A Hierarchical Quality of Service Control Architecture for Configurable Multimedia Applications

Baochun Li
Electrical and Computer Engineering
University of Toronto
bli@eecg.toronto.edu

William Kalter, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
kalter,klara@cs.uiuc.edu

Abstract

In order to achieve the best application-level Quality-of-Service (QoS), multimedia applications need to be dynamically tuned and reconfigured to *adapt* to fluctuating computing and communication environments. QoS-sensitive adaptations are critical when applications run in general-purpose systems, with no mechanisms provided for supporting resource reservations and real-time guarantees. Such adaptations are triggered by resource availability variations caused by best-effort resource allocations in unpredictable open environments.

In this paper, we argue that adaptations are most effective to achieve a better QoS when performed within applications, where they may be optimized towards the best performance tradeoffs across various application parameters with different semantics. However, we believe that decisions about when and how adaptations should occur need to be coordinated, and formalized as a generic algorithm to be applied to a wide range of applications. For this purpose, we first identify an application model to focus on a set of application-specific tuning "knobs" and critical parameters, then propose a polynomial-complexity QoS probing algorithm to quantitatively capture the run-time relationships between the two sets of parameters. Finally, we present a *hierarchical adaptive QoS control architecture* to bridge the gap between original "triggers" of adaptation and actual tuning "knobs" to be invoked. To prove the validity of our architecture and algorithms, we present *Agilos*, a middleware implementation of our hierarchical architecture. Under its control, we show that a configurable multimedia tracking application is able to deliver optimal performance even when operating in unpredictable open environments.

1 Introduction

Typical distributed multimedia applications in deployment today present time-varying demands for computational and communication resources when delivering critical Quality-of-Service (QoS) to their users. These complex applications span multiple hosts and are organized in a client-server or peer-to-peer fashion, over best-effort IP or multi-service broadband networks. QoS provisioning via resource reservations for the maximum level of resource requirement may not be feasible for best-effort IP networks, or cost-effective for multimedia applications with bursty and unpredictable traffic profiles. Under these circumstances, applications need to be dynamically tuned and reconfigured to *adapt*, triggered by resource availability variations caused by unpredictable open environments.

We claim that such adaptations are most effective to achieve a better QoS when performed with the cooperation of applications, rather than *only* within the system layers, such as network protocol stacks, since the ultimate objective of such adaptations should be to achieve satisfactory application-level QoS at all times. Coordination of adaptation should be detached from applications themselves. Such *application-level adaptations* are achieved by requesting applications to export a *control interface* that includes a set of application specific *tuning "knobs"*, which are interfaces to activate adaptive mechanisms on a set of *tunable QoS parameters*.

However, there exists no coherent definition for the term *application-level QoS*. We hold the view that different applications (and even different contexts of using the same application) may have different definitions and preferences on what subset of application parameters should be optimized or kept stable. In this paper, we focus on a set of application-level *critical QoS parameters*, and believe that the ultimate driving force of any adaptations should be the delivery of optimal and stable performance of these critical QoS parameters. The introduction of critical QoS parameters as adaptation objectives leads to the problem of bridging the "gap" between two categories of parameters: critical and tunable QoS parameters. Critical parameters represent adaptation goals, and tunable parameters are the only "knobs" we may use.

In order to provide such coordinated adaptation control to any applications, a "roadmap" of two phases has to be in place. First, we need to capture the inherent characteristics of such applications, since we assume that there exists no prior knowledge with respect to their execution models. This resembles the procedure of *system identification* in control theory. Particularly, we need to quantitatively capture and analyze the relationships between two sets of parameters: critical and tunable. If the critical parameters are not observable on-line, we need to invoke off-line algorithms to capture such relationships. Such algorithms are referred to as *probing algorithms*. Second, based on the knowledge collected in the previous phase, we need to design a framework to actively control the application behavior on-line. In this paper, we focus on the design of the first phase of our roadmap, while presenting a brief overview of the architecture that we have proposed for the second phase.

Our key contributions in this paper are the following. First, we identify a model to characterize the internal structure and behavior of distributed multimedia applications. Particularly, we use a two-level application component model to characterize application structure, and a set of application QoS parameters may be derived from the input and output of each application component. This set is further categorized into *tunable*, *non-critical* and *critical* QoS parameters, depending on their run-time semantics. Second, we propose a novel graph-based relationship model for these parameters, and present a *QoS probing algorithm* to quantitatively capture the run-time relationships between critical and tunable parameters. We have shown that with certain optimizations, the computational complexity of such an algorithm may be reduced from exponential to polynomial. Finally, we present a *hierarchical QoS control* architecture as an entity to actively coordinate and control the adaptation timing and selection of adaptation mechanisms within applications. The architecture is implemented as middleware, referred to as *Agilos*. It derives from the control-theoretical framework proposed in our previous work [16] to bridge the gap between monitoring resource variations and tuning application parameters. The *active* role of our design is illustrated in Figure 1. This approach is advantageous over existing middleware-based passive adaptation approaches [15, 28] in the sense that the middleware-based architecture is clearly aware of what the application needs, i.e., what subset of parameters are critical.

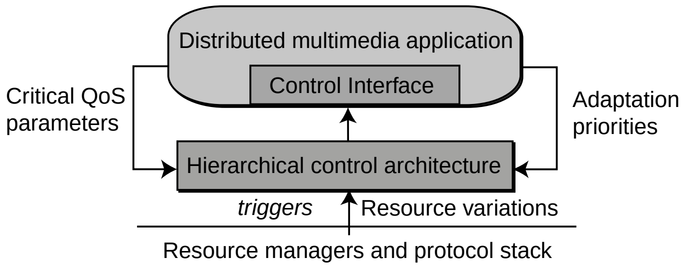

Distributed multimedia application
Control Interface
Critical QoS parameters
Adaptation priorities
Hierarchical control architecture
triggers
Resource variations
Resource managers and protocol stack

Figure 1: The Role of Hierarchical Control Architecture

As a case study, we evaluate the hierarchical control architecture by studying its impact on *Omnitrack*, an *omni-directional visual tracking application*. *Omnitrack* is a distributed multimedia application that streams live video from remote camera servers (with different viewing angles to the objects) to the clients, where moving objects are tracked with local tracking algorithms named *trackers*. The critical parameter is the *tracking precision*, defined as the precision with which trackers follow moving objects. We will use examples from this application throughout the paper.

The remainder of this paper is organized as follows. Section 2 presents the two-level application component model, as well as categorization of application QoS parameters. Section 3 describes our graph-based model to characterize relationships between different categories of application parameters, presents a QoS probing algorithm, and analyzes its computational complexity. Section 4 presents the hierarchical control architecture, using analytical results from previous work [16] and the probing algorithm from Section 3. Section 5 uses *Omnitrack* as our case study, and show that our architecture is effective in preserving the quality of critical QoS parameters. Sections 6 and 7 review related work and conclude the paper.

2 Application Model and Parameter Categorization

In this paper, we consider a generic two-level application component model to characterize the structure of applications. The original concepts in this model are borrowed from previous work of the EPIQ project [13], where flexible applications are divided into *tasks*, which form a *directed acyclic graph* (DAG) representing their input-output relationships. We adopt this representation to view a collection of interconnected *application components* on a single host. Beyond a single end host, we group the entire distributed application into *clients* and *services*, with each client or service running in one of the networked hosts. The collection of clients and services forms another directed graph (not necessarily acyclic) to represent their service provisioning relationships. This graph is referred to as the *service provisioning graph*. Naturally, services may be clients themselves to other services. Figure 2(a) illustrates such a two-level characterization. In the remainder of this paper, the first level is referred to as the *component level*, and the second level as the *service level*.

Furthermore, we consider a scheme of categorizing application QoS parameters. For this purpose, we focus on a single application component. We assume that this component accepts input with a QoS level $\mathbf{Q}^{in}$ and generates output with QoS level $\mathbf{Q}^{out}$, both of which are *vectors* of *application QoS parameter* values. In order to process input and generate output, a specific amount of resources $\mathbf{R}$ is required, which is a vector of resource types. Figure 2(b) illustrates such characterization in terms of QoS parameters and

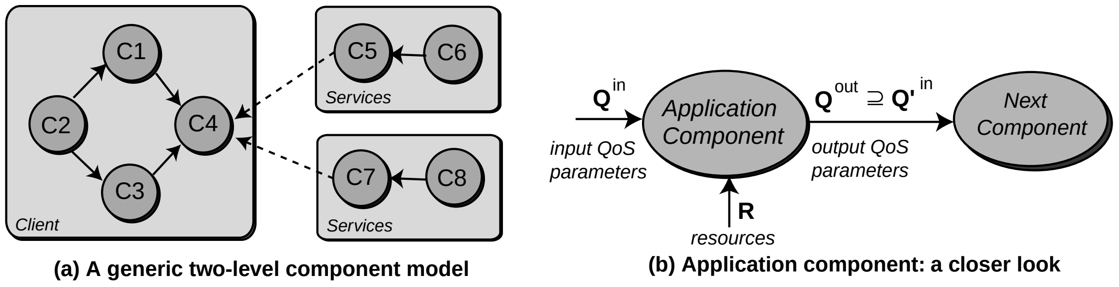


Figure 2: A Two-level Component Model for Applications

resources. At the component level, if we view any subgraph of our DAG representation as a "black box", its input vector $\mathbf{Q}^{in}$ and output vector $\mathbf{Q}^{out}$ may also be straightforwardly derived.

