Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications

Joseph Loyall, Richard
Schantz, John Zinky, Partha
Pal, Richard Shapiro, Craig
Rodrigues, Michael
Atighetchi, David Karr
BBN Technologies
jloyall@bbn.com

Jeanna M. Gossett
The Boeing Company
Jeanna.Gossett@mw.boeing.
com

Christopher D. Gill
Washington University,
St. Louis
cdgill@cs.wustl.edu

Abstract

The Quality Objects (QuO) middleware is a set of extensions to standard distributed object computing middleware that is used to control and adapt quality of service in a number of distributed application environments, from wide-area to embedded distributed applications. This paper compares and contrasts the characteristics of key use cases and the variations in QuO implementations that have emerged to support them. We present these variations in the context of several actual applications being developed using the QuO middleware.

1. Introduction

Distributed Object Computing (DOC) middleware has emerged and gained acceptance for the development and implementation of a wide variety of applications in a wide variety of environments. As DOC middleware has gained acceptance and has been applied to a broader variety of use cases, there has been a natural growth in extensions, features, and services to support these use cases. For example, the Minimum CORBA specification [15], the Real-time CORBA 1.0 specification [16], and the Real-Time Specification for Java (RTSJ) [2] are examples of extensions and services that have grown out of a need to support embedded and real-time applications.

We have developed a DOC middleware extension called Quality Objects (QuO) [24], which supports adaptive quality-of-service (QoS) specification, measurement, and control, and which we have described in a number of earlier papers. QuO is being used in a number of demonstrations and applications, ranging from wide-area distributed applications to embedded real-time systems. These diverse use-cases have led to a natural set of usage patterns, tailorings, and enhancements to the QuO middle-ware that has simultaneously broadened its applicability and refined its focus on the specific problems of particular environments.

This paper describes several applications developed using QuO middleware and compares and contrasts the usage patterns exhibited by them. We describe the particular flavors of QuO that have been developed to support the characteristics of these use-cases. Section 2 provides a brief overview of the QuO middleware. More detail about QuO can be found in [12, 13, 17, 18, 21, 24]. Section 3 describes, compares, and contrasts the various usage patterns that have emerged for the QuO middleware. Section 4 describes the different implementations of QuO available for specific use-cases. All of these implementations provide similar QuO functionality and features, but with characteristics tailored for the specific use-cases. Section 5 describes three specific applications being developed using QuO middleware that provide concrete examples of the use-cases described in Section 4. Section 6 discusses some issues arising from the different implementations. Finally, Section 7 provides concluding remarks.

2. Overview of the adaptive QuO middleware

Figure 1 illustrates a client-to-object logical method call. In a traditional CORBA application, a client makes a logical method call to a remote object. A local ORB proxy (i.e., a stub) 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.

Quality Objects (QuO) is a distributed object computing (DOC) framework designed to develop distributed ap-
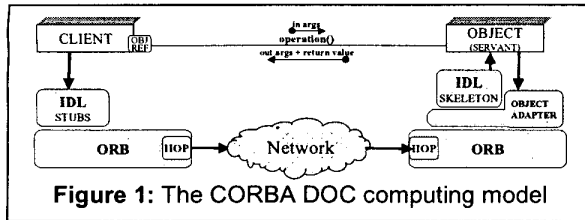
Figure 1: The CORBA DOC computing model

plications 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. By providing these features, QuO opens up distributed object implementations [11] to control an application's functional aspects and implementation strategies that are encapsulated within its functional interfaces.

A method call in the QuO framework is a superset of a traditional DOC call, and includes the following components, illustrated in Figure 2:

- *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, QuO applications may use property managers and specialized ORBs. Property managers are responsible for managing a given QoS property (such as the availability property via replication management [5] or controlled throughput via RSVP reservation management [1]) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects. In some cases, the managed property requires mechanisms at lower levels in the protocol stack. To support this, QuO includes a gateway mechanism [18], which enables special purpose transport protocols and adaptation below the ORB.

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 be-

havior. To support the added role of QoS developer, we are developing a QuO toolkit, described in earlier papers such as [12], [13] and [21], and consisting of the following components:

- *Quality Description Languages (QDL)* for describing the QoS aspects of QuO applications, such as QoS contracts (specified by the Contract Description Language, CDL) and the adaptive behavior of objects and delegates (specified by the Structure Description Language, SDL). CDL and SDL are described in [12, 13].
- The *QuO runtime kernel*, which coordinates evaluation of contracts and monitoring of system condition objects. The QuO kernel and its runtime architecture are described in detail in [21].
- *Code generators* that weave together QDL descriptions, the QuO kernel code, and client code to produce a single application program. Runtime integration of QDL specifications is discussed in [12].

3. Usage patterns of the QuO adaptive middleware

CORBA and other DOC frameworks, such as Java RMI, are being used to implement diverse types of distributed applications in diverse environments, from wide-area networks such as the Internet to embedded systems [6, 8, 22]. These applications and environments exhibit different characteristics and, while they are supported by DOC middleware in general, they also use services tailored to support their specific characteristics.

