Application of the QuO Quality-of-Service Framework to a Distributed Video Application

David A. Karr, Craig Rodrigues,
Joseph P. Loyall and Richard E. Schantz
BBN Technologies
karr@acm.org, {crodrigu, jloyall, schantz}@bbn.com

Yamuna Krishnamurthy and Irfan Pyarali
Department of Computer Science
Washington University
{yamuna, irfan}@cs.wustl.edu

Douglas C. Schmidt
Electrical & Computer Engineering Department
University of California
schmidt@uci.edu

Abstract

Adaptation of distributed software to maintain the best possible application performance in the face of changes in available resources is an increasingly important and complex problem. We discuss the application of the QuO adaptive middleware framework and the CORBA A/V Streaming Service to the development of real-time embedded applications. We demonstrate a standards-based middleware platform for developing adaptive applications that are better architected and easier to modify and that can adapt to changes in resource availability to meet QoS requirements. These are presented in the context of a video distribution application. The application is developed using QuO and the A/V Streaming Service, and uses adaptive behavior to meet timeliness requirements in the face of restrictions in processing power and network bandwidth. We present experimental results we have gathered for this application.

1. Introduction

Middleware for Distributed Object Computing (DOC) is an emerging and increasingly accepted tool for development and implementation of a wide variety of software applications in a wide variety of environments. The application of DOC middleware to real-time, embedded software (RES) has resulted in the emergence of middleware support for the strict quality of service (QoS) requirements of RES use cases. For example, the Minimum CORBA specification [11], the Real-time CORBA 1.0 specification [11], and the Real-Time Specification for Java (RTSJ) [7] are examples of extensions and services that have grown out of a need to support embedded and real-time applications. Adaptation of distributed software to maintain the best possible application performance in the face of changes in available resources is an increasingly important and complex problem for RES applications. We have been developing QuO, a middleware framework supporting adaptive distributed-object applications.

In this paper, we apply QuO to the development of an Unmanned Aerial Vehicle (UAV) video distribution application, in which an MPEG video flow adapts to meet its mission QoS requirements, such as timeliness. We discuss three distinct behaviors that adapt to restrictions in processing power and network bandwidth: reduction of the video flow volume by dropping frames, relocation of a software component for load balancing, and bandwidth reservation to guarantee levels of network bandwidth. We have developed a prototype application which uses QuO, the TAO real-time ORB, and the TAO Audio/Video (A/V) Streaming Service to establish and control video transmission via a distribution process to viewers on computer displays. The TAO A/V Streaming Service is an implementation of the CORBA A/V Streams specification [10], which grew out of the need to transmit multimedia data among distributed objects.

The application demonstrates a standards-based middleware platform for RES applications and that adaptation can be effectively controlled by a superimposed QuO contract to regulate performance problems in the prototype that are induced by processor or network load. Our experience also shows that the use of the QuO framework, in contrast to typical ad-hoc performance-optimization techniques, which become entangled with basic functionality, leads to a form of aspect-oriented programming with a beneficial *separation of concerns*. This results in code that is clearer and eas-

ier to modify, and promotes re-usability of software under different requirements.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to QuO. Section 3 describes the UAV implementation. Section 4 describes the way in which the UAV prototype can adapt to resource constraints using QuO while still delivering its live video feed in a timely manner using the A/V Streaming Service. In Section 5, we present empirical results showing the improvement in performance under load provided by adaptation and the software engineering benefits of using the QuO framework to implement QoS concerns. Section 6 discusses related work. Section 7 projects future work on this research. Finally, Section 8 presents some concluding remarks.

2. An Adaptive Framework for DOC

2.1. The Benefits of Adaptation

Except in systems whose deployment environment is extremely stable, operational DOC systems will encounter more-or-less temporary conditions that impinge on their available computing resources (e.g., processing power and network bandwidth) and have a consequent effect on the ability of the system to deliver the QoS needed by users. For example, other applications may require resources, hardware may fail, or the network may be reconfigured.

It is also possible in many cases that the desired QoS may change depending on the usage patterns at any given time, e.g., some use cases may require a great volume of data (high precision) even if this comes at the expense of latency (timeliness); other use cases may reverse these priorities.

It is desirable, therefore, that software systems be able to adapt to these varying resources and needs. Because of this, many existing systems already incorporate specialized adaptations (typically ad-hoc and local to a subsystem) to adapt to at least some of these variations. In other cases, adaptive behaviors are encoded in more general-purpose software; for example, the TCP protocol will adjust its data rate upward as bandwidth becomes available to support transmission of more data, and downward when not enough bandwidth is available to transmit data at the current rate. Such adaptations tend to be "one size fits all," however, and prove to be poor behaviors for certain applications.

2.2. The Benefits of a Framework

Adaptation is made more complicated by application-level issues. For example, consider a video-editing system in which the user can fast-forward to a desired section of the video and then copy that section frame by frame to a new file. During the fast-forward mode, the most important performance characteristic may be that the position in the video (measured in seconds since the beginning) advance at a constant rate. The number of frames actually transmitted during any given period of real time is less critical, provided the user is shown enough frames to detect what scene or action is being shown. Once the "copy" mode is entered, however, it is critical to copy every frame, even if it is not possible to do so at the normal speed of motion of the video.