Formally, we define the vectors $\mathbf{Q}^{in}$, $\mathbf{Q}^{out}$ and $\mathbf{R}$ as follows:

$$\begin{aligned} \mathbf{R} &= [R_1, R_2, \ldots, R_m]^T \\ \mathbf{Q}^{in} &= [p_1^{in}, p_2^{in}, \ldots, p_n^{in}]^T \\ \mathbf{Q}^{out} &= [p_1^{out}, p_2^{out}, \ldots, p_k^{out}]^T \end{aligned} \tag{1}$$

where $R_1, R_2, \ldots, R_m$ are the required resource types and measured with their respective units. In the *Omnitrack* example, $m = 2$, and R_{cpu} is measured with CPU load percentage, while R_{net} is measured with bytes per second.

We assume the vector of all application QoS parameters $Q = [Q^{in^T}, Q^{out^T}]^T$. We further classify the parameters in the vector Q into three distinct categories:

- **Critical QoS parameters.** We assume all critical parameters, which are elements in the vector $\mathbf{Q_{cr}}^{out}$, belong to the set of output QoS parameters. Let $\mathbf{Q_{cr}}^{out} = [p_{i_1}^{out}, p_{i_2}^{out}, \ldots, p_{i_l}^{out}]^T$, while $1 \leq i_1, i_2, \ldots, i_l \leq k$. The objective of adaptation is focused on these critical QoS parameters.

- **Tunable QoS parameters.** Without loss of generality[1], we assume that all input QoS parameters are *tunable*.

- **Non-critical QoS parameters.** It follows that any parameters in the vector $\mathbf{Q}^{out}$ that do not belong to the category of *critical QoS parameters* are non-critical.

[1]If an input QoS parameter is not tunable, we may view it as an output parameter instead.

For simplicity of notations, we redefine $\mathbf{Q}^{in}$ and ${\mathbf{Q_{cr}}}^{out}$ as follows, removing extra superscripts and renumbering the subscripts:

$$\begin{aligned}\mathbf{Q}^{in} &= [p_1, p_2, \dots, p_n]^T \\ {\mathbf{Q_{cr}}}^{out} &= [p_1^{cr}, p_2^{cr}, \dots, p_l^{cr}]^T\end{aligned} \tag{2}$$

Since all tunable QoS parameters are input parameters and all critical QoS parameters belong to the set of output QoS parameters, it is natural that when the tunable parameters are actively controlled, the critical parameters will consequently change. In this case, the critical parameters are claimed to be *dependent* on the tunable parameters. In addition, in most applications critical parameters are also *dependent* on a certain subset of resource types, since when resource availability changes, they effectively change the observed values of critical parameters.

In order to characterize the complete set of parameters that critical parameters depend on, we define a new vector $\mathbf{P}^{in}$ to include the relevant resource types:

$$\mathbf{P}^{in} = [{\mathbf{Q}^{in}}^T, \mathbf{R}^T]^T = [p_1, p_2, \dots, p_n, R_1, R_2, \dots, R_m]^T \tag{3}$$

For coherent notations, we redefine $p_{n+i} = R_i$, $1 \leq i \leq m$, so that:

$$\mathbf{P}^{in} = [p_1, p_2, \dots, p_n, p_{n+1}, \dots, p_{n+m}]^T \tag{4}$$

3 Modeling and Probing Application QoS Parameters

Our adaptation objective is to control the tunable parameters to achieve stability and optimality with respect to critical QoS parameters. However, in some complex multimedia applications, critical QoS parameters may not be directly observable on-line. One example is the *tracking precision* in the Omnitrack application: the tracking precision is critical to overall application QoS, but it is impossible to dynamically monitor the precision of tracking live moving objects, since their positions and movements are not known *a priori*. In these applications, we need to capture the relationships between critical and tunable QoS parameters before the application goes on-line.

In this section, we propose a QoS probing algorithm that bridges the gap between the set of critical and dependent parameters by capturing the dependency relationship between ${\mathbf{Q_{cr}}}^{out}$ and $\mathbf{P}^{in}$. For this purpose, we first propose a generic *Bipartite Graph Model* to characterize such relationships between critical and dependent parameters (i.e. tunable parameters and resources). However, we are able to show that the computational complexity of the QoS probing algorithm derived from this model is exponential.

In order to solve this problem, we present an optimization method to extend the generic Bipartite Graph Model to a Hierarchical Graph Model. Based on such an improved model, we present a hierarchical QoS probing algorithm. Finally, we derive an upper bound for its computational complexity, and prove that the computation may be achieved in polynomial time.

3.1 Bipartite Graph Model for Application QoS Parameters

The brute-force way of designing a QoS probing algorithm is to discover the relationship between any particular critical parameter and all its dependent parameters. For this purpose, we need to construct a *Directed Bipartite Graph*, with all elements in ${\mathbf{Q_{cr}}}^{out}$ forming one partition of the graph, and all elements in $\mathbf{P}^{in}$ forming the opposite partition. If node p_i^{cr} depends on p_j, $1 \leq i \leq l, 1 \leq j \leq n+m$, there exists a directed edge from node p_i^{cr} to node p_j in the directed bipartite graph.

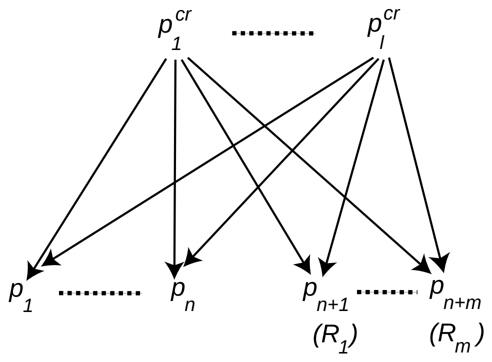


Figure 3: Bipartite Graph Model for Application QoS Parameters

The objective of a QoS probing algorithm is to tune the parameters in $\mathbf{P}^{in}$ and observe the values of critical parameters. As an initial step, we assume that each critical parameter in ${\mathbf{Q_{cr}}}^{out}$ depends on all the parameters in $\mathbf{P}^{in}$. Figure 3 shows an example. Obviously, the subgraph consisting of one critical parameter and all parameters in $\mathbf{P}^{in}$ is a *two-level tree*, with the selected critical parameter as root and all the parameters in $\mathbf{P}^{in}$ as leaves.

The critical step in the QoS probing algorithm is to discover the relationship between dependent nodes.

For this purpose, 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 value beyond this range is either not possible or not meaningful. For example, the *frame rate* may vary in the range of $[1, 30]$ (in frames per second). Hence, the dependency between each critical parameter and their dependent parameters can be characterized by f_i, defined as:

$$\begin{aligned} p_i^{cr} &= f_i(p_1, p_2, \ldots, p_{n+m}) \\ \{p_k\}_{min} &\leq p_k(t) \leq \{p_k\}_{max} \\ k = 1, 2, \ldots, n+m&, i = 1, 2, \ldots, l \end{aligned} \tag{5}$$

```
for each critical parameter $p_i^{cr} (1 \leq i \leq l)$:
    for $k_1 = \{p_1\}_{min}$ to $\{p_1\}_{max}$ step $\{p_1\}_{increment}$
        for $k_2 = \{p_2\}_{min}$ to $\{p_2\}_{max}$ step $\{p_2\}_{increment}$
            ...
            for $k_{n+m} = \{p_{n+m}\}_{min}$ to $\{p_{n+m}\}_{max}$ step $\{p_{n+m}\}_{increment}$
                log the observed value for $p_i^{cr}$
```

Figure 4: The QoS Probing Algorithm based on the Bipartite Graph Model

With the Bipartite Graph Model, the algorithm for computing the dependency relationship between each critical parameter p_i^{cr} and the parameters in $\mathbf{P}^{in}$ is straightforward, as shown in Figure 4. For each parameter $p_j (1 \leq j \leq n + m)$, it starts at value $\{p_j\}_{min}$ and increases by $\{p_j\}_{increment}$ until it reaches $\{p_j\}_{max}$. Hence the possible number of distinct values for p_j is

$$N_{p_j} = \frac{\{p_j\}_{max} - \{p_j\}_{min}}{\{p_j\}_{increment}} \tag{6}$$

There may be one complication in the algorithm shown in Figure 4. Although resource parameters vary when the application is running on-line, it may not be feasible to control their values during profiling runs. In this case, if a resource parameter is one of the dependent nodes, one of the solutions is to control the values of the critical parameter p_i^{cr} instead and observe the changes of the resource parameters $p_j, n+1 \leq j \leq n+m$. Similar to the tunable parameters, p_i^{cr} starts from $\{p_i^{cr}\}_{min}$ and increases by $\{p_i^{cr}\}_{increment}$ until it reaches $\{p_i^{cr}\}_{max}$, and the number of possible values $N_{p_i^{cr}}$ follows the same definition as in Equation 6. However, this solution is not feasible if p_i^{cr} is not tunable itself. To summarize, we need to assume that either p_i^{cr} or its dependent resource parameters need to be tunable in the profiling runs.

Let $N_{max} = \max\{N_{p_1}, N_{p_2}, \ldots, N_{p_{n+m}}\}$. The computational complexity is $O(l * {N_{max}}^{(n+m)})$. This shows that when the number of tunable parameters and resource types increases, computation increases exponentially.

