Adaptive Middleware Architecture for a Distributed Omni-Directional Visual Tracking System

Baochun Li, Won Jeon, William Kalter, Klara Nahrstedt, Jun-Hyuk Seo*
Department of Computer Science
University of Illinois at Urbana-Champaign
b-li,wonjeon,kalter,klara,jseo@cs.uiuc.edu

Abstract

In different areas of applications such as education, entertainment, medical surgery, or space shuttle launching, distributed visual tracking systems are of increasing importance. In this paper we describe the design, implementation and evaluation of a distributed omni-directional visual tracking system, developed at the University of Illinois at Urbana-Champaign, with the Adaptive Middleware Architecture as the core of the system. With respect to both operating systems and network connections, adaptation is of fundamental importance to the tracking system, since it runs in an environment with large performance variations and without support of Quality of Service (QoS) guarantees.

We present (1) The design of an Adaptive Middleware Architecture that systematically supports application-aware quality adaptation; (2) A complete integration of user preferences and offline profiles in order to assist the middleware architecture and dynamically steer the adaptation path and behavior; (3) The design and experiments with a distributed omni-directional visual tracking system, showing the viability of our approach. In such an application, we show that with the support of the Adaptive Middleware Architecture, tracking precision can be kept stable under dynamic variations of the environment such as the fluctuation of available network bandwidth, variations in CPU load, and dynamic changes in the location and speed of the tracked objects.

Keywords: *Visual tracking, resource adaptation, adaptive middleware, adaptive application*

1. Introduction

Distributed visual tracking systems are becoming of increasing interest to several application areas such as education, entertainment or space shuttle launching. These tracking systems are distributed multimedia applications with a complex behavior, and are expected to deliver a desired level of Quality of Service (QoS) with key parameters such as the *tracking precision*[1]. In reality, however, these tracking systems run on general-purpose operating systems such as Windows NT, and over shared networks such as the Internet, both of which lack mechanisms for guaranteeing the strict timeliness and quality requirements. Dynamic variations in resource availability are caused by either dynamic demands of other concurrently running applications, or physical resource limitations in a heterogeneous network environment.

The presence of these dynamic variations in resource demands and availability calls for the adoption of on-the-fly adaptation mechanisms. These complex tracking applications are thus desired to adapt themselves and adjust their resource demands dynamically. Frequently, the purpose of adaptation is to reach one or more critical performance criteria, such as the *tracking precision*, by optimally adjusting functionalities and parameters internal to the application. These applications are said to be *flexible*, with a tolerable range of QoS parameters to allow room for adaptations to occur.

These flexible applications rely on the guidance of the *Adaptive Middleware Architecture* to meet their critical performance criteria. In the Adaptive Middleware Architecture, we address the problem of optimizing the *strategy* of adaptation. Some applications, such as video streaming, have a certain degree of adaptive strategies built in. However, these adaptive strategies are *ad hoc* and fail to consider the following concerns: (1) From the system's point of view, does the adaptation strategy conflict or being unfair to other concurrent applications in the same end system? (2) From the application's point of view, does the strategy focus on optimizing one or more critical perfor-

*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.

[1]The *tracking precision* is defined as the distance between the center of tracked region and center of the object. We wish to keep this distance stable and as small as possible.

mance criteria? Strictly speaking, an adaptation strategy guides the tradeoff among different application quality and system performance parameters. The optimization towards critical performance criteria may not be necessary in adaptive mechanisms that perform simple QoS tradeoffs, such as adaptive video playback. However, within a complex application where critical performance criteria depend on tuning multiple parameters or switching configurations, an optimized adaptation strategy becomes crucial to the success of application-aware adaptation.

Our approach and contribution in this paper are the following: (1) Design of an Adaptive Middleware Architecture with a fair, stable and configurable *adaptation strategy* related to resource consumption. In the middleware architecture, we introduce *Adaptors* that observe both system-level and application-level states, and propose control algorithms based on control theory to guarantee adaptation *stability* and *fairness* properties. (2) A strong focus on a specific critical performance criteria, such as the *tracking precision* in the tracking application. We introduce *Tuners* and *Configurators* that attempt to create an optimal adaptation strategy so that the criteria are met by trading off other less critical quality and performance parameters. For example, we can tradeoff perceptual image quality to preserve the *tracking precision*. (3) In order to gain precise knowledge about the adaptive behavior in the application and the relationship between critical criteria and resource consumptions, we introduce the Probing and Profiling Service in the middleware architecture, so that the adaptation strategy can be generated to be highly application-specific. (4) We present a distributed Omni-Directional visual tracking application as a case study, with multiple tracking servers and the capability of on-the-fly server switching. In order to assist such switching, we introduce *Negotiators* in the middleware architecture so that coordination among different end systems are more effective. (5) We show the complexity involved in the application in order to preserve the tracking precision, and the effectiveness of our middleware architecture via a series of experimental results in various scenarios.

The rest of this paper is organized as follows. In Section 2, we give an overview of the omni-directional visual tracking system and the Adaptive Middleware Architecture. In Section 3 we present the design issues of the Adaptive Middleware Architecture components such as the Adaptor, Configurator, Negotiator and the Profiling Service. In Section 4 we describe related implementation issues. In Section 5 we present experimental results with the visual tracking application. Sections 6 and 7 review previous related work and conclude the paper.

2. Overview

We give a brief introduction of the behavior and complexities involved in the distributed omni-directional visual tracking application, followed by an overview of the Adaptive Middleware Architecture.