Protocols such as TCP and UDP can be configured to adapt within a reasonable set of parameters in either one of the two application modes described above. The difficulty is that this application must be able to switch between one mode and the other during its execution. This complicates the application's interface to network facilities; the code that implements the "fast forward" and "copy" functions must be tangled up with code to achieve the needed QoS at any instant in the communications protocols. The complexity of this greatly increases when other considerations (e.g., running in different computing environments, or other user preferences) are taken into account. It is therefore desirable to *separate the concerns* of the program's functional specification and these condition-dependent optimizations of QoS. Separation of concerns is the primary objective of Aspect-Oriented Programming [3].

The use of an appropriate framework to handle these QoS concerns alongside the functional code created by the application developer, enables a separation of concerns. As we will see, this results in code that is much clearer, is developed at a greater speed, and is easily modifiable to meet new performance requirements.

2.3. The Benefits of QuO

Quality Objects (QuO) is a distributed object computing (DOC) framework designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time.

In a client-to-object logical method call over a typical DOC toolkit, a client makes a logical method call to a remote object. In a traditional CORBA application, the client does this by invoking the method on a local ORB proxy. The proxy marshals the argument data, which the local ORB then transmits across the network. The ORB on the server side receives the message call, and a remote proxy (i.e., a skeleton) then unmarshals the data and delivers it to the remote servant. Upon method return, the process is reversed.

A method call in the QuO framework is a superset of a traditional DOC call, including the following components:

- Contracts specify the level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.

• Delegates act as local proxies for remote objects. Each delegate provides an interface similar to that of the remote object stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.

• System condition objects provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.

In addition to traditional application developers (who develop the client and object implementations) and mechanism developers (who develop the ORBs, property managers, and other distributed resource control infrastructure), QuO applications involve another group of developers, namely QoS developers. QoS developers are responsible for defining QuO contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of QoS developer, we have developed a QuO toolkit, consisting of the following components:

• Quality Description Languages (QDL) for describing the QoS aspects of QuO applications, such as QoS contracts and adaptive behavior, described in [5, 6].

• The QuO runtime kernel, which coordinates evaluation of contracts and monitoring of system condition objects, described in [14].

• Code generators that weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program, discussed in [5].

The QuO contract offers a number of powerful abstractions for programming QoS in a DOC application. These include *regions*, which abstract the notion of regions of operation which may depend on user preferences or on the condition of the computing environment (as reflected by the system conditions). The QuO contract may also contain *states*, an abstraction on which one can program a state machine whose inputs are the changing system conditions.

The QuO middleware currently supports CORBA applications in C++ and Java, and Java RMI applications.

3. The Unmanned Air Vehicle application

As part of an activity for the US Navy at the Naval Surface Warfare Center in Dahlgren, Virginia, USA, we have been developing a prototype concept application for use with an Unmanned Aerial Vehicle (UAV). A UAV is a remote-controlled aircraft that is launched in order to obtain a view of an engagement, performing such functions as spotting enemy movements or locating targets. A UAV can receive remote-control commands from a ship in order to perform such actions as changing its direction of flight or directing a laser at a target.

The prototype supports the UAV concept of operation by disseminating data from a UAV throughout a remotely located ship. As shown in Figure 1, there are several steps to this process:

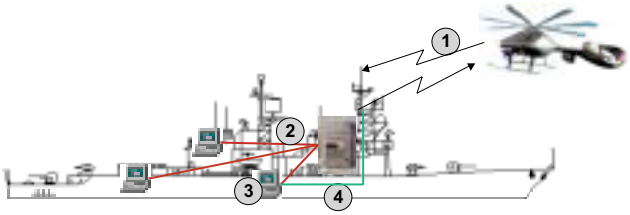

Figure 1. Typical UAV operation

1. Video feed from off-board source (UAV).
2. Distributor sends video to hosts on ship's network.
3. Users' hosts receive video and display it.
4. Users analyze the data and send commands to the UAV to control it.

Our prototype simulates the first three of these steps. The command phase of the fourth step is observed as a requirement to be able control the timeliness of data displayed on a user's video monitor: if the data is too stale, it will not represent the current situation of the physical UAV and the scene it is observing, and the user cannot control the UAV appropriately. Hence, for example, for such uses it is not acceptable to suspend the display during a period of network congestion and resume the display from the same point in the video flow when bandwidth is restored.

3.1. Prototype architecture

Figure 2 illustrates the initial architecture of the demonstration. It is a three-stage pipeline, with an off-board UAV sending MPEG video to an on-board video distribution process. The off-board UAV is simulated by a process that continually reads an MPEG file and sends it to the distribution process. The video distribution process then sends the video frames to multiple video display processes.