3.2 Hierarchical Model for Application QoS Parameters

Typically, one critical parameter only depends on a limited number of tunable QoS parameters and resources. Furthermore, we have observed that if two critical parameters depend on a common set of parameters in $\mathbf{P}^{in}$, they may have similar dependency relationships with such a common set. If this similarity of relationships can be captured and then shared by both critical parameters, computational complexity may be reduced dramatically.

Along this line, we introduce a set of *intermediate QoS parameters* that critical parameters depend on. These intermediate QoS parameters may be either non-critical QoS parameters in the output QoS, or other internal parameters within the application component. In addition, they depend on the parameters in $\mathbf{P}^{in}$. To maximize the dependency relationship sharing among critical parameters, intermediate nodes are organized hierarchically. Figure 5 shows an instance of the possible dependency relationships.

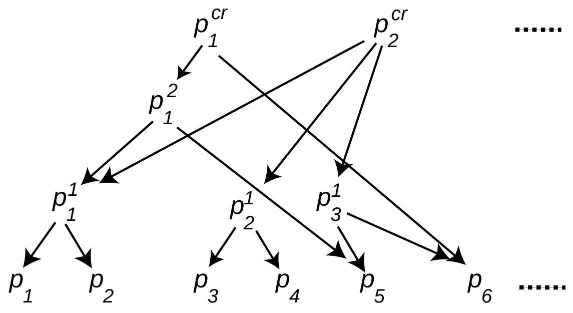


Figure 5: Hierarchical Model for Application QoS Parameters

The hierarchy in Figure 5 presents the following properties:

- The subgraph composed of a critical parameter node p_i^{cr} $(1 \leq i \leq l)$ and all downstream nodes is essentially a multi-level directed tree, with root as p_i^{cr} and a subset of parameters in $\mathbf{P}^{in}$ as leaves. We refer to such a tree as Tree(p_i^{cr}), and the hierarchical graph is $\cup\{\text{Tree}(p_i^{cr})\}$.
- Two critical parameters share dependency on some parameters by sharing a subtree. For example, in Figure 5, the subtree with source at p_1^1 is shared by both critical parameters p_1^{cr} and p_2^{cr}.

- All intermediate nodes have at least outdegree 2, meaning that an intermediate node should at least depend on two child nodes. If such node exists that has an one-to-one relationship with its child node, it only introduces redundant computation. Therefore, this node will be removed and replaced by a directed link from its parent node to its only child node. For example, Figure 6(a) shows that the intermediate node p_y with outdegree 1 is removed, and the links $p_z \rightarrow p_y$ and $p_y \rightarrow p_x$ are consolidated to a direct link $p_z \rightarrow p_x$.

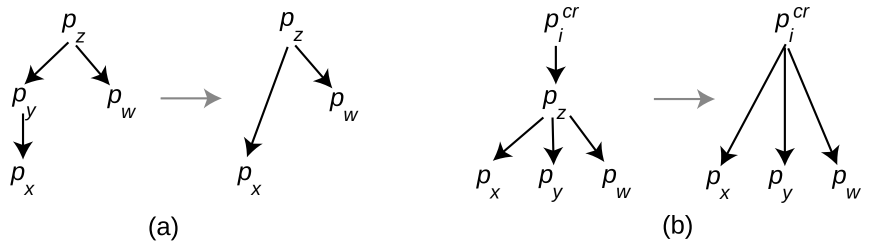


Figure 6: Redundant node removal

- All the critical parameters $p_i^{cr}, 1 \leq i \leq l$, should also have at least outdegree 2. If p_i^{cr} has only one child, this child will be removed and its children become children of p_i^{cr}. Figure 6(b) shows that the only child p_z of p_i^{cr} is removed and the children (p_x, p_y, p_w) of p_z turn into children of p_i^{cr}.

- The notation p_i^j denotes that the node is at level j, and is the i^{th} node at this level. We calculate the level of a node by counting from bottom, with leaves (parameters in $\mathbf{P}^{in}$) at level 0, that is, $p_i = p_i^0, i = 1, 2, \ldots, n+m$.

  1. If a node only depends on leaves, its level is 1. For example, p_1^1 only depends on p_1 and p_2, hence its level is 1.

  2. If a node has children, its level number is defined to be one greater than the maximum level number of its child nodes. For example, p_1^2 depends on p_1^1 (level 1) and p_5 (level 0), its level is thus 2.

As an example, Figure 7 shows the hierarchical graph[2] for Omnitrack. Omnitrack is an application with *tracking precision* as the only critical parameter, it becomes the root of its multi-level tree.

Assume the parent node p_i^j depends on k child nodes $p_{i_1}^{j_1}$, $p_{i_2}^{j_2}$, ..., $p_{i_k}^{j_k}$. The dependency can thus be characterized by a function $f_{p_i^j, p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}}$, defined as:

[2]Since Omnitrack has only one critical parameter, the hierarchical graph essentially becomes a multi-level tree.

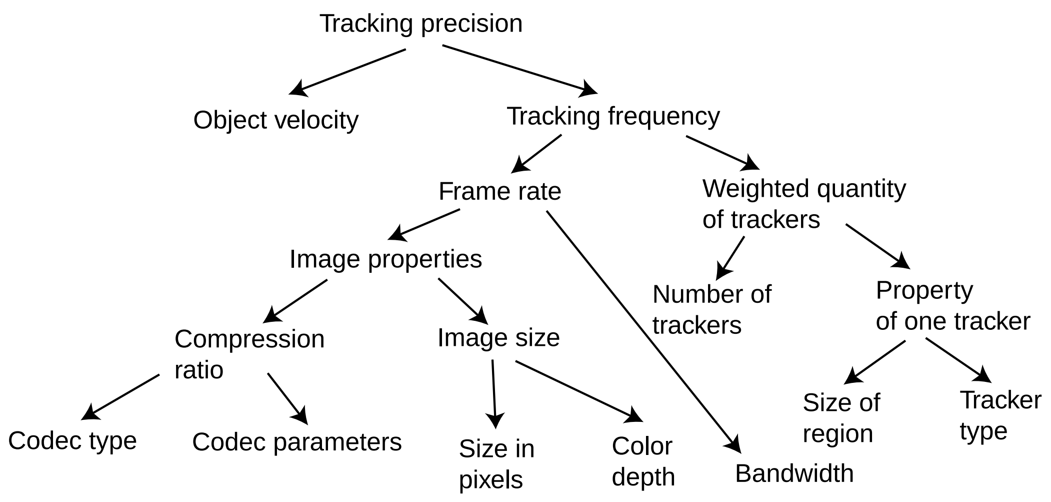


Figure 7: Hierarchical Graph for Omnitrack

$$\begin{aligned} p_i^j = f_{p_i^j, p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}}(p_{i_1}^{j_1}, p_{i_2}^{j_2}, \ldots, p_{i_k}^{j_k}) \\ \{p_{i_s}^{j_s}\}_{min} \leq p_{i_s}^{j_s}(t) \leq \{p_{i_s}^{j_s}\}_{max} \\ s = 1, 2, \ldots, k \end{aligned} \tag{7}$$

A hierarchical QoS probing algorithm is proposed as shown in Figure 8. For each critical parameter, we calculate all the dependency functions between one non-leaf node and its dependent child nodes. The calculation is performed from bottom to top of the hierarchical graph.

Similar to the definition in Equation 6 for the number of steps with respect to tunable QoS parameters, we define the number of steps for p_k^j as follows:

$$N_{p_k^j} = \frac{\{p_k^j\}_{max} - \{p_k^j\}_{min}}{\{p_k^j\}_{increment}} \tag{8}$$

We have two additional notes:

- As we have mentioned earlier in the probing algorithm for our Bipartite Graph Model, if a resource parameter is one of the dependent nodes, we control the values of the parent node instead and observe the changes of the resource parameter. The variable *list* in the algorithm keeps track of the parameters to be observed.

```
for each critical parameter $p_i^{cr}$ and its associated Tree$(p_i^{cr})$
    for level j = 1, j $\leq$ depth(Tree$(p_i^{cr})$), j++ (from bottom to top level)
        for each node $p_k^j$ at level j
            $d := d(p_k^j)$ // outdegree
            assume its children are $p_{k_1}^{j_1}, p_{k_2}^{j_2}, \ldots, p_{k_d}^{j_d}$
            list := $\{p_k^j\}$ // observing list
            for each child $p_{k_w}^{j_w}$, $1 \leq w \leq d$
                if $p_{k_w}^{j_w}$ is a leaf and is not a resource parameter then
                    Value$(p_{k_w}^{j_w})$ iterates from $\{p_{k_w}^{j_w}\}_{min}$ to $\{p_{k_w}^{j_w}\}_{max}$, step $\{p_{k_w}^{j_w}\}_{increment}$
                else if $p_{k_w}^{j_w}$ is a leaf and is a resource parameter then
                    if $p_k^j \in$ list then
                        list := list - $\{p_k^j\}$
                        list := list + $\{p_{k_w}^{j_w}\}$
                        Value$(p_k^j)$ iterates from $\{p_k^j\}_{min}$ to $\{p_k^j\}_{max}$, step $\{p_k^j\}_{increment}$
                    else
                        list := list + $\{p_{k_w}^{j_w}\}$
                else // non-leaf node
                search the dependency log and find the sorted value set for $p_{k_w}^{j_w}$
                Value$({p_{k_w}}^{j_w})$ iterates elements in the sorted value set for $p_{k_w}^{j_w}$
            log observed values for parameters in list
```

Figure 8: The Hierarchical QoS Probing Algorithm

- If a child node p_k^j is not a leaf, it must depend on some other lower level nodes, and the dependency between them should have already been calculated and saved in the log. The calculated value set for this node should be within the range $[\{p_k^j\}_{min}, \{p_k^j\}_{max}]$, and each value is rounded to the nearest discrete value $\{p_k^j\}_{min} + s * \{p_k^j\}_{increment}, 0 \leq s \leq N_{p_k^j}$. The sample values for this child node will be retrieved from the log.