2.1. Distributed Omni-Directional Visual Tracking

Implemented based on the *XVision* [3] project and in Windows NT, the *Distributed Omni-Directional Visual Tracking* system 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 Adaptive Middleware Architecture.

As illustrated in Figure 1, the visual tracking system is client-server based. The viewing area is surrounded by a set of pan/tilt cameras, referred to as *omni-directional cameras*, each served by a tracking server. The client is responsible to perform the following. (1) Accept user input (visual specification of the desired tracked objects) from the *User View Controller*; (2) Receive a video stream from one of the tracking servers, the *active server*; (3) Execute multiple computationally intensive tracking algorithms, referred to as *trackers*, on the tracked moving object; (4) Display the video with overlayed illustration of tracking results.

The selection of the active server that currently transmits video streams to the client is controlled by the *Gateway*. If the moving object gets out of the view, the system can either pan/tilt the server's camera to reach a better view of the object, or switch from one server to another in order to improve the view of the tracked object. In the cases of camera movement or server switching, the user may need to re-adjust the visual specification of desired tracked objects on the client.

The critical performance criteria of the tracking application is the *tracking precision*. However, due to the complexity involved in the application, it is not trivial to determine adaptation strategies that optimize the performance related to this critical criteria. These inherent complexities include: (1) The computational complexity and resource demands of multiple concurrent trackers are time-variant and specific to the object movement speed and the types of these trackers; (2) The tracking precision may be affected by parameters in multiple dimensions, such as video quality, CPU availability, as well as the shape and speed of objects, which is

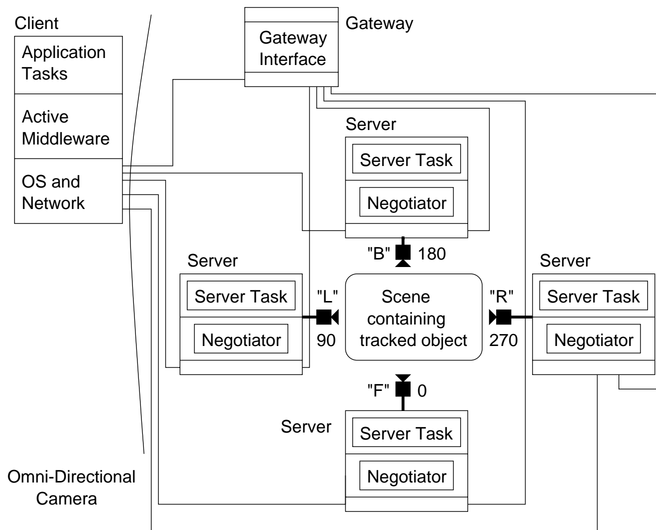


Figure 1. The Distributed Omni-Directional Visual Tracking System

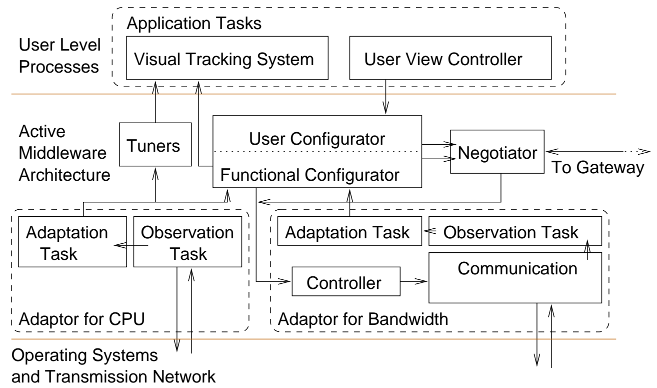


Figure 2. An Overview of the Adaptive Middleware Architecture

affected by switching servers that have diversely different camera angles. These observations make it non-trivial to determine the effects on tracking precision when actual resource availability varies.

In addition, the application includes a *User View Controller* that facilitates the integration of user preferences in the decision-making process of adaptation strategy. With the User View Controller, the user may (1) observe the streaming video, (2) visually determine which moving object in the video will be tracked, (3) determine the type of trackers being used, (4) move the pan/tilt tracking camera, and (5) switch to a different camera server for a better view.

Finally, we emphasize that the tracking application is multi-threaded on the client. The two primary threads are the *tracking* thread, and the *video streaming* thread. The tracking thread is responsible for the execution of the trackers, as well as the display of video and tracking results. The video streaming thread is responsible for video transfer from the server. We note that the application-specific QoS parameters are different in each thread. Within the CPU-intensive tracking thread, multiple trackers are executed iteratively and video frames are displayed in a Windows message loop, thus the key parameter is the *tracking frequency*, defined as the number of times that the tracker iteration loop can be executed per second. In contrast, within the throughput-intensive video streaming thread, the key parameter is the *frame rate*, which is defined as the rate at which frames are streamed to the client.

2.2. Adaptive Middleware Architecture

The ultimate objective of the Adaptive Middleware Architecture is to control application-aware adaptation behavior and to optimize the *adaptation strategy* towards application-specific performance criteria. In order to accomplish these goals, the middleware architecture consists of *Adaptors*, *Tuners*, *Configurators* and *Negotiators*. These components cooperatively monitor the application and system states, control applications to carry out adaptation decisions, and eventually meet the pre-specified performance criteria, such as the tracking precision. Figure 2 illustrates such an architecture.

The major responsibilities of the Adaptive Middleware Architecture are the following.