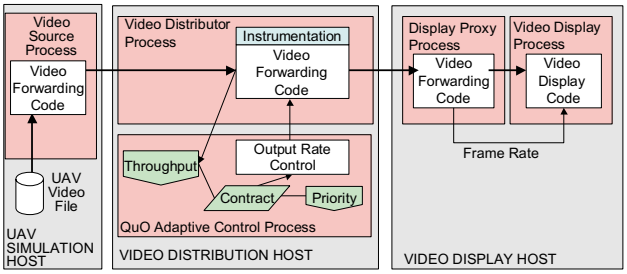

Figure 2. UAV Prototype Architecture

QuO adaptation is used as part of an overall system concept to provide load-invariant performance. Video displays located throughout the ship must display the current images observed by the UAV with acceptable fidelity, regardless of

the network and host load, in order for the shipboard operators to achieve their missions (e.g., flying the UAV or tracking a target). There are several ways to achieve this goal by appropriate adaptations to various conditions of the system. Among the possible adaptive strategies are:

- Send a reduced amount of data, e.g., by dropping frames of the video. The resultant video appears as if the camera had simply captured fewer images per second, without affecting the speed at which objects in the scene move.
- Move the distributor from an overloaded host to a different host where more CPU is available.
- Use a bandwidth reservation protocol to ensure that the distributor is able to send the necessary data to the viewers through the network, even when the network is congested.

3.2. A/V Streams transport

All remote method calls in this architecture are made via TAO, the real-time ORB developed by the Distributed Object Computing Group at Washington University in St. Louis [12]. However, in early versions of the prototype, ad-hoc TCP connections were made between the processes in order to transmit video data. This made reconfiguration of the system processes (*e.g.*, changing the number of processes or their locations) difficult, as it was necessary for each process to know the specific hosts and ports that would be used to establish each connection. In current versions, we have replaced this flow connection setup between the various processes with the TAO A/V Streaming Service [8]. This is an implementation of the CORBA A/V Streaming Service [9], which supports multimedia applications, such as video-on-demand. The TAO A/V Streaming Service is layered over TAO and ACE [13], which handle flow control processing and media transfer, respectively.

The CORBA A/V Streaming Service controls and manages the creation of streams between two or more media devices. Although the original intent of this service was to transmit audio and video streams, it can be used to send any type of data. Streams are terminated by endpoints that can be distributed across networks and are controlled by a stream control interface, which manages the behavior of each stream.

The CORBA A/V Streaming Service combines (1) the flexibility and portability of the CORBA object-oriented programming model with (2) the efficiency of lower-level transport protocols. The stream connection establishment and management is performed via conventional CORBA operations. In contrast, data transfer can be performed directly via more efficient lower-level protocols, such as ATM, UDP, TCP, and RTP. This separation of concerns addresses the needs of developers who want to leverage the language and platform flexibility of CORBA, without incurring the overhead of transferring data via the standard CORBA inter-operable inter-ORB protocol (IIOP) operation path through the ORB.

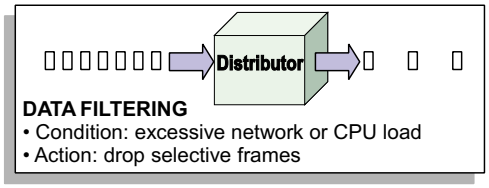

Figure 3. Adaptation by filtering frames

4. Adaptation in UAV

In this section, we discuss some performance issues in our UAV concept application, and adaptive behaviors that address these issues.

A bottleneck may occur in the application because at some point along the video transport path there are not enough resources to send the entire video to the viewers in real time. For example, the distributor host may not have enough CPU available to dispatch video frames to all viewers at that rate, or there may be insufficient bandwidth in the network path to one or more viewers. In either of these cases, we detect the bottleneck by tracking the number of frames received by the distributor and the number of frames displayed by each viewer, and comparing them. Also, if the transport provides some form of back pressure, e.g., as in our TCP version, we can measure the rate at which the distributor is able to send frames to viewers.

One adaptation to these conditions is simply to reduce the amount of data being sent. Depending on user requirements, it may be possible to omit some frames of the video entirely, resulting in an end-user video that displays the motion of the scene in real time (i.e., objects that move across the real-life scene at constant speed appear to move at constant speed in the video), but without the total illusion of continuously displayed motion that can be attained at frame rates of 24 frames or more per second. Figure 3 shows a mechanism for reducing the number of frames in the video stream. For example, if the distributor receives a video at 30 frames per second, it can reduce resource usage by deleting two of every three frames to produce video output at 10 frames per second.

Alternatively to reducing the number of frames, the amount of data per frame might be reduced. This would typically reduce the image quality of each frame.