According to the algorithm, it is obvious that if a node in Tree(p_i^{cr}) and a node in Tree(p_j^{cr}) share a subtree, the calculation for the subtree may be done only once for both trees and the dependency function can thus be shared by both nodes. For example, in Figure 5, the subtree rooted at p_1^1 exists in both Tree(p_1^{cr}) and Tree(p_2^{cr}). After the dependency function $f_{p_1^1, p_1, p_2}$ is determined, it will be shared by parent nodes p_1^2 in Tree(p_1^{cr}) and p_2^{cr} in Tree(p_2^{cr}).

As an example, in the case of *Omnitrack* shown in Figure 7, Figure 9 illustrates the dependency relationships between the critical parameter, *tracking precision*, and the values of *object velocity* and *tracking frequency*. The output of the inner loop (by only tuning tracking frequency) is shown as bold dotted lines.

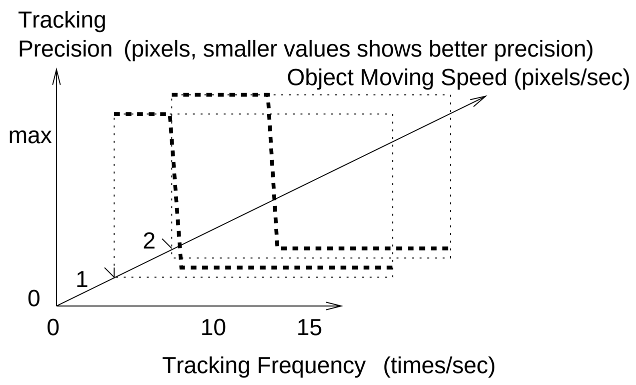


Figure 9: Probing Algorithm in a Qualprobe: An Example

3.3 Computational Complexity Analysis for the Hierarchical QoS Probing Algorithm

In this section, we analyze the computational complexity of this hierarchical QoS probing algorithm. The algorithm is for the hierarchical graph $\cup\{\text{Tree}(p_i^{cr})\}$; however, the computation is sequentially performed for each $\text{Tree}(p_i^{cr})$, $i = 1, 2, \ldots, l$, therefore we first consider a single tree $\text{Tree}(p_i^{cr})$. It is obvious that the complexity depends on the number of non-leaf nodes in the tree. We first demonstrate that the number of non-leaf nodes is bounded by the number of leaves in $\text{Tree}(p_i^{cr})$.

Let $W(T)$ and $L(T)$ denote the number of non-leaf nodes and number of leaves in a Tree T, respectively. Let $h(T)$ denote the depth of the tree T.

Lemma: Given a directed tree T. If outdegree for each non-leaf node is no less than 2, then $W(T) < L(T)$.

Proof. We proceed by induction as follows:

Basis

if $h(T) = 2$, then T is a two-level tree. The only non-leaf node is the root itself, hence $W(T) = 1$. Since outdegree for a non-leaf node is no less than 2, the root has at least two children which are all leaves, i.e. $L(T) \geq 2$. Apparently $W(T) < L(T)$.

Induction

Assume for any tree T of depth $d = h(T)$, if outdegree for each non-leaf node is no less than 2, then $W(T) < L(T)$. We have to show that a tree T' of depth $d + 1 = h(T')$ has the same property.

- By removing leaves from T', we have a reduced tree T which is of depth d. Obviously all the nodes in T are internal nodes in T', and we have $W(T') = W(T) + L(T)$. Since outdegree for each non-leaf nodes in T' is no less than 2, and T is part of T', therefore outdegree for each non-leaf nodes in T is no less than 2. By induction hypothesis, we know that $W(T) < L(T)$.

- Each non-leaf node in T' has at least two children, so for each leaf in T, which is actually an internal node in T', has at least two children that belong to leaves in T'. Therefore we have $2 * L(T) \leq L(T')$.

As a result, we have $W(T') = W(T) + L(T) < L(T) + L(T) = 2 * L(T) \leq L(T')$, that is, $W(T') < L(T)$. This completes the proof. **QED.**

For each node p_k^j in Tree(p_i^{cr}), we assume that its outdegree is $d(p_k^j)$. Let $d_{max}^i = \max\{d(p_k^j)\}$, the maximum outdegree of the nodes in Tree(p_i^{cr}). Let $d_{max} = \max\{d_{max}^i\}$, $1 \leq i \leq l$, the maximum outdegree of the nodes in $\cup\{$Tree$(p_i^{cr})\}$.

In addition, let $N_{max} = \max\{N_{p_k^j}\}$.

Theorem: The computational complexity of the hierarchical QoS probing algorithm is bounded by $O(l * (n+m) * (N_{max})^{d_{max}})$. l is the number of critical parameters, n is the number of tunable parameters and m is the number of resource parameters.

Proof. For Tree(p_i^{cr}) in the hierarchical graph $\cup\{$Tree$(p_i^{cr})\}$, we count the number of non-leaf nodes in this tree. According to the lemma, the number of non-leaf nodes is less than the number of leaves, which is $n+m$. Therefore, the total time of profiling one critical parameter is less than $(n+m) * (N_{max})^{d_{max}}$. For l critical parameters, the upper bound is $O(l * (n+m) * (N_{max})^{d_{max}})$. This is obviously polynomial. **QED.**

4 The Hierarchical Adaptive QoS Control Architecture

In this section, we present a brief overview of our QoS control architecture, which focuses on goals in the second phase of our roadmap shown in Section 1. In order to accomplish the objective of bridging the gap between resource-level adaptations and application requirements, we propose three *control tiers* in the hierarchical architecture. Figure 10 illustrates the architecture, along with the example application *Omnitrack*.

1 - **The resource adaptation tier.** In this tier, *adaptor* and *observer* components are included for adapting to resource variations, particularly *triggering events* that signal significant changes in resource

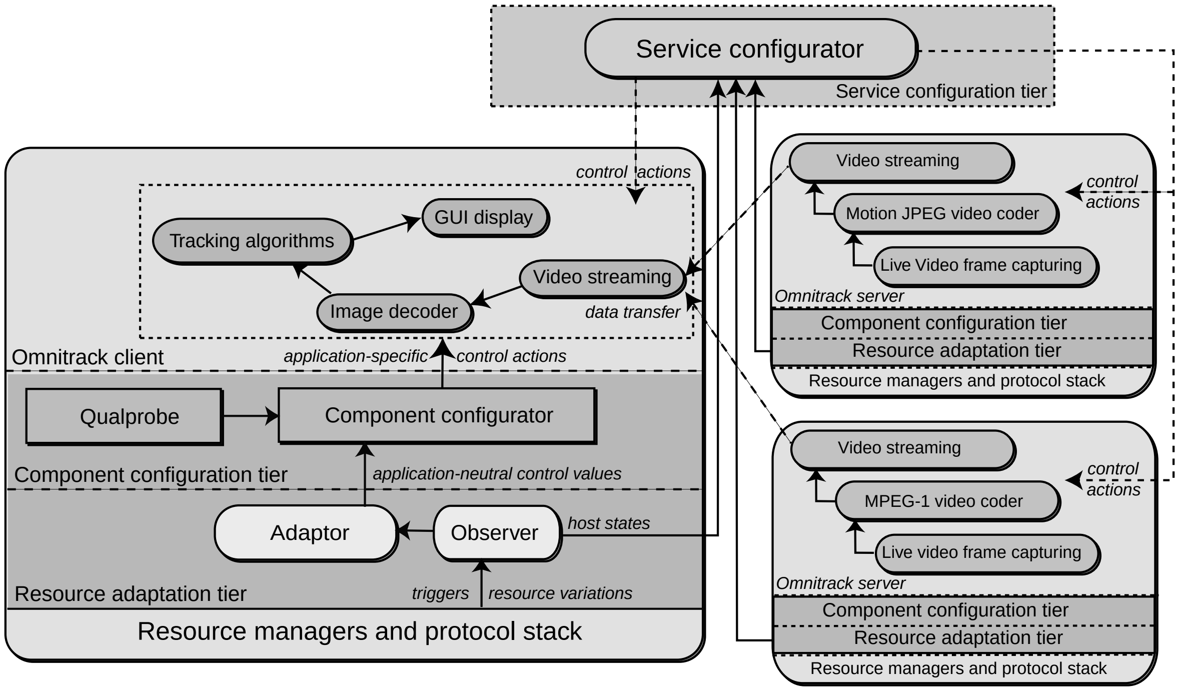


Figure 10: Components in the Hierarchical Control Architecture

availability. It is the tier that closely monitors changes in the distributed system, by directly probing either resource managers (such as schedulers or protocol stacks), or the applications under control.

