, which is minimized. As another example, Noble et al. in [7] investigated



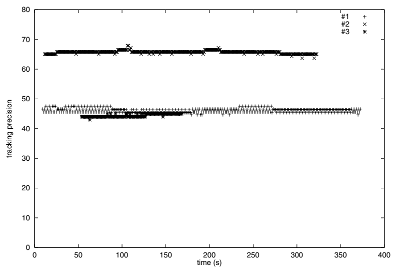
(a) Throughput



(b) Chopped Image Size



(c) CPU Load



(d) Tracking Precision

Fig. 10. Scenario 2

in an application-aware adaptation scheme in the mobile environment. Similarly to our work, this work was also built on a 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.

Another related group of previous work studies the problem of dynamic resource allocations, often at the operating systems level. Noteworthy work are presented in [8][9][10]. The work in [8] 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 [9], the global resource management system was proposed, which relies on middleware services as agents to assist resource management and negotiations. In [10], 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.

Recently, in addition to studies in the networking and resource management levels, many active research efforts are also dedicated to various adaptive func-

Table 1. Control Actions produced by the Configurator (follow the time scale in Figure 10(c))

Time (sec)	*Control Action from Configurator*
28.22	`uncompress`
51.24	`add_tracker`
67.37	`compress`
167.7	`drop_tracker`
320.4	`drop_tracker`

tionalities provided by middleware services. For example, [11] proposes real-time extensions to CORBA which enables end-to-end QoS specification and enforcement. [1] 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. The work noted in [12] builds a series of middleware-level agent based services, collectively referred to as *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, our approach is both unique and orthogonal in the following aspects. First, in defining QualProbes services, we defined a novel layered model for application-specific QoS parameters, for the purpose that the relationships between such parameters and system resource usage can be probed and profiled with ease. Second, our *Agilos* middleware is *active*, in the sense that rather than attempting to transparently provide adaptive services, it actively controls the applications themselves so that the applications, *not* the middleware components, are the ones to adapt. Third, our work is orthogonal in the sense that we leverage the advantages of any service enabling platforms, including both standard CORBA services or those with customized ORBs. Fourth, 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 Agilos middleware with respect to multiple resources, notably CPU and network bandwidth.

6 Conclusion

This paper 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* services 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 paper is that we have provided a unique approach to "see through" the behavior of the application, especially when environmental or requirement changes may occur. In addition, we have presented some preliminary experimental results with *OmniTrack*, a complex multimedia application that we have developed, in order to verify that our approaches are effective in assisting the understanding of the application, and generating the "fuel" of the Configurator, a key component in the *Agilos* architecture.

References

1. J. Zinky, D. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 1997. 257, 271
2. 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. 257
3. B. Li and K. Nahrstedt, "Dynamic Reconfigurations for Complex Multimedia Applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1999. 259
4. G. Hager and K. Toyama, "The XVision System: A General-Purpose Substrate for Portable Real-Time Vision Applications," *Computer Vision and Image Understanding*, 1997. 267
5. 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. 269
6. V. Bharghavan, K.-W. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer, "The TIMELY Adaptive Resource Management Architecture," *IEEE Personal Communications Magazine*, 8 1998. 269
7. 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*, Oct. 1997. 269
8. R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A Resource Allocation Model for QoS Management," in *Proceedings of 18th IEEE Real-Time System Symposium*, 1997. 270
9. 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. 270
10. 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 System Symposium*, 1997. 270
11. D. Schmidt, D. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Requests," *Computer Communications Journal*, 1997. 271
12. 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*, Dec. 1998, pp. 307–317. 271