A second adaptation is to move the distributor to a host that does not suffer the bottleneck, either because of better network location or because of greater available CPU resources. Figure 4 illustrates this adaptation. A new instance of the distributor must be started on the new host, and new communication paths must be formed between the

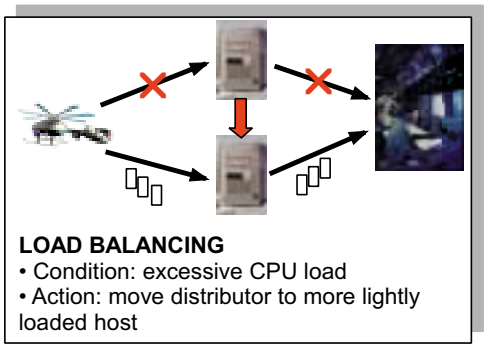

Figure 4. Adaptation by moving distributor

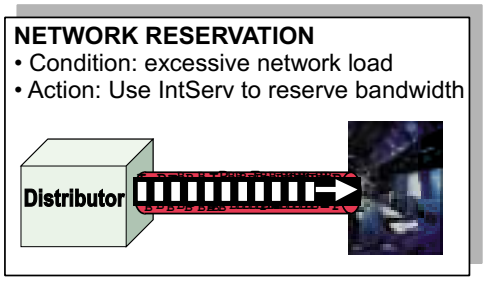

Figure 5. Adaptation by reserving bandwidth

UAV, the new distributor instance, and the viewer to replace the corresponding paths that led through the old distributor instance. The old distributor can then be halted and its paths torn down.

A third adaptation applies when the bottleneck is due to competing data flows that take up some of the network bandwidth needed by the UAV. This adaptation reserves a certain amount of network bandwidth (using the Resource Reservation Protocol (RSVP)) for the distributor's communication paths so that a sufficient rate of data can be transmitted. Figure 5 illustrates this adaptation.

It is also possible to combine these adaptations in various ways. For example, it might be necessary not only to move the distributor to a new host, but also to send a reduced video flow (e.g., fewer frames) to certain viewers through RSVP-enabled links.

Which adaptations should be used can depend on the mission-critical tasks that each user of the system has to perform. There may be many simultaneous uses of a given video flow that passes through a given distributor, each with a unique set of requirements. For example, the video might be used simultaneously to guide the UAV's flight, to control on-board systems (such as a laser to "paint" a target), or to collect reconnaissance data. Some of these uses may require only a few images per second, others depend critically on getting all video frames sent by the UAV; some may require a high-resolution, wide-angle view, while others need only a part of the image, or lower resolution. Some tasks may have higher priority than others, and so should be the last to suffer degradation of service, and these priorities may change dynamically during a mission. QuO provides the flexibility to accommodate such a diverse set of uses: different sets of requirements can be embodied in different contracts. QuO also provides an interface for a resource manager to control the use of resources by the application; for example, the resource manager can set the value of a special system condition that causes the application to relinquish resources that are needed by other applications in the system.

4.1. Adaptation in a video domain

At the beginning of Section 4, we presented three adaptive behaviors. Of these behaviors, load balancing and network reservation can be implemented without regard for the details of the video encoding; an alternative encoding can be employed without needing to change these behaviors. The implementation of data filtering to reduce the volume of video data, however, is highly dependent on the video encoding format itself.

In order to perform data filtering in the UAV prototype, we employ the technique of reducing the frame rate transmitted from the distributor to the viewer. Similar techniques can be applied elsewhere in the data path, of course, in particular between the UAV itself and the distributor. (To reduce the quality of the individual frames displayed, it is necessary to trans-code the contents of the frames themselves; tools to perform this task exist but are not yet used in our application). But the frame rate must not be reduced in such a way as to create a "slow motion effect"; that is, a vehicle that crossed the field of view of the UAV camera in 2.5 seconds should cross the application display in 2.5 seconds, and so forth for all other action in the video, in order that the display continue to present a true and up-to-the-moment view from the UAV itself. For the purposes of experiments performed on our prototype, therefore, we assumed that the UAV transmits video data at the rate of 30 frames per second, which is received by the distributor at that rate (when system resources permit), but the distributor implements an adaptive behavior that sends out a smaller number of frames representing the action that occurs during each second. The subset to be sent is selected by *dropping* (eliminating) some frames from the video, and sending out the remaining frames at a reduced rate.

Our options for efficiently dropping frames are limited by the MPEG encoding format. MPEG-1 [1], which we use in this project, utilizes three distinct types of frame. An *I-frame* (standing for "intraframe") is a compressed image of a single frame. A *P-frame* (standing for "predictive") contains only the data necessary to correctly extrapolate an image from a previously-displayed frame (an I-frame or another P-frame, and which may or may not have been the im-

mediately preceding frame). A *B-frame* (standing for "bidirectional") requires the images from two other frames (either of which can be an I-frame or a P-frame), one displayed prior to and one displayed after the B-frame. In a typical 30-frame-per-second MPEG encoding, each *group of pictures* (GOP) consists of a single I-frame, four P-frames, and ten B-frames, as shown in Figure 6.