1. The Adaptive Middleware Architecture interacts with the underlying operating system, which is Windows NT in our implementation, and accurately observes the current state of the system and application, mainly with respect to resource availability. It is preferred to integrate this feature in the middleware architecture rather than applications themselves. This feature is implemented in a component referred to as the *Observation Task*.

2. The middleware architecture needs to decide the adaptation choices and actions to be carried out in the application, so that the adaptive behavior is both stable and fair to other concurrent applications in the same end system, and highly configurable in terms of *adaptation agility*. The agility represents the *sensitivity* or *responsiveness* of the application when adapting itself to external disturbances. These responsibilities of the Adaptive Middleware Architecture are integrated in a component referred to as the *Adaptation Task*. Since it depends on the accurate observations produced by the

Observation Task, we refer to the combination of both components as the *Adaptor*.

3. In order to balance between globally optimized and fair control decisions and the requirements of meeting diversely different critical performance criteria in different applications, we introduce the *Tuners* and *Configurators*. These components translate the output of control algorithms in the Adaptors into the actual parameter-tuning actions or reconfiguration choices to be carried out during the execution of applications. The *Tuners* are responsible for parameter-tuning actions, whereas the *Configurators* are in charge of *functional* reconfiguration choices, and are activated at a much lower frequency.

4. In extreme cases, in order to deal with prolonged period of limited resources and degraded qualities, *Negotiators* are activated to coordinate with other end systems. In the case of the tracking application, the Negotiators are responsible to locate the new active camera server via the Gateway. After a successful negotiation, the Negotiators are also responsible for a smooth video transition from the new server, soliciting user preferences when necessary.

The Adaptive Middleware Architecture is designed to be generic, rather than specific to the tracking application. In the end system, for example, there may exist one Adaptor per resource type, but more than one Configurators, each controlling a concurrent application. In other words, the Adaptors are associated with resource types, while the Configurators are application-specific. Furthermore, the Tuners are associated with both resource types and applications, i.e., each Tuner corresponds to a single resource Adaptor, as well as a single application. From an implementation standpoint, for the Adaptive Middleware Architecture to be generic and able to control a wide variety of flexible applications, the interaction among different middleware components and the applications is through a specific service enabling platform, with the current implementation being CORBA 2.0 implemented in Windows NT.

3. Components in the Adaptive Middleware Architecture

3.1. Adaptors

In order to control the adaptive behavior of flexible applications with a global awareness of resource availability, so that fairness, stability and adaptation agility properties can be mathematically reasoned about and proved, we integrated a *Task Control Model* as proposed in our previous work [5] into the design of our middleware Adaptors. This is complementary to the design of Tuners and Configurators presented in Section 3.2. The Adaptors promote global resource awareness, while Tuners and Configurators focus on application-specific nonlinear adaptation possibilities.

In the *Task Control Model*, the application task to be controlled by the middleware Adaptor is referred to as the *Target Task*. Within the middleware Adaptor, the *Adaptation Task* carries out the control algorithm, and the *Observation Task* observes the current system status. For the purpose of developing and reasoning about properties in the adaptation behavior, we adopted a control-theoretical approach to model the Target Task and the control algorithm carried out in the Adaptation Task.

In the Target Task, we assume multiple tasks competing for a shared resource pool. Each task T_i makes *new requests* u_i for resources in order to perform their actions on inputs and produce outputs. These requests may be *granted* or *outstanding*. If a request is granted, resources are allocated immediately. Otherwise, the request waits with an outstanding status until it is granted. The system grants requests from multiple tasks with a constant *request granting rate* y. Figure 3 illustrates the Task Control Model. In our previous work [5], we were able to derive control algorithms in the Adaptation Task in this scenario, and prove stability and fairness properties based on the derived control algorithm.

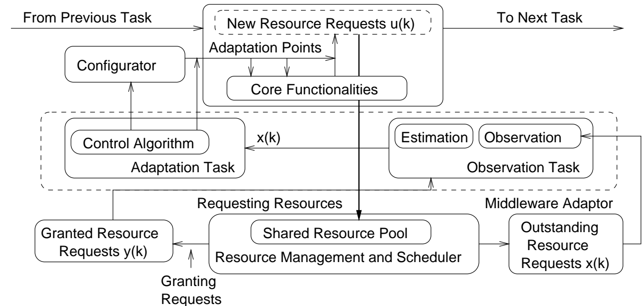


Figure 3. The Task Control Model

3.2. Tuners and Configurators

The *Tuner* and *Configurator* determine discrete control actions based on application-specific needs and control values produced by the Adaptors. They serve as an extension to the Adaptation Task in the Task Control Model, augmenting its capability so that it is highly application-specific. Similar to the role of the Task Control Model in the design of the middleware Adaptors, we leverage the rich semantics and features in existing fuzzy control systems to design a Fuzzy Control Model [6] for Tuners and Configurators. An illustration for this design, with the Configurator as an example, is given in Figure 4.

3.2.1 The Fuzzy Control Model

The model consists of five components. The *fuzzy inference engine* implements particular fuzzy control algorithms defined in the application-specific *rule base* and *membership functions* for linguistic variables. The *input normalizer*, *fuzzifier* and *defuzzifier* prepare input values for the fuzzy inference engine, and convert fuzzy sets (the decisions made by the inference engine) to the actual real-world control actions for the applications.