2 - **The component configuration tier.** Based on the input from the resource adaptation tier, *component configurators* are responsible for making decisions on when and what adaptive mechanisms are to be invoked, based on a set of *adaptation rules* that are application-specific. The goal of adaptation is to provide optimal QoS with respect to critical parameters. In addition, a *qualprobe* component is introduced in this tier, in order to implement the QoS probing algorithm presented in Section 3. This component is used to discover the relationship between critical and tunable parameters, so that *adaptation rules* may be specified.

3 - **The service configuration tier.** A complex multimedia application frequently involves clients using various on-demand multimedia services, intermediate filtering services (e.g. transcoders) and optional third-party value-added services (e.g. watermarking). In this tier, a *service configurator* controls and coordinates distributed adaptation decisions involving a topological change in such service provisioning relationships.

Since the goal of architecture is to actively control configurable applications at the component or service levels, a feedback control loop should be formed to steer applications so that they react to resource fluctuations. Such a control loop is formed with an integrated framework including the adaptors, observers, component and service configurators, shown in Figure 11. As the feedback loop is formed, it is natural to adopt a hybrid *control-theoretical* approach to design algorithms in the components.

Our previous work [16] has presented mathematical reasoning for our linear control-theoretical algorithms used in resource adaptors, as well as the design of component and service configurators with fuzzy control theory, where the *adaptation rules* are specified based on fuzzy logic. We have shown that the final control decisions are both globally fair to other concurrent applications, and particularly tailored to the application's critical QoS parameters. We briefly review the design of each control tier in the following sections.

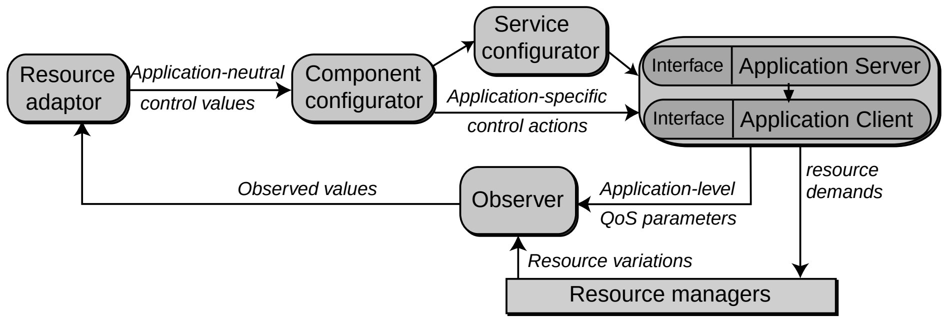


Figure 11: Viewing the Hierarchical Control Architecture as a Control Loop

4.1 Resource Adaptation Tier: Resource Adaptors

There are several design objectives of the resource adaptors. First, they react to variations in resource availability, where the output is governed by standard linear control algorithms, such as PID control. Second, with respect to resources, they maintain fairness among all concurrent applications within the same end host. Third, the control algorithm is parameterized and highly configurable so that a variety of different *adaptation responsiveness* can be achieved. The resources adaptors and observers are designed based on our component model for applications. We have customized the standard PID control algorithm so that it is used to respond to variations in observed variations in resources. In our previous work [16], we have proved that the output of such a customized control algorithm is fair with respect to end-host resources (e.g. CPU) to all concurrent applications according to their weights, with configurable adaptation responsiveness. Figure 12 shows the resource adaptation tier.

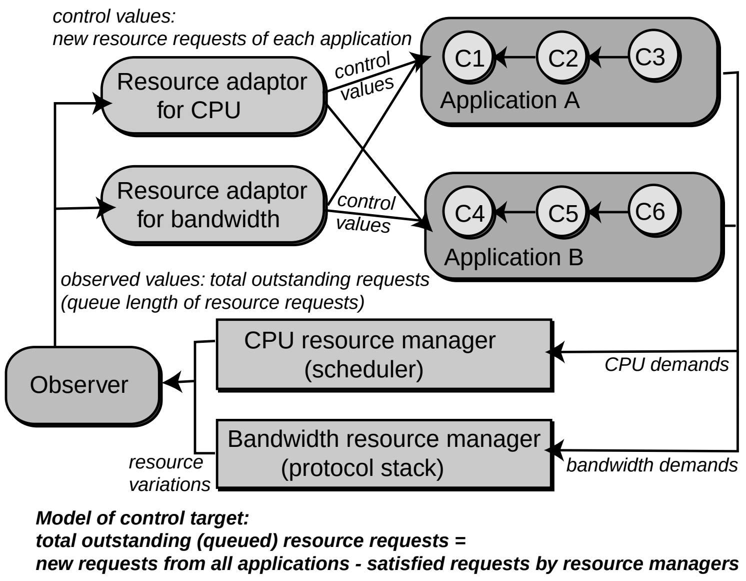

control values:
new resource requests of each application
Resource adaptor for CPU
control values
Resource adaptor for bandwidth
control values
C1
C2
C3
Application A
C4
C5
C6
Application B
observed values: total outstanding requests (queue length of resource requests)
CPU resource manager (scheduler)
CPU demands
Observer
Bandwidth resource manager (protocol stack)
bandwidth demands
resource variations
Model of control target:
total outstanding (queued) resource requests =
new requests from all applications - satisfied requests by resource managers

Figure 12: The Resource Adaptation Tier

Resource adaptors are neutral to applications and specific to resource types, such as CPU and network bandwidth. Such adaptors control all concurrent applications sharing the same end-host resource in an end host. Two properties in an adaptor are configurable: the *weights* of importance for each application, and *adaptation responsiveness*. Weights assigned to applications represent the relative importance of applications sharing the same pool of resources, i.e., application with a higher weight is allowed more resources following the weighted max-min fairness property. In addition, *adaptation responsiveness* expresses the sensitivity of reacting to variations. To ensure fairness, it is configurable on a system-wide basis for all applications.

4.2 Component Configuration Tier: Component Configurators

The component configuration tier focuses on preferences and requirements specific to individual applications on the same end host. It is responsible for adaptations at the component level. *Component configurators* are responsible for making decisions with respect to which adaptive mechanism is to be activated, and when the adaptation should occur. *Qualprobe* components implement the QoS probing algorithm presented in Section 3. It "learns" application behavior by run-time probing techniques, so that better strategies may be built into component configurators by specifying proper adaptation rules.

4.2.1 Component Configurators

Component configurators determine discrete control actions based on application-specific needs and control values produced by resource adaptors. In previous work [16], a *Fuzzy Control Model* was proposed to design the component configurator. The adoption of fuzzy logic is justified by the observation that multiple reconfiguration and parameter-tuning options span different domains, and that the controllable regions and variables within the application are discrete and non-linear. In such a scenario, fuzzy logic allows the specification of such a decision-making process with a small number of *fuzzy rules*. The non-linearity of the fuzzy controller matches naturally to the complexities brought by having multiple adaptation choices.

The Fuzzy Control Model utilizes fuzzy logic to express application-specific adaptation choices in a configurable *rule base*, which “fuels” a generic *fuzzy inference engine* to derive the exact control decisions. It contains two parts: *linguistic rules* consisting of a set of linguistic variables and values, and *membership functions* for linguistic values. A typical linguistic rule is:

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

Such a rule specifies that if the CPU adaptor allows high CPU utilization and the bandwidth adaptor allows only low network throughput, then reconfigure the application to activate the video encoder component. A typical membership function for a linguistic value such as `high` can be expressed as:

```
class high (4) is  80  90  100  100
```

This represents four deterministic points of a trapezoid-shaped membership function shown in Figure 13, with an output in the range $[0, 1]$, representing the possibility that adaptation should happen.

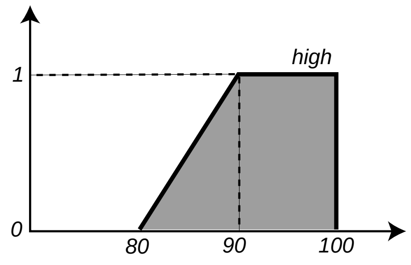


Figure 13: A typical membership function defined in the rule base

The application-neutral output of the resource adaptors is piped into the component configurator, fuzzified as input to the inference engine based on its rule base. Any output from the inference engine is then used to directly control adaptation paths in the applications. Figure 14 shows such a design within the configurator. In the example shown, the application-neutral output of the bandwidth adaptor may be *"decrease new bandwidth resource requests to* **x***"*, and the output of the component configurator may be *"activate the video encoder MPEG-2"*.

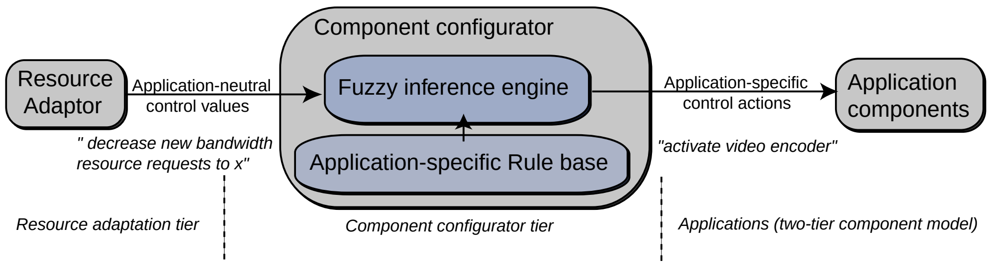

Component configurator
Resource Adaptor
Application-neutral control values
Fuzzy inference engine
Application-specific control actions
Application components
" decrease new bandwidth resource requests to x"
Application-specific Rule base
"activate video encoder"
Resource adaptation tier
Component configurator tier
Applications (two-tier component model)

Figure 14: The Design of Component Configurator

4.2.2 Qualprobe

The *qualprobe* is a QoS probing and profiling service that implements the probing algorithm presented in Section 3. It maintains three responsibilities: First, it allows applications to specify its critical QoS parameters to the component configurator, along with its relationship with other controllable QoS parameters. Second, as a QoS probing service, it actively probes application components in controlled benchmarking runs, in order to obtain detailed QoS mapping functions among application QoS parameters. Third, as a QoS profiling service, it logs all probing results in QoS profiles. The actual specification of the rule base in component configurators is based on these profiles.