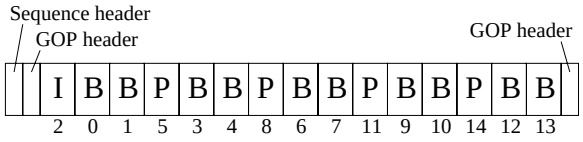

Figure 6. Sequence of frames in MPEG file

Our adaptation strategies must consider this encoding scheme. For example, if we could simply drop every second frame, we would be left with 15 frames out of every 30, and could send the video at the rate of 15 frames per second without affecting the apparent speed of motion of the scene. (At this rate the human eye would be able to detect a slight flickering or stroboscopic effect as one image was replaced by the next, because the 1/15 second interval between images is a little longer than the threshold for distinguishing successive still images from true continuous change in the scene.) This particular example is impractical in the videos we worked with, however, because some of the frames dropped would have been I-frames, but 16 other consecutive frames (all other frames in the same GOP, and the first two B-frames in the next GOP) depend directly or indirectly on each I-frame, so dropping a single I-frame results in more than 0.5 second of the video being lost.

Further, in support of the application, one of whose requirements is to track moving images as continuously as possible, it is highly desirable to minimize any intervals in the video during which the image remains still. Hence it is not desirable to display, say, the I-frame and eight subsequent frames in a GOP, and drop the remaining six frames. While this scheme incurs the load entailed in transmitting 18 frames per second, it suffers intervals of 7/30 second during which no motion is seen; this is a longer interval than if the video were displayed at a steady rate of only 5 frames per second (6/30 second between frames). From a human-factor point of view, it is desirable that the intervals between the correct display times of frames be as uniform as possible.

Because of these issues and the dependencies between frames, the best frame-dropping protocols drop B-frames when only a few frames are to be dropped (because a missing P-frame implies an interval of at least 1/5 second —six frames— between the correct times of displayed frames). There are 20 B-frames in each second of video, so this technique can bring the sending rate down to 10 frames per second. To drop more frames, P-frames can then be dropped. I-frames should be dropped only if intervals of 1 second or more between images are acceptable.

If each frame in the MPEG format contained a timestamp showing the time at which it is supposed to be displayed (e.g., expressed in milliseconds from the time when the video started), it would be relatively easy to delete frames from the video one at a time without causing jitter (frames displayed sooner or later than the correct time). But the frames do not include such a timestamp, so there are fewer good options for dropping frames. For example, if we were to drop one frame out of every three, we would be left with a video in which, to minimize jitter, some images in a sequence should be displayed 1/30 second after the previous image, and some 1/15 second after. But the typical MPEG video player is designed to read a single frame rate from the sequence header and to display all frames in the sequence at that constant rate; indeed the MPEG format is designed to support nothing more sophisticated.

A technique for "dropping" frames without incurring this display-timing difficulty is to replace B- or P-frames with "dummy" frames rather than dropping them entirely [2]. The dummy frame contains only the minimal information to allow the viewer to display an image similar to the preceding (or following) image, but omits all the new image information that the B- or P-frame itself would have contained; hence this frame is very small, and the bandwidth required to transmit the video is reduced, although not as much as if the frame were eliminated entirely.

For our current implementation we chose to drop frames entirely in such a way that the remaining frames are displayed at a constant rate. This implementation provides us with three different levels of QoS among which to adapt the application, as determined by the frame rate:

- 30 frames per second. This is done by transmitting the video intact. When this rate is achieved it represents the highest level of QoS.
- 10 frames per second. This is done by dropping all B-frames from the video, and transmitting all the I- and P-frames. At this level of QoS, most perception of motion in the video scene is preserved, but a careful human observer can detect by eye the transitions from one image to the next.
- 2 frames per second. This is done by dropping all P- and B-frames from the video, and transmitting all I-frames. At this level of QoS, the image changes frequently enough for some motion to be judged, but finer details of motion (and some very short-lived actions) can be lost entirely.

It is then possible to adaptively switch among these three frame rates by assigning each frame rate to a different region of a QuO contract, and setting the frame-dropping protocol at any given time according to the current region. (We also implemented frame rates of 1 frame per second and slower, but due to their extremely low levels of fidelity we excluded these from our adaptive behaviors.)

In actual MPEG videos we examined, mean sizes of I frames were in the range between 11500 and 14000 bytes, of P frames in the range between 4900 and 7000 bytes, and of B frames in the range between 2900 and 3400 bytes. In the sample video provided by the Navy and selected as the test case for the UAV application, I-frames averaged approximately 13800 bytes, P-frames approximately 5000 bytes, and B-frames approximately 2900 bytes. The approximate size in bits of two average GOPs is therefore

$$(2(13800) + 8(5000) + 20(2900)) \cdot 8 = 1004800$$

(*i.e.*, near the capacity of a 1.5 Mbit link). This is the bandwidth requirement of sending one second of the video at the full rate of 30 frames per second.