The advantages of adopting this design model are as follows. (1) Taken the fact that multiple reconfiguration options and parameter-tuning possibilities exist in a typical complex application, the controllable regions and variables within the application are in most cases discrete, non-linear and complex by nature. On the other hand, a fuzzy control system is naturally a nonlinear control system, in which the relationships between inputs and control outputs are expressed by using a small number of *linguistic rules* stored in a *rule base*. The nonlinearity of the fuzzy controller matches naturally with the nonlinearity of controllable regions and adaptation possibilities within an application. (2) The model, using membership functions for linguistic variables and the rule base, is inherently generic and highly configurable according to specific application needs.

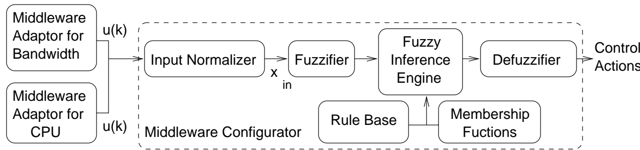


Figure 4. The Overall Architecture of the Fuzzy Control Model

3.2.2 The Rule Base

The decisions of selecting linguistic values and rules in the rule base are based on a combination of human expertise and trial-and-error experiments on the particular application. The tradeoff is to decide on a minimum number of linguistic rules, while still maintaining the desired accuracy to achieve an acceptable adaptation performance. All of the linguistic values in the rule base should use words of a natural or synthetic language, such as `moderate` or `below_average` for the linguistic variable `cpu`. These values are modeled by fuzzy sets. The design of the *Rule Base* involves the generation of a set of conditional statements in the form of if-then rules. Examples of these rules are:

/* Linguistic rules corresponding to bandwidth adaptation */

if rate is very_high **then** rate_demand is chopped_image
if cpu is very_high **and** rate is below_average **then** rate_demand is compress

/* Linguistic rules corresponding to bandwidth adaptation */

if cpu is very_high **and** rate is above_average **then** cpu_demand is add_tracker
if cpu is below_average **and** rate is very_low **then** cpu_demand is drop_tracker

where *rate* and *rate_demand* are linguistic variables, *very_high* and *below_average* are linguistic values, characterized by their membership functions. Each rule defines a fuzzy implication that performs a mapping from fuzzy input state space to a fuzzy output value. After the defuzzification process, the fuzzy output value directly corresponds to a particular control action within the application.

3.2.3 The Difference between Tuners and Configurators

Both Tuners and Configurators implement the Fuzzy Control Model, however there are reasons to distinguish between the two. Tuners are designed to control the application by parameter-tuning actions (data adaptation actions, for example the modification of frame size or frame rate), and are activated frequently, normally for small-scale adjustments in the application. In contrast, the Configurators are designed to activate functional reconfiguration actions, when the necessity arises to adapt to fundamental changes in the environment, which cannot be remedied by parameter-tuning adjustments. Adaptation timing is different between Tuners and Configurators, the latter are activated much less frequently.

3.3. Smart Offline Probing and Profiling Service

As noted in the introduction, one of the most important objectives in the design of Adaptive Middleware Architecture is to carefully create an *adaptation strategy* that meets the critical performance criteria within the application, by trading off the quality of other less critical parameters. In the tracking application, the critical performance criteria is the *tracking precision*, since once the trackers lose track, the perceptual video quality does not contribute at all. In this section, we focus on a smart *offline probing/profiling* service that is complementary to the Tuner.

The need for the offline probing/profiling service is explained as follows. (1) Online measurements of critical application performance parameters are not possible. For example, in visual tracking the *tracking precision* cannot be readily measured by source-level instrumentation, since in live video the precise location of the moving object at any instant is unknown. (2) Relationships between critical parameters and tunable parameters by the Tuner need to be known, most conveniently as a continuous function, or at least as a lookup table. For example, the mapping between tracking precision and frame rate or tracking frequency is desired in order to create optimal adaptation strategies and

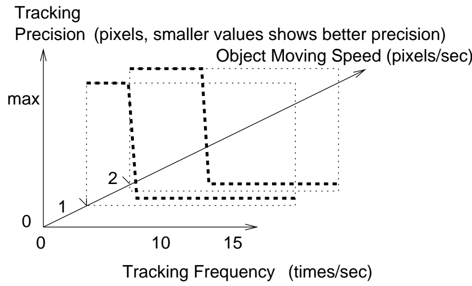


Figure 5. Offline Profiling for Visual Tracking

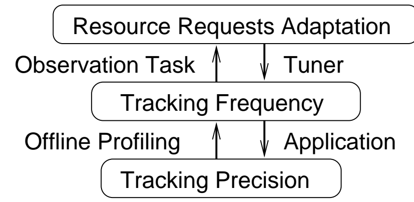


Figure 6. The Process towards Optimal Adaptation

meet the performance criteria. (3) The above described relationship may not be a straightforward single-variable linear function, more variables may be involved. For example, in visual tracking the *moving speed* at which the object is in motion is a key factor affecting the function between tracking precision and frequency. When the *moving speed* is slow, a stable precision only requires relatively low tracking frequency. However, when speed of the object is fast, precision requires much higher tracking frequency, thus complicates the specification of the functional relationship between precision and tracking frequency.

We present our approach, referred to as *smart offline probing/profiling*, based on our case in the visual tracking application.

Offline Probing/Profiling is defined as the trace measurement procedures of tracking precision *before* actual live video is processed. This technique is valid based on the following assumptions. (1) Though tracking precision cannot be measured in live video, it can be measured in artificial animations used as benchmarks of visual tracking, since the specific positions of artificially animated objects can be easily computed. (2) The relationships among three parameters, the *precision*, *moving speed* and *tracking frequency*, stay the same after switching the contents from artificial animations to live video, assuming the same hardware and OS platform.