The qualprobe is designed to be application-neutral, thus we require that all critical and intermediate QoS parameters are *observable*. This demands that their run-time values may be measured. 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 a qualprobe retrieves these values from the Property Service when the measurements are invoked.

4.3 Service Configuration Tier: Service Configurators

As shown in Figure 10, the service configuration tier includes a centralized service configurator, which adds support for distributed adaptation mechanisms that effectively remove the limitation that adaptation can only be performed on a single end host. If we examine the two-level component model in Section 2, the service configurator operates and focuses in the service level of the model, by reconfiguring service provisioning relationships in a complex application.

The objective of such a distributed adaptation scheme is to optimally exploit available services across the network, taking into account their *properties* that are not considered in the component configuration level. By reconfiguring the mapping among services and clients, the service configurator seeks to find an optimal service configuration for the application. At run-time, the service configurator is able to make topological changes in the service level of the component model. For example, in *Omnitrack*, when bandwidth becomes insufficient, the best adaptation for a particular client may be to switch to a server that serves a different video codec format that requires less bandwidth. The service configurator decides when such adaptation should happen.

The input to the service configurator is the current states on each host composing the service provisioning graph. These states include resource availability, user preferences and application parameters, which are obtained by the observer. The output of the service configurator is a set of control actions that controls individual hosts required to change to topology of the service provisioning graph. Figure 15 shows the role of a service configurator, along with the topology of the service provisioning graph. Both clients and services are involved in the graph.

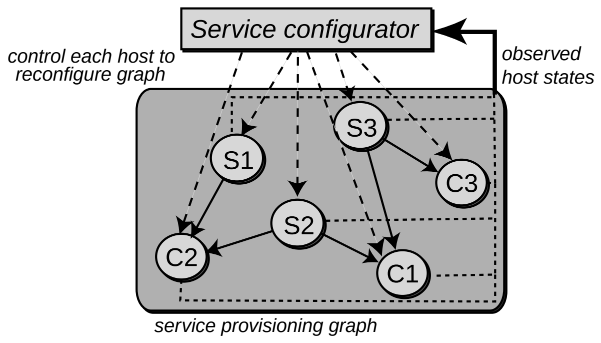

Service configurator
control each host to reconfigure graph
observed host states
S1
S2
S3
C1
C2
C3
service provisioning graph

Figure 15: The Role of Service Configurator

Activities in the service configuration tier are transparent to the clients. At any given time, each client is being serviced by a set of servers that the service configurator decides to be most appropriate.

The internal structure of the service configurator is shown in Figure 16. It maintains a *state table* for each of the clients and services. Between these is a *service provisioning graph* which maintains the current topology of service provisioning relationships. This graph is updated by the *central processing* module when individual hosts are controlled to change the topology.

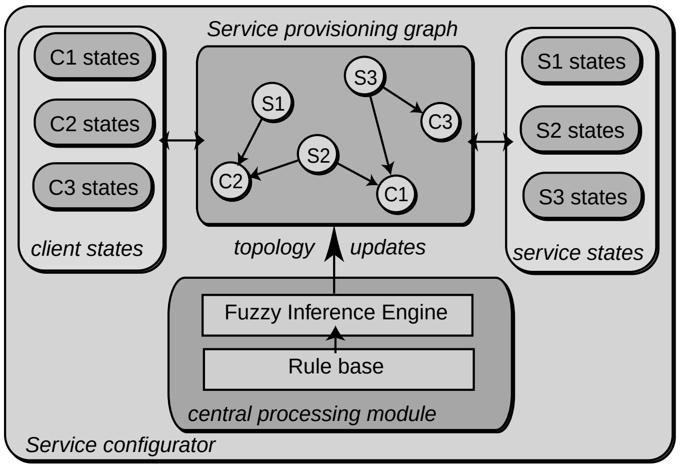


Figure 16: Structure of Service Configurator

The *central processing* module is responsible for computing required topological changes in the service provisioning graph, if any. We have adopted the fuzzy logic based *fuzzy inference engine* for the purpose of processing input states from hosts and generating a control decision. As in the *component configurator*, such an inference process is based on a *rule base* and states of individual hosts in the service provisioning graph. In *Omnitrack*, an example of a *rule* in the rule base is as follows:

```
if (service_load is low) and (service_angle is close) then service_ranking is high;
```

In this example, `service_angle` is a dynamically generated value derived from the state of a particular client and service, including the actual angle of the client's desired view, the view that the service offers and the difference between them. On the other hand, `service_load` is the monitored CPU load currently observed at the service components on a server. The better a server matches the criteria specified, the higher the ranking of a service will be. The highest ranked service will be selected to serve the client. Additional states can be encoded into each rule to create a more sophisticated rule base.

5 Case Study: Omnitrack

We have deployed *Omnitrack* as an example application under the control of our hierarchical control architecture. The architecture is implemented in C++ as a set of *middleware* components, which we later refer to as *Agilos* (**Agil**e Q**oS**). *Agilos* and *Omnitrack* are both implemented to execute on two dual Pentium Pro 200

Mhz PCs, one Pentium MMX 200Mhz PC, and one Pentium II 400Mhz PC. All PCs run Windows NT 4.0 as their operating systems. *Agilos* middleware components rely on ORBacus as the CORBA infrastructure for communicating with the applications. We have deployed all tracking clients and servers over a 10Mbps standard Ethernet segment.

In order to simulate the network bandwidth fluctuations, we plug in a throughput simulator to simulate a network through multiple 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 satisfaction criterion is to maintain the stability of its overall tracking precision.

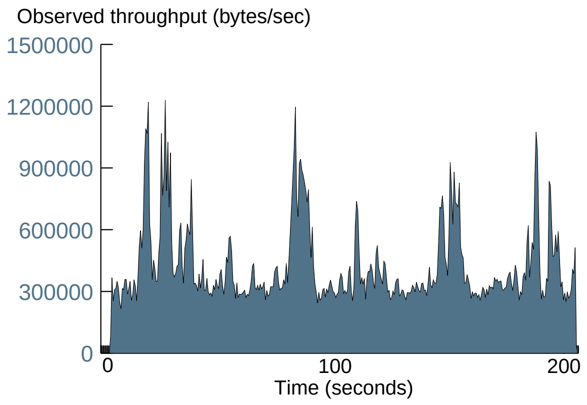
5.1 Scenario 1: Testing the Resource Adaptation Tier

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 service configurator. 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:

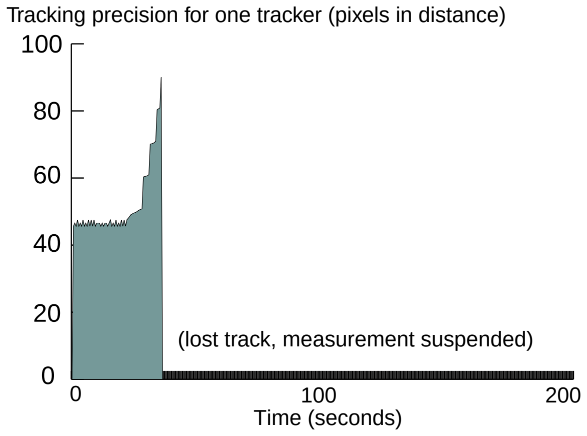
```
if (rate is moderate) then rateaction:= size;
if (rate is low) then rateaction:= size;
if (rate is high) then rateaction:= size;
```

This rule base introduces the effects of “bypassing” the component configurator in our tests, so that only the resource adaptation tier, including the resource 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.