If we drop the rate to 10 frames per second by eliminating the B-frames, the bandwidth required, in bits per second, falls to approximately

$$(2(13800) + 8(5000)) \cdot 8 = 540800$$

and if we drop the rate to 2 frames per second by eliminating the P-frames as well, the required bandwidth in bits per second falls to approximately

$$2(13800) \cdot 8 = 220800.$$

That is, reducing the frame rate from 30 to 10 (a 67 percent reduction) reduces the bit rate by 46 percent, and reducing the frame rate from 30 to 2 (a 93 percent reduction) reduces the bit rate by 78 percent. These are substantial reductions of bandwidth and other system requirements, so it is not hard to find system conditions under which the full bandwidth is not supportable, but one of the reduced-bandwidth adaptations is. The reduction in bit rate is not proportional to the reduction in frame rate, because the frames that must be dropped first are precisely those frames that have the greatest dependency on other frames (and the fewest frames depending on them), and consequently the encoded sizes of these dropped frames are relatively small. On the other hand, reduction in the perceived value of the reduced-frame-rate display to a human viewer also is not proportional to the reduction in frame rate, judging from the informal reactions of people who have watched demonstrations of the application adapting.

4.2 QuO Contract

Figure 7 shows a QuO contract that adapts the distributor to available resources by increasing or decreasing the rate at which frames are transmitted to viewers. We show it to illustrate the high-level adaptation-oriented abstraction that QuO provides, and to illustrate the isolation of these adaptive behavior aspects from the rest of the code. This contract is substantially similar to the one used in the prototype, but

```
contract UAVdistrib (
  syscond quo::ValueSC timeInRegion,
  syscond quo::ValueSC actualFrameRate,
  callback InstrCallback instrControl,
  callback SourceCtrlCallback sourceControl)
{
  region NormalLoad (actualFrameRate >= 27)
    { }
  region HighLoad ((actualFrameRate < 27 and
                    actualFrameRate >= 8))
    {
      state Duty until (timeInRegion >= 3)
                (timeInRegion >= 30 -> Test)
        { }
      state Test until (timeInRegion >= 3)
                (true -> Duty)
        { }
      transition any->Duty {
        sourceControl.setFrameRate(10);
        timeInRegion.longValue(0); }
      transition any->Test {
        sourceControl.setFrameRate(30);
        timeInRegion.longValue(0); }
    }
  region ExcessLoad (actualFrameRate < 8)
    {
      state Duty until (timeInRegion >= 3)
                (timeInRegion >= 30 -> Test)
        { }
      state Test until (timeInRegion >= 3)
                (true -> Duty)
        { }
      transition any->Duty {
        sourceControl.setFrameRate(2);
        timeInRegion.longValue(0); }
      transition any->Test {
        sourceControl.setFrameRate(10);
        timeInRegion.longValue(0); }
    }
  transition any->NormalLoad {
    instrControl.setRegion("NormalLoad");
    sourceControl.setFrameRate(30); }
  transition any->HighLoad {
    instrControl.setRegion ("HighLoad"); }
  transition any->ExcessLoad {
    instrControl.setRegion ("ExcessLoad"); }
};
```

Figure 7. QuO Contract for UAV

with certain details of syntax and non-essential functionality elided in order to fit it legibly within a single column of text. The contract divides the operational conditions of the distributor into three QuO regions:

- **NormalLoad:** Entered when resources are adequate to transmit the video to the viewers at the full bit rate. In this region, the distributor sends 30 frames per second.

- **HighLoad:** Entered when there are not adequate resources to transmit the video at the full bit rate. In this region, the distributor sends 10 frames per second. Intermediate frames of the video are dropped so that the remaining frames, displayed at the rate of 10 per second, depict normal-speed motion.

- **ExcessLoad:** Entered when there are not adequate resources to transmit the video even at the reduced bit rate required for 10 frames per second. In this region, the distributor sends 2 frames per second, again dropping intermediate frames in order to preserve the speed of motion.

This contract communicates with the rest of the system in several ways. First, the system condition object `actualFrameRate` is set periodically by the distributor; its value is the actual number of frames sent in the previous second. The contract uses this system condition to gauge the amount of data that the distributor has resources to send. This particular measurement is quite general-purpose; whether the restricted resource is bandwidth, CPU, or an I/O device, to the extent that this restriction affects the ability of the distributor to transmit video at its desired frame rate, the deficiency will be detected in the form of a reduced actual frame rate. The prototype implementation averages the rate over a period of one second; a shorter period may be practical, but the rate must be averaged over some period in order to be measurable and accurate. An alternative that might provide quicker reaction is to monitor more basic system conditions such as processor load and network load, and to *predict* when the achievable frame rate is likely to be reduced rather than merely *observing* it, but such schemes entail tradeoffs, such as the greater complexity of calibrating the predictions, and the failure to detect performance problems caused by conditions that are not measured.

A disadvantage of estimating the capacity to send frames by measuring only the frames actually sent is that it is difficult to detect when there is excess capacity (and when the frame rate might safely be increased). This contract addresses this problem by occasionally attempting to send the video flow at the next higher frame rate from its current setting. The frequency and duration of these "tests" is controlled by a state machine within the current region, which alternates between the Duty and Test states at certain intervals of time. The time in any state is measured by the system condition `timeInRegion`, which is set to zero every time there is a state transition and is thereafter incremented once per second.