In the actual implementation, we probe and create a three-dimensional offline profile for precision, moving speed and tracking frequency. An illustration of such a profile is shown in Figure 5.

In order to obtain this profile, we keep *moving speed* fixed and measure traces of tracking precision under different tracking frequencies, which can be affected by manipulating tunable parameters in the Tuner, such as frame size (to cope with variations in bandwidth) or number of concurrent trackers (to cope with variations in CPU capacity). We then increase the moving speed by a small margin and repeat the process. This procedure continues until the maximum moving speed.

In order to store the profile obtained by offline trace measurements, we note from Figure 5 that the tracking precision can be approximated by a binary variable, i.e., it can be either in state *lost track* or *stable*. This makes it possible to store only the *cutoff frequency* at individual moving speed levels, saving the amount of data to be stored. This does not apply to other applications that do not show the binary property with respect to their critical performance parameters.

Once the profiles are obtained, the visual tracking application can be readily switched to online mode, capturing live video and tracking objects on the fly. The adaptation strategy can be intuitively explained as follows: The Observation Task observes the tracking frequency and calculates the volume of *outstanding resource requests* to the CPU Adaptor. The Adaptor produces an *adapted resource request rate* that preserves the desired weighted fairness, stability and agility properties. These request rate values are inputs to the Tuner and Configurator, where they are mapped back to the application-specific parameters, in our case, the tracking frequency. The Tuner decides if an appropriate adaptation action should be activated. If the observed tracking frequency is higher than the cutoff frequency by a pre-defined margin, adaptation is not activated. Otherwise, adaptation is activated according to the fuzzy inference engine, which is based on the rule base in the Fuzzy Control Model. The offline profiles are critical to assist creating an optimal set of rules and membership functions being used in the Tuner and Configurator. Figure 6 illustrates the procedure.

3.4. Integration of User Preferences

In the Adaptive Middleware Architecture, we adopt a hybrid approach in the decision-making process of a particular adaptation strategy. While most of the adaptation choices and decisions are generated automatically within the middleware, we believe that user preferences and choices should be given a high priority and be integrated in the decision-making process.

The *User View Controller* is included to facilitate this integration. In the case of the tracking application, the User View Controller is provided for the user to be involved in the adaptation process. The adaptation possibilities that require

user intervention are the following. (1) The user may initiate, drop and replace various types of trackers on-the-fly; (2) The user may control the movement of a pan/tilt camera to obtain a better view; (3) When stationary cameras are used and movement is not possible, the user may switch the view of the camera by switching tracking servers; (4) After view movements or switches, the user may visually select the objects again so that trackers can regain the desired tracking precision.

3.5. Negotiators

The Adaptor and Configurator are usually capable of making the necessary adjustments to adapt to dynamic resource availability. However, there are instances when further adaptation capabilities are necessary. When there is a prolonged period of limited resources the quality of the application may be degraded beyond usability, despite the best effort adaptations within the end system. For the sake of more effective adaptations, we introduce the *Negotiators* in the middleware architecture so that smooth coordinations can be achieved among different end systems.

We focus on the distributed omni-directional tracking application as a case study. Various solutions exist when a prolonged period of limited resources are present. One case is that when this occurs, the best solution may be to switch from an uncompressed to a compressed video stream. This adaptation strategy is controlled by the Configurator, and is useful when CPU utilization is manageable but network bandwidth is low. Similarly, it may be beneficial to switch from compressed to uncompressed video when the CPU is over-utilized and there is network bandwidth available. Another case for which no amount of adaptation can compensate is when the object being tracked goes out of the server's camera range. In all of these cases, the client needs the ability to switch from one server to another. In the first case, the client can switch to a server transmitting data in the desired format. In the second case, the client can switch to a server which has the object being tracked in its view.

Several components of the client are used to facilitate this omni-directional functionality. The user can turn the camera or move to a new server with a better view of the moving object through the User View Controller. This GUI sends messages to the user Configurator within the adaptive middleware. The user Configurator then sends a request to the Negotiator to turn the camera or switch to a new server in the requested direction. The Configurator may also request that the Negotiator switches servers if the fuzzy logic engine decides a change from compressed to uncompressed video or vice versa is necessary. These control messages are referred to as requests because they are not guaranteed to produce the desired result. For example, if the Configurator requests compressed video but there is no compressed video server then the request will fail.

Once the client's Negotiator receives a move or turn request, it forwards the request to the Gateway. The Gateway is responsible for finding the best server that matches the requested parameters and facilitating a smooth transition from one server to the other on behalf of the client. To do this, the Gateway requires that each server provides two pieces of information to the Gateway: the format with which it is transmitting data and what view of the overall scene the server's camera has. These parameters are sent to the Gateway when the server first comes online. The second parameter - what view the server has - requires that an arbitrary coordinate system is imposed on the scene. The scene is given four directions - front, left, back and right, or FLBR. One direction is designated as front and any camera with a full frontal view of the scene is at position zero. The positions then correspond with increasing degrees clockwise from the zero position: L is 90, B is 180, and so on. Combinations of directions may also be used, such as FL for position 45. It is important to distinguish that the positions are relative to the scene containing the tracked object, not the object itself. Though the object may move throughout the scene, the coordinate system never changes. Figure 7 shows an example containing three servers. Server 1 is transmitting compressed motion-JPEG data and is located at F, position 0. Server 2 is transmitting uncompressed video and is located at L, position 90. Server 3 is also transmitting uncompressed video and is located at FR, position 315.