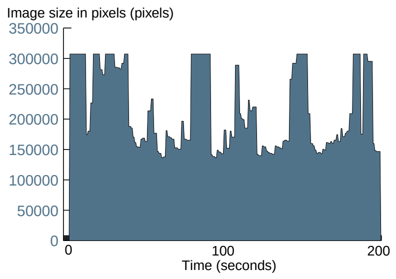
The results we have obtained are shown in Figure 17, divided in four parts. Figure 17(a) shows the observed variations in network throughput for the video being streamed from the tracking server to the client. Figure 17(b) shows the tracking precision for one tracker when the *Omnitrack* application is not supported by *Agilos*. After *Agilos* support is activated, Figure 17(c) shows the corresponding reactions generated by the



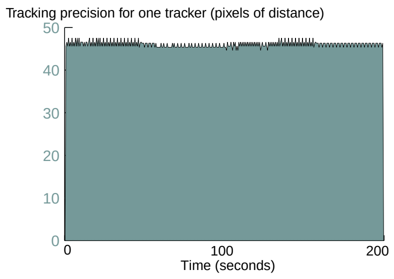
(a) Observed Throughput



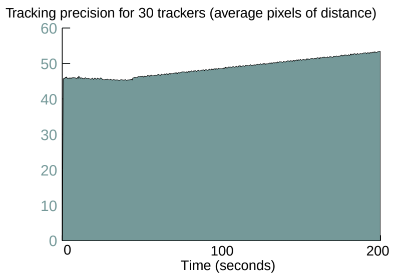
(b) Tracking Precision for One Tracker (without *Agilos*)



(c) Image Size in Pixels After Adaptation



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



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

Figure 17: Experimental Results for Scenario 1

component configurator of *Agilos* with the simplified rule base introduced previously. Such results reflect the capabilities of the bandwidth adaptor and observer in the resource adaptation tier. Figure 17(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 17(b), and is kept stable for the entire duration of measurements. Figure 17(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 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 resource adaptation tier of the hierarchical architecture is effective in maintaining the stability of tracking precision for one tracker, even with the simplified rule base that we have used in the component configurator. This demonstrates the validity of our approach with respect to this tier. 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.

5.2 Scenario 2: Testing the Component Configuration Tier under Fluctuating Bandwidth and CPU Availability

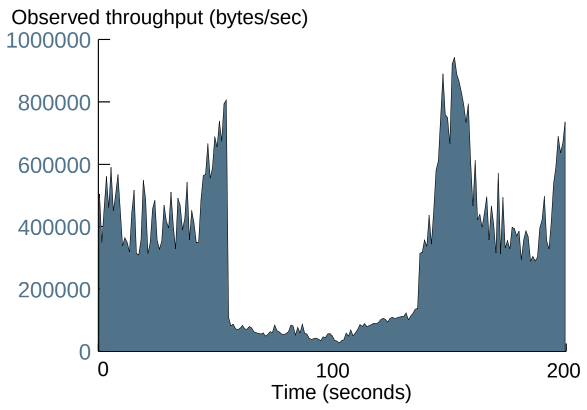
In this scenario, we deploy the component configurator using a full-fledged rule base, and emphasize testing the effectiveness of the Fuzzy Control Model. However, rather than testing under one fluctuating resource type, we examine the adaptation results when both CPU and network bandwidth are fluctuating. In this case, both CPU and network adaptors are actively in effect to support the decision-making process in the component configurator. The rule base is defined with the assistance of a *qualprobe*, by probing the relationships between tracking precision and other QoS parameters such as the tracking frequency and frame rate.

Figure 18 shows the results we have obtained. Table 1 shows the control actions generated by the component configurator at their respective starting times. The timing of these control actions are also visually embedded in Figure 18(b).

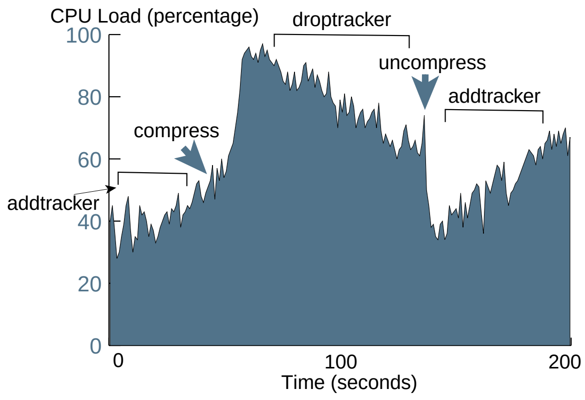
Start Time (sec)	*Control Action from Configurator*
4-40	`addtracker`
56	`compress`
62-120	`droptracker`
139	`uncompress`
145-178	`addtracker`

Table 1: Control Actions generated by the Component Configurator

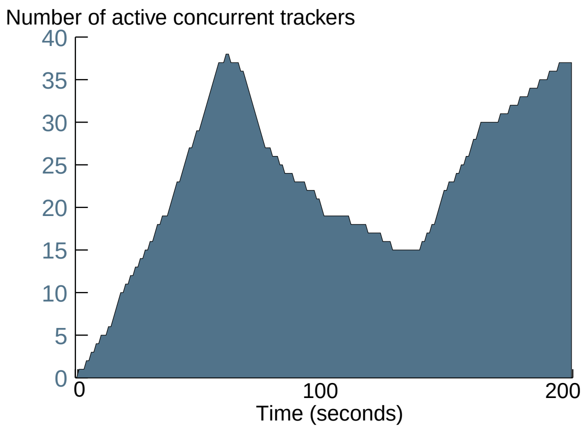
Our analysis on these experimental results is as follows. In Figures 18(a) and 18(b), we show observations with respect to network throughput and end-system CPU load. These observations drive the resource adaptors, whose output drives the behavior of component configurators. In the case of this scenario, a hybrid combination of parameter-tuning and reconfiguration 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



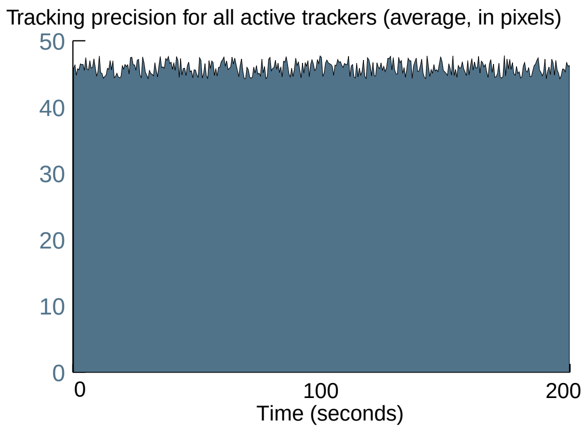
(a) Network Throughput



(b) CPU Load



(c) Number of Active Concurrent Trackers



(d) Average Tracking Precision for All Trackers

Figure 18: Experimental Results for Scenario 2

trackers. Since the trackers are executed in a round-robin fashion, 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 18(c) shows the number of active trackers, which is the result of the control actions `droptracker` and `addtracker` generated by the component configurator. The reconfigurations `compress` and `uncompress` 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 18(d), this combination of parameter-tuning and reconfiguration choices 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 component configurator, is crucial to maintaining the stability of the critical QoS parameter. This is manifested by a comparison between Figure 18(d) in this scenario and Figure 17(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.

5.3 Scenario 3: Testing the Service Configuration Tier

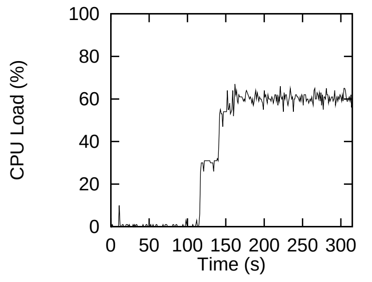
In this scenario, we proceed to evaluate the service configuration tier. We have designed three experiments to demonstrate its capabilities, focusing on providing each Omnitrack client with a QoS-satisfactory Omnitrack server. Primarily, we evaluate the *central processing* module by judging how well the service configurator 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. The servers were placed on three hosts, in descending order of 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 service configurator was placed on "PII 400".

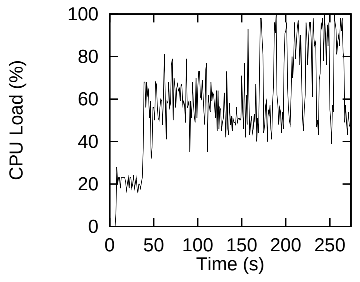
For the first experiment, we have manually selected the server for each client to achieve the best balance of the overall server load. This experiment was performed as a control for evaluating the second and third experiments. The results of this experiment are shown in Figure 19 (a-c).

For the second experiment, we repeated the first experiment, and allow the service configurator to select the server for each client. The results of this experiment are shown in Figure 19 (d-f). As expected, the service configurator has satisfied all client requests while performing an equivalent level of load balancing which had been performed manually in the first experiment. However, in this experiment the relative load on each server was close enough such that a random or round robin placement of the clients would have also performed an adequate job of load balancing. To prove that the service configurator would respond equally well within a biased service configuration, we switched the fastest server, "PII 400", from serving Motion JPEG to uncompressed video. This change greatly reduced the processor overhead for this server. In the third experiment, we allowed the service configurator to select servers with the new configuration.

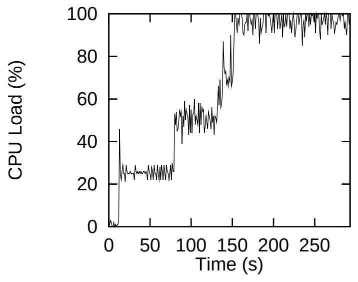
This time, the service configurator placed every client on “PII 400”. Even with all six clients being serviced, the CPU load on this host was still lower than the load on the other servers, each of which executing one server with no connections. Thus, we have successfully demonstrated that the service configurator is able to generate an optimal configuration of service provisioning relationships.