Based on the value of `actualFrameRate`, then, the contract selects the correct region from among NormalLoad, HighLoad, and ExcessLoad, which in turn controls the frame rate via execution of the `sourceControl.setFrameRate` callback, which is called on transition to the NormalLoad region or to the Duty states of the other two regions. Then, if the contract is in a Duty state, after the value of `timeInRegion` passes a predetermined threshold the contract will transition to the Test state for a few seconds, at which time it sets a higher frame rate. The **until** clause of each state prevents any transitions out of that state for a few seconds, ensuring that a stable measurement of the new achievable frame rate is made; at the end of this time, if the test succeeds (that is, if the actual frame rate is observed to be at the requested rate) a contract reevaluation results in a change of regions. Otherwise, the contract begins a new Duty cycle in the same region (unless, of course, insufficient resources for the HighLoad region force a transition down to the ExcessLoad region).

The `instrControl` callback enables the contract to communicate with a resource manager that monitors and controls the resource usage and location of application processes. We installed the prototype in an environment controlled by such a resource manager. The transition into the ExcessLoad region caused the contract to execute the code in `transition any->ExcessLoad`, which in turn transmitted an indicator of the region to the resource manager. The resource manager then restarted the distributor on a different host where more resources were available.

5. Results

5.1. Adaptation controls latency

We performed experiments to test the effectiveness of our adaptive behavior in the UAV application. We ran the three stages on three Linux boxes, each with a 200MHz processor and 128MB of memory. The video transport was TCP sockets.

At time $t = 0$, the distributor started. Shortly after this, the video began to flow. At $t = 60$ seconds, $t = 62$ seconds, and $t = 64$ seconds, we started three load-simulating processes on the same host as the distributor, each attempting to use 20 percent of the maximum processing load (a total of 60 percent additional processing load). This reduced the distributor's share of processing power below what it needed to transmit video at 30 frames per second. At $t = 124$ seconds, we removed the load. At time $t = 300$ seconds (approximately), the experiment terminated. The basic premise is that the full load was applied for a duration of one minute, starting after the pipeline had had time to "settle in," and ending a few minutes before the end of measurement so we could observe any trailing effects.

This scenario was run twice, once without QuO attached and without any adaptation (the control case) and once with a QuO contract causing adaptation (the experimental case). For the purposes of this experiment, the only adaptation enabled was to reduce bandwidth by dropping frames.

Figure 8 shows the effect of the increased load on the latency of the video stream. In this graph, the x-axis represents the passage of time in seconds from the start of the video and the y-axis represents the "lateness" of each image in seconds, i.e., the additional latency (in delivery to the viewer) caused by the system load. That is, if all images were delivered with the same latency, the graph would be a constant zero. The label "Load" indicates the period of time during which there was contention for the processor. Without QuO adaptation, the video images fall progressively further behind starting when the contention first occurs, and the video does not fully recover until some time after the contention disappears.

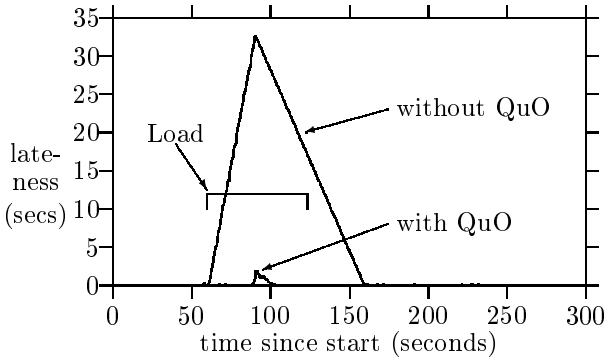

Figure 8. Effect of adaptation on latency

Adaptation	Lateness (sec)	
	Mean	Maximum
No (without QuO)	5.400	32.696
Yes (with QuO)	0.067	1.930

Table 1. Mean and maximum frame lateness

Table 1 summarizes these results. The lateness values in all these figures are based on the timing of the I frames, which occur 2 times per second and (ideally) are not biased between non-adaptive and adaptive cases since our adaptations never drop I frames. (We also obtained numbers for all frames, and they are similar to these.) Lateness for each frame was calculated by counting the I frames seen so far and computing how long it should have taken to get to the nth frame. The maximum lateness, then, is the greatest delay between the time we expected a frame to be displayed and the time that frame actually was displayed. The mean lateness is dependent on the period over which it was averaged (in this case, five minutes), but assuming the same period is used in all cases, the mean lateness is a legitimate figure of merit, with lower values representing better performance.

The outcome of this experiment demonstrated that adaptation leads to improved performance of the application. The added latency caused by adverse system conditions (in this case, excessive CPU load) occurs in a sharply reduced magnitude and duration when adaptation is enabled, and the video image is continuously usable for its intended real-time purpose despite the fluctuation.

5.2. Software engineering with QuO

The effectiveness of QuO as a software engineering framework is exemplified in Table 2. In this table, "display time of frames" refers to issues concerning the rate at which frames are displayed. These issues arose because the viewer implementation we used mixed QoS concerns in an ad-hoc fashion with application function code. Because of this manual tangling of concerns, the code was unnecessarily complex and it was difficult to modify or even fully understand its behavior. Adaptation of the program to new performance criteria (i.e., fast local decoding and display hardware but variable quality of video input, as opposed to its original domain in which the video input was of uniform quality but decoding and display could suffer delays) required an unacceptably high investment of effort by programmers. On the other hand, changing the parameters of behaviors controlled by QuO often took only minutes (for a simple change in, say, a threshold value) to a day (for more substantial changes in behaviors exhibited).