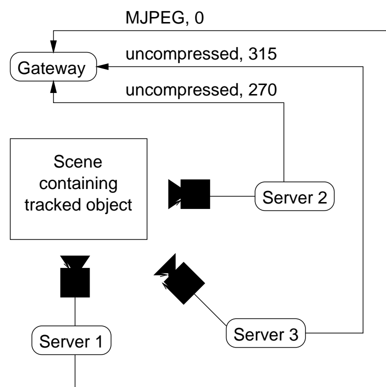


Figure 7. An Example of Three Servers

When the Gateway receives a turn request from a client, it forwards the message to the client's server. If the server is capable of pivoting the camera in the specified direction then it does. If not, it reports to the Gateway that it could not turn and the Gateway tries to move the client to a new server in the given direction. This essentially transforms failed turn request into a move request.

When the Gateway receives a move request from a client first tries to find the best server which matches the requested

parameter. If the message requests to move in a particular direction, the best server is the server closest to the client's current server in the given direction with the same video format. If the message requests to change compression, the best server is the server closest to the client's current server in either direction with the requested video format. The Gateway then authenticates the client with the best new server. If the server accepts, the Gateway informs the client to switch servers and unathenticates the client with the client's old server. If the server rejects, the Gateway tries again with the next best server. This continues until a server accepts or until all matching servers, if there were any, have rejected the client. If this happens the Gateway informs the client that the request could not be performed. Currently servers do not reject clients but future implementations could take advantage of this feature. For example, a server could reject a client when doing so would degrade the QoS of video with the server's current clients. Figure 8 shows a typical successful implementation of the server switching protocol.

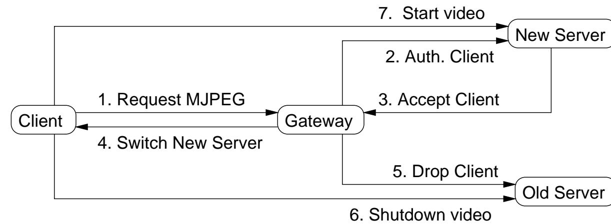


Figure 8. A Typical Successful Implementation of the Server Switching Protocol

Beyond performing as server switching mediator, the Gateway attempts to maintain a stable video stream on the client's behalf. For example, if a server becomes unstable or goes offline, the Gateway automatically attempts to switch the server for each client which had been connected to that server. The Gateway searches for servers closest to the old server with the same video format and moves the affected clients to those servers. Server instability is determined by the Gateway pinging each server at regular interval. Failed acknowledgments force the Gateway to assume that the server has become unstable or gone offline.

One of the most time consuming aspects of switching servers is opening a new TCP connection with the new server every time a switch occurs. To avoid this delay, the Gateway initially supplies the client with a list of all servers and updates the client as new servers come on and offline. The client opens a control connection with each server even though it may only use one server at a time. This way, if the client does switch servers, the connection with the new server is already established. The client is prevented from utilizing several servers at once because a server will only supply video to the client if it has been authenticated by the Gateway in the preceding protocol.

4. Implementation Issues

The distributed omni-directional visual tracking system is implemented in Windows NT, with its core being the Adaptive Middleware Architecture discussed in the last section. All interactions among various components within the middleware architecture and the application are via CORBA, with interfaces clearly defined in IDL. This ensures that the middleware architecture is generic and not tightly bound to the applications.

4.1. Adaptation Choices

Divided in two major categories, we have identified the adaptation choices in the tracking application as the following.

4.1.1 Adaptation of Communication Bandwidth Requirements

First, there exist several options for parameter-tuning actions during an uncompressed image transfer. (1) The *image size* can be enlarged or reduced to adjust bandwidth requirements, by *chopping* the edges. The tradeoff is that the smaller the image, the higher the probability that the objects move out of the range. (2) The *image size* can be enlarged or reduced by *scaling*. The tradeoff is a higher CPU load for real-time per-frame scaling. (3) The *color depth* can be altered. Existing choices for coding one pixel are 24 bits RGB, 16 bits packed RGB, 8 bits grayscale or 1 bit black-and-white.

Second, if we consider functional reconfiguration choices, *compression* and corresponding *decompression* can be activated, using available choices such as Motion-JPEG and streaming MPEG-2. In some case, for example when hardware is needed, this activation is made possible by switching between camera servers, i.e., to those servers that support the suitable codec formats. In other cases, bandwidth requirements are reduced dramatically at the expense of increased CPU load. In both cases, the adaptations require control feedback from the client's Bandwidth Adaptor to its counterpart in the server.

4.1.2 Adaptation of CPU Requirements

The tracking algorithms are inherently computationally intensive. In the current implementation, there are three frequently used tracking algorithms. *Line* tracking and *corner* tracking are edge based algorithms, *SSD* tracking is a region based algorithm. Experiments show that different tracking

algorithms present diverse computational requirements. In addition, the application can run multiple algorithms tracking multiple objects simultaneously, with each algorithm referred to as a *tracker*, and the tradeoff being increased computation load. These facts motivate the following reconfiguration choices: (1) Add additional trackers to utilize idle CPU; (2) Drop running trackers to decrease CPU demand; (3) Replace existing trackers by less or more computationally intensive trackers. Finally, parameter-tuning adaptation may also be applied by modifying the size of the *tracked region* of a specific tracker, effectively tuning the computational load of the tracker. The tracked region is defined as the searching range of the tracker in the feature detection stage of computation.