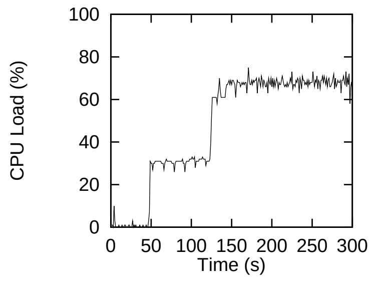
(a) PII 400



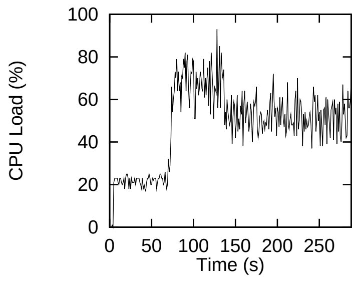
(b) PPro 200



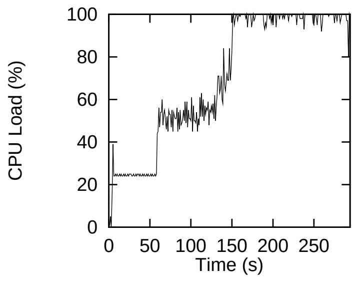
(c) P 200



(d) PII 400



(e) PPro 200



(f) P 200

Figure 19: Experiments with the Service Configurator

To summarize, we conclude that with the assistance of the fuzzy inference engine and a suitable rule base, the service configurator is able to correctly select a suitable server to optimize the service provisioning relationship.

6 Related Work

Traditional approaches of providing QoS guarantees in end systems and communication networks have been to rely on deterministic resource reservation and scheduling algorithms, which assume complete *a priori* knowledge of application requirements. These include bandwidth reservation and admission control protocols (such as RSVP for IETF int-serv networks [27]), packet scheduling algorithms in wireline and wireless networks [26, 17], and CPU reservation mechanisms [19, 8]. At the networking front, recent work towards reducing complexity at core network routers has also led to the introduction of Differentiated Services (diff-serv) [20]. Regardless of differences among various approaches, the ultimate objective of all related approaches is to provide a certain statistical or deterministic guarantees to the applications, by providing system-level solutions. Deploying these solutions may require modifications to the existing OS and networking protocols, which is problematic in heterogeneous environments of a large scale. Still, these approaches are necessary for hard real-time applications.

In comparison, our work focuses on the effectiveness of *adaptive QoS* controlled by a centralized architecture. Hence, there is no need to replace the existing general-purpose systems and best-effort networks, while the application may still keep the QoS of their critical parameters. This approach is most suitable for multimedia applications with soft real-time requirements. Typically, the approaches assume that applications can tolerate multiple levels of QoS, which vary in resource requirements. Given the requirements of different QoS levels, an adaptation mechanism can determine the right QoS level depending on load conditions. Several related previous projects also take this approach. Abdelzaher *et al.* [2] has proposed algorithms to select an appropriate QoS level according to *QoS contracts*, based on feedback control mechanisms. Brandt *et al.* [6] 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.

Resource management mechanisms at the systems level were developed to take advantage of adaptation. 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. We give several examples. In the work represented by [18, 5], 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. Noble *et al.* in [21] investigated 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. However, our approach is different in the sense that our design incorporate both parameter-tuning and reconfiguring choices in the application, and attempt to capture specific application characteristics at run-time. The Q-RAM architecture [22] 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 [12], the global resource management system was proposed, which relies on middleware services as agents to assist resource management and negotiations. In [24], 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.

In addition to the above resource management mechanisms, many active research efforts are also dedicated to various adaptive functionalities provided by *middleware services*. For example, Schmidt *et al.* [25] proposes real-time extensions to CORBA which enables end-to-end QoS specification and enforcement. The *QuO* project [28] 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. Kon *et al.* in [15] proposes a reflective middleware architecture *DynamicTAO* in order to transparently reconfigure itself to adapt to environmental changes. Finally, the GARA architecture [10, 9] integrates both CPU and network QoS provisioning by providing resource reservations to a required lower bound, and graceful adaptation beyond such guaranteed level.

Finally, recent work has focused on QoS-aware monitoring and probing mechanisms at the application level [11, 4, 3, 14, 7, 23, 1]. For example, Al-Shaer *et al.* [3, 4] has presented HiFi, an active monitoring architecture for monitoring distributed multimedia systems. In this model, changes in internal application parameters are modeled as *events*, application requirements modeled as filters, the monitoring process is thus modeled using the event-subscription model. Abdelzaher [1] has presented an on-line least squares estimator for estimating system parameters in QoS-aware web servers with a linear execution model. Similar to our work, *keleher-icdcs et al.* has favored the alternative of a centralized resource manager, and proposed an API for applications to specify tuning hooks to this manager. Chang *et al.* [7] has also proposed to detach adaptation decisions from applications, and has provided a "sandbox" implementation to tune resource availability in the process of off-line measurement of application behavior. In comparison, our work has similar goals with respect to detaching the decision-making process from applications, and probing the internal application behavior by an off-line algorithm. Furthermore, we defines the notion of *application QoS* using the quality of *critical parameters* that may not be observable on-line (e.g. tracking precision), and may depend on multiple tunable parameters and resources. Unlike the work in [1], we do not assume prior knowledge of a particular execution model in an application; our research focus is on learning the relationship between critical parameters and their dependents by an off-line probing algorithm, rather than the on-line monitoring process itself as in [4, 1, 23]. Assuming a lack of observability on-line with respect to critical parameters, our original contribution is to propose an polynomial off-line QoS probing algorithm to profile the application behavior off-line using benchmarking runs.

7 Conclusions and Future Work

The objective of this work is to actively control the application behavior so that it adapts optimally to external variations caused by an open, unpredictable environment. We have first defined application QoS as the quality of a set of critical parameters, and based on such assumptions, presented an efficient QoS probing algorithm for off-line probing of functional relationships between critical and tunable parameters. We have integrated the probing algorithm with our previous work, and designed a *hierarchical QoS control architecture* as a middleware solution to meet our goals. With the assistance of our probing algorithm, the architecture is able to capture the run-time behavior of the applications, and focus on the quality of critical parameters. In our extensive experiments with a complex distributed multimedia application, *Omnitrack*, we have verified that the critical parameter, tracking precision, may be maintained by trading off non-critical parameters, even with the presence of an open unpredictable environment.

There remains to be seen how other types of applications may benefit from our architecture, with respect to the effectiveness of the adaptation. We believe that other types of multimedia applications, including video conferencing and streaming, will benefit from our framework, since they are practically subsets of the Omnitrack application that we are currently using. We are in the process of applying our architecture to different categories of application beyond multimedia, particularly web server serving dynamic contents.

8 Acknowledgments

This work was partially 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. The authors would also like to thank Won Jeon for his contributions in the *Agilos* implementation, and the anonymous reviewers for their constructive comments in the review process.

References

[1] T. Abdelzaher. An Automated Profiling Subsystem for QoS-Aware Services. In *Proceedings of Second IEEE Real-Time Technology and Applications Symposium*, 2000.

[2] T. Abdelzaher and K. G. Shin. End-Host Architecture for QoS-Adaptive Communication. In *Proceedings of Fourth IEEE Real-Time Technology and Applications Symposium*, pages 121–130, June 1998.

[3] E. Al-Shaer. Active Management Framework for Distributed Multimedia Systems. *Journal of Network and Systems Management*, March 2000.

[4] E. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A New Monitoring Architecture for Distributed Systems Management. In *Proceedings of IEEE 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 171–178, May 1999.

[5] V. Bharghavan, K.-W. Lee, S. Lu, S. Ha, J. Li, and D. Dwyer. The TIMELY Adaptive Resource Management Architecture. *IEEE Personal Communications Magazine*, August 1998.

[6] 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*, pages 307–317, December 1998.

[7] F. Chang and V. Karamcheti. Automatic Configuration and Run-time Adaptation of Distributed Applications. In *Proceedings of Ninth Symposium on High Performance Distributed Computing (HPDC-9)*, pages 11–20, August 2000.

[8] Z. Deng, J.-S. Liu, and J. Sun. Scheduling of Real-Time Applications in an Open Environment. In *Proceedings of 18th IEEE Real-time Systems Symposium*, pages 308–319, 1997.

[9] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *Proceedings of 8th International Workshop on Quality of Service*, May 2000.

[10] I. Foster, A. Roy, V. Sander, and L. Winkler. End-to-End Quality of Service for High-End Applications. *IEEE Journal on Selected Areas in Communications, Special Issue on QoS in the Internet*, 1999.

[11] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed Performance Monitoring: Methods, Tools, and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):585–598, June 1994.

[12] 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.

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

[14] P. Keleher, J. Hollingsworth, and D. Perkovic. Exposing Application Alternatives. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems (ICDCS 99)*, pages 384–392, May 1999.

[15] F. Kon, R. Campbell, and K. Nahrstedt. Using Dynamic Configuration to Manage a Scalable Multimedia Distributed System. *Computer Communication Journal, Special Issue on QoS-Sensitive Distributed Network Systems and Applications*, 2000.

[16] 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*, 17(9):1632–1650, September 1999.

[17] S. Lu, V. Bharghavan, and R. Srikant. Fair Scheduling in Wireless Packet Networks. In *Proceedings of ACM SIGCOMM ’97*, pages 63–74, 1997.

[18] 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.

[19] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High Speed Networking*, 8(3-4):227–255, 1998.

[20] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. *Internet RFC 2638*, July 1999.

[21] 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*, October 1997.

[22] 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.

[23] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive Control of Distributed Applications. *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.

[24] 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.

[25] D. Schmidt, D. Levine, and S. Mungee. The Design and Performance of Real-Time Object Requests. *Computer Communications Journal*, 1997.

[26] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9:101–124, May 1991.

[27] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network Magazine*, pages 8–18, September 1993.

[28] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3, 1997.