Property Adapted	Coding Method	Time to develop
Display time of frames	Ad hoc	Months (or never)
Transmission load	QuO	Hours

Table 2. Impact of QuO framework on development time

6. Related work

Hemy *et al.* present an adaptive MPEG transmission application that also does frame-dropping, but in a slightly different way [2]. Where we delete frames entirely, they insert "dummy" frames into the MPEG flow in order to replace the dropped frames. In this way they achieve most of the possible reduction in bandwidth without changing the frame rate used by the viewer (although the rate of sending new images is reduced by the same factor as in the UAV prototype).

The *Agilos* middleware project implements a hierarchical adaptive QoS control architecture [4], similar to QuO in some ways. However, QuO supports application-specific, local adaptation as well as cooperative adaptation across applications using shared system condition objects, contracts, and resource or property managers. In contrast, Agilos supports a global, control-theoretic mechanism incorporating all applications in an environment, including possibly unrelated applications, for application-neutral resource control.

7. Future work

We are currently implementing bandwidth reservation for the MPEG flows over A/V Streams, including developing QuO contracts for the UAV that use the full set of adaptations described in Section 4. This will allow us to test UAV adaptation under conditions in which multiple system conditions (such as CPU load and network congestion) vary independently, and in which multiple adaptive strategies (such as bandwidth reservation and frame dropping) may be combined.

We have recently implemented support for multiple viewers, dynamically created and connected at run time, and independently adaptable. We also plan to support multiple distinct video flows in the system involving multiple sources and distributors. This will enable us to investigate the interaction of multiple adapting entities in a complex system.

QuO itself is still an evolving framework —for example, new syntax for states and for "locking down" regions was added to the contract language in order to more clearly express contract features used by the contract of the UAV prototype— and the development of the UAV application will continue interacting with the evolution of QuO, both as a testing ground for improvements to the framework and as a driver of more innovations.

8. Conclusion

In this paper, we described a prototype multimedia application, the UAV video distribution system, and the standards (MPEG video and CORBA A/V Streams) we used to implement it. We described QuO, a framework for distributed object computing designed to enable adaptive optimization of distributed software systems. We demonstrated how QuO can interact with distributed-object applications —even one in which the communication path to be adapted is not a client-server method call and return— in order to implement application- and implementation-specific adaptations to system performance issues. Further, we presented empirical results that show that QuO adaptation significantly improves application performance in the presence of system load. Moreover, our experience was that the use of the QuO framework made the implementation and redesign of adaptive behaviors easier for developers than ad-hoc methods typically used, because it enables adaptive behaviors to be separated from the basic video functions, making them easier to understand and modify.

Acknowledgements

This work is sponsored by DARPA and US AFRL under contract nos. F30602-98-C-0187 and F33615-00-C-1694. The authors would like to gratefully acknowledge the support of Dr. Gary Koob, Thomas Lawrence, and Mike Masters for this research. The authors would also like to acknowledge the contributions of Michael Atighetchi, Tom Mitchell, John Zinky, and James Megquier, the Naval Surface Warfare Center (NSWC) Dahlgren, VA (in particular Paul Werme, Karen O'Donoghue, David Alexander, Wayne Mills, and Steve Brannan), and the DOC groups at Washington University, St. Louis, and University of California, Irvine, to the research described in this paper.

References

[1] D. L. Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, April 1991.

[2] M. Hemy, P. Steenkiste, and T. Gross. Evaluation of adaptive filtering of MPEG system streams in IP networks. In *IEEE International Conference on Multimedia and Expo 2000*, New York, New York, 2000.

[3] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[4] B. Li and C. Nahrstedt. *QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications.* Springer-Verlag, 2000.

[5] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. Karr, R. Vanegas, and K. R. Anderson. *QoS Aspect Languages and Their Runtime Integration.* Springer-Verlag, 1998.

[6] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, April 1998.

[7] S. Microsystems. The real time specification for java. Internet URL http://java.sun.com/aboutJava/ communityprocess/first/jsr001/.

[8] S. Mungee, N. Surendran, and D. C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[9] Object Management Group. *Control and Management of Audio/Video Streams: OMG RFP Submission*, 1.2 edition, Mar. 1997.

[10] OMG. *Control and Management of Audio/Video Streams, OMG RFP Submission (Revised), OMG Technical Document 98-10-05*. Object Management Group, Framingham. MA, Oct 1998.

[11] OMG. *CORBA 2.4 Specification, OMG Technical Document 00-10-33*. Object Management Group, Framingham. MA, Oct 2000.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

[13] D. C. Schmidt and T. Suda. An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems. *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, 2:280–293, December 1994.

[14] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.