4.2. Tracking Thread

Within the tracking thread, each tracking algorithm is executed inside the Windows message loop, similar to the following simplified source skeleton:

```
while (GetMessage(&msg, NULL, 0, 0)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
  // Polls for new CORBA events
  reactor->dispatchOneEvent(timeout);
  // Receives one frame from streaming
  // client thread with shared buffer
  captureFrame(frameBuffer);
  // multiple trackers iteration loop
  while (i = 0; i < TrackerNum; i ++) {
    tracker[i]->track();
  }
}
```

The tracking thread is highly CPU intensive. There are two primary causes. First, the Windows `WM_PAINT` message is handled to draw video frames to a window, and the bitmap stretching process is CPU intensive. Second, calls to the `track()` method for each trackers are themselves CPU intensive as well. The adaptation choices related to CPU requirements are executed within this thread.

4.3. Video Streaming Thread

The *Video Streaming* thread in the tracking application consists of two channels:

1. Control channels. They are primarily used to coordinate between client and the server for their adaptation choices. The control channels are setup when a new connection is made between the server and the client by the request of the client. The channel between client and server carry control information such as video format (uncompressed or MJPEG compressed), video quality and properties, camera position, etc. For example, when the Adaptive Middleware Architecture decides to switch to a compressed video stream, control channels are used to notify the server so that the following frames can be sent as compressed. Alternatively, control channels may be used to notify the Gateway, so that the active tracking server can be switched to a server with the appropriate encoding capabilities (such as MPEG streaming hardware support).

2. Multimedia streaming channels. They are used for live video streaming. All channels and data transmission are implemented with WinSock 2 in Windows NT.

Implementation-wise, the video streaming thread coordinates with the tracking thread using a shared ring buffer. Proper synchronization is implemented so that both threads can access the buffer correctly.

5. Experimental Results

Both the Adaptive Middleware Architecture and the distributed omni-directional visual tracking application are implemented on Windows NT 4.0 on the x86 platform. The Gateway runs on a Pentium Pro with 128 MB RAM. One server runs on 300 MHz Pentium II with 64 MB RAM and another runs on a Pentium Pro with 128 MB RAM. The client runs on a 200 MHz Pentium MMX with 64 MB RAM. Both servers are equipped with with Sony EVI-D30 pan/tilt digital cameras. All PCs are connected via the 10 Mbps Ethernet.

5.1. Experimental Scenarios

We present experimental results in three different experimental scenarios.

(1) An animated video sequence is streamed from the server to the client using Motion-JPEG compression. The reason for using animated video sequence is that it allows us to measure directly the tracking precision and generate offline profiles via the probing/profiling service. The profiles include measurements of tracking precisions that are not possible to measure directly in live video. 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 omni-directional 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

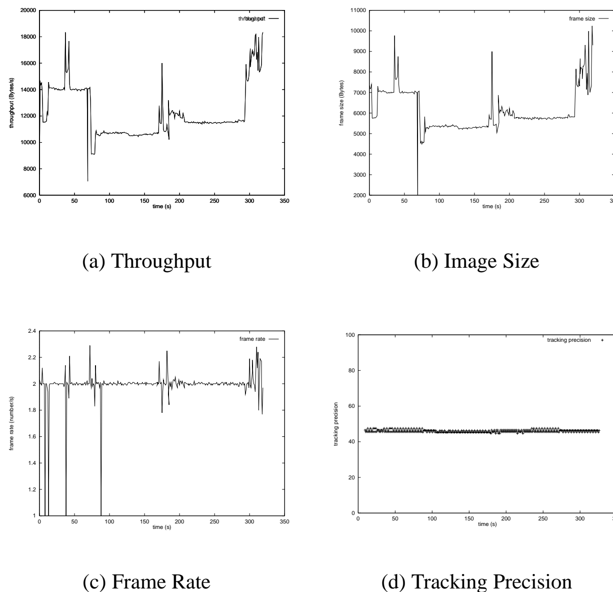
action simultaneously, such as compression and dropping trackers. We finally measure the tracking precision and show that the tracking precision remains stable with fluctuating CPU availability.

(3) With the assistance of the Gateway and Negotiators, we switch the active server from an uncompressed video server to a Motion-JPEG compressed video server, changing the camera view and the video formats by the switch. We measure the camera initialization time, repositioning time and the server switching delay.

5.2. Experimental Results

5.2.1 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 the Adaptive Middleware Architecture.



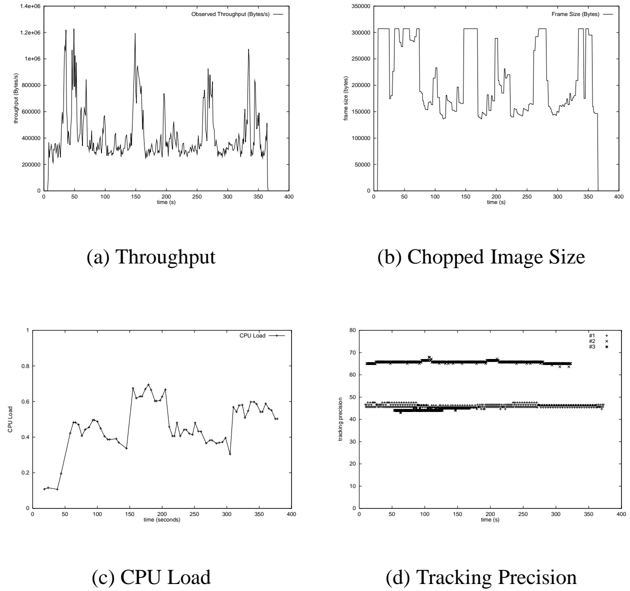
(a) Throughput (b) Image Size

(c) Frame Rate (d) Tracking Precision

Figure 9. Scenario 1

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



(a) Throughput (b) Chopped Image Size

(c) CPU Load (d) Tracking Precision

Figure 10. Scenario 2

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

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

5.2.3 Scenario 3

In this scenario, we measure the time necessary to start a minimal omni-directional camera based on Scenario 2. The Gateway is started first, immediately followed by the server. The camera initialization time is the period between starting

time of Gateway and finishing time of server registration at the Gateway. Figure 11 illustrates the *Camera Initialization Time*.

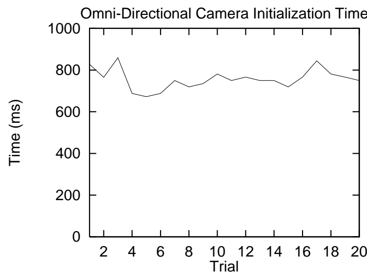


Figure 11. Camera Initialization Time

Once the omni-directional camera is established, the client connects to the tracking server and start receiving video. The *Client Initialization time*, shown in Figure 12, is the time it takes from starting time of the client till the display of video.

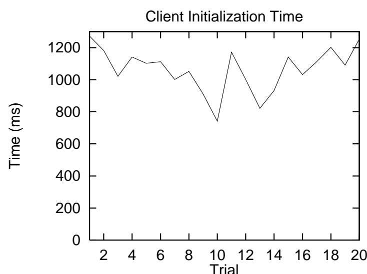


Figure 12. Client Initialization Time

Figure 13 shows the *Server Switching Time*. To measure the server switching time, two servers serve live video with different video formats, e.g., uncompressed and MJPEG compressed. The user requests to switch from one server to the other by clicking on the *move* button on the User View Controller. The switching time is the period between the time when a user clicks the button on the User View Controller and the time when the client retrieves the video stream from the new server.

Figure 13. Server Switching Time

6. 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 [7, 1], 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 [8] 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.

Another related group of previous work studies the problem of dynamic resource allocations, often at the operating systems level. Noteworthy work are presented in [9, 4, 10]. The work in [9] 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 [4], 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.

In contrast, our work distinguishes in domain, focus and solutions. For example, our work in the Task Control Model focuses on the analysis of the actual adaptation dynamics, which is more natural for modeling with a control-theoretic approach, rather than overall system utility factors. In addition, rather than focusing on a multi-machine environment running a single complex application, our work focus on an environment with multiple applications competing for a limited amount of shared resources, which we believe is a common scenario easily found in many actual systems. Thirdly, we focus on optimizing a critical performance criteria in the application, by trading off other less critical parameters by adaptation. Finally, our work focuses on proposing various schemes for the middleware compo-

nents to actively control the application, rather than providing resource allocation and management services in the execution environment to meet the application's needs. In other words, we focus on adapting applications, rather than resource allocations in the system.

Recently, in addition to studies in the networking and resource management levels, many active research efforts are also dedicated to various adaptive functionalities provided by middleware services. For example, [11] proposes real-time extensions to CORBA which enables end-to-end QoS specification and enforcement. [12] 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 [2] 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 work is orthogonal to the above approaches, since the Adaptive Middleware Architecture is based on underlying service enabling platforms, which is CORBA in our experimental testbed. In addition, we attempt to provide adaptation support to the applications proactively, rather than integrating adaptation mechanisms in CORBA services so that they are provided transparently to the applications. Furthermore, we attempt to develop mechanisms that are as generic as possible, applicable to applications with various demands and behavior. Finally, we attempt to provide support in the Adaptive Middleware Architecture with respect to multiple resources, notably CPU and network bandwidth.

7. Conclusion

This paper has presented several new contributions in the area of multimedia computing and networking. First, the design and implementation of an adaptive middleware architecture is outlined which assists a flexible application with data adaptation and functional adaptation in an integrated fashion. Second, we discuss services such as probing and profiling which provide the bridge between critical quality parameter(s), which application cares about, and underlying system resource parameters and other less critical application parameters. Third, we have implemented a unique distributed omni-directional tracking application which allows visual tracking, camera movement tracking and view tracking using camera switching capabilities. This flexible and complex application allows us examine new integrated adaptation strategies in the data domain and functional domain and their impact on application performance.

The overall work is unique because it integrates the feedback of an adaptive middleware framework onto the design of a multimedia application and the feedback of an application onto the design of an underlying system support under the common goal of satisfying the critical application performance requirements.

References

[1] 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.

[2] 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, Dec. 1998.

[3] G. Hager and K. Toyama. The XVision System: A General-Purpose Substrate for Portable Real-Time Vision Applications. *Computer Vision and Image Understanding*, 1997.

[4] 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.

[5] B. Li and K. Nahrstedt. A Control Theoretical Model for Quality of Service Adaptations. In *Proceedings of Sixth International Workshop on Quality of Service*, 1998.

[6] B. Li and K. Nahrstedt. Dynamic Reconfigurations for Complex Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1999.

[7] 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.

[8] 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.

[9] 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.

[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.

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

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