For example, CORBA IDL is of general use in exposing the functional interfaces of objects, while hiding the implementation details. However, it can be argued that this is more important for heterogeneous, distributed applications, where the implementation details might include multiple languages, platforms, operating systems, and mechanisms than it is for embedded applications. Meanwhile, QuO has been designed to provide *customized* support for adaptive distributed object computing with quality-of-service requirements, above and beyond the generic
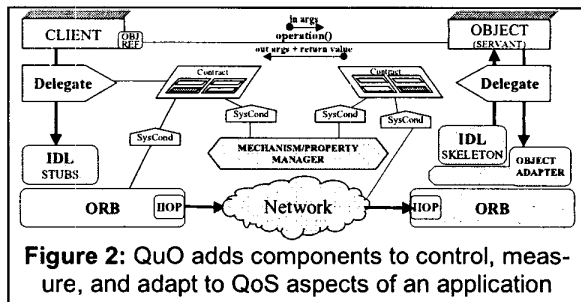

Figure 2: QuO adds components to control, measure, and adapt to QoS aspects of an application

	WAN Distributed Applications	Embedded Distributed Applications
Data Content and Size	Can vary widely	Often predictable and constrained
Network Latency	Can vary widely	Often local or non-existent
Dominant Resource	Network bandwidth	CPU cycles
Platforms	Often heterogeneous and remote	Typically homogeneous
Languages	Sometimes heterogeneous	Typically one
Object Distribution	Can be dynamic	Typically fixed and local
Object Location	Can be dynamic	Often preset and fixed

Table 1: Comparison of characteristics of WAN and (avionics) embedded applications

features of general-purpose middleware. QuO has features, services, and implementations that have emerged to support the needs of specific usage patterns. This section describes some of these usage patterns and the characteristics that QuO provides to support them. Section 4 will describe specific QuO-supported applications that provide examples of these usage patterns.

3.1 Wide-area distributed object applications

Wide-area network applications, which have gained prominence due to the emergence of the Internet and new networking technologies, have characteristics that differ significantly from traditional, non-networked or locally networked applications. In this section and the next, we contrast WAN-based distributed applications and embedded distributed applications, to motivate the QuO features that have emerged to support these two different application contexts. This discussion is summarized in Table 1.

WAN applications often utilize components that 1) exist on heterogeneous hosts, 2) are implemented in multiple languages, 3) are not discovered until runtime (e.g., through a Naming Service or DNS lookup), and 4) for which network latency is an issue as much as, or more than, CPU availability. Many WAN applications exhibit some or all of the following distinguishing characteristics:

- *Widely varying data content and size* – Servers often have little control over the amount, quality, or content of data that clients send to them.
- *Widely varying network latency times* – The distance between objects, the capacity of the networks, and the amount and size of competing traffic can all contribute to unpredictable delays in message and data delivery.
- *Application performance can be dominated by network transport times* – Higher and less predictable network latency plays a larger role in the performance of WAN applications.
- *Heterogeneity in platforms and languages* – Server objects can be written in a variety of languages and be hosted on a variety of platforms within an application. The specifics of language and platform are often hidden be-

hind a common interface language, like CORBA IDL or HTML.

- *Dynamic distribution of objects* – References to objects can be obtained dynamically, using a service such as the CORBA Naming Service, the CORBA Trading Service, or DNS. This means that objects can migrate, different objects can service subsequent requests, and so forth.

3.2 Embedded distributed object applications

In contrast, embedded distributed applications, such as avionics sensor-actuator applications, typically operate within more resource constrained, but more predictable, environments. They usually must operate within tight timing deadlines (e.g., sensor data must be processed before the next data element is acquired from the same sensor) and therefore cannot abide varying data size or content. However, since they typically exist within LANs, across a hardware bus, or on a single processor, there is significantly more predictability in resource availability, communication latency, object location, and nature of object implementation.

In this paper, we are concentrating on a class of embedded avionics and shipboard embedded applications, which exhibit some or all of the following distinguishing characteristics:

- *Predictable data content and size* – Data size is generally constrained so that it can be processed within a fixed period. Likewise, a single sensor, or a small set of sensors, generally provides data with predictable content and size.
- *Fewer variances in network latency* – Embedded applications often exist on a single processor or on a LAN, so that network latency is low and fairly predictable. There is some external data input, but most data transport between embedded components is local, with smaller, more controlled network latency times.
- *Application performance is often dominated by CPU allocation* – Scheduling the CPU so that all real-time tasks meet their deadlines is a dominant feature of many embedded applications. Message processing is equally likely to be constrained by processor contention as by network contention.

• *Homogeneity in platforms and languages* – Embedded applications typically run on only one processor or a few identical processors in a LAN, using a single operating system, and are typically written in one language.

• *Objects are typically fixed and predefined* – The numbers and types of objects are often predefined and object instances are usually local and created up front.

3.3 Event channel, periodic tasking

Sensor-actuator applications, such as those found on avionics platforms, often follow an event-driven, periodic tasking model. In such a model, an avionics application consists of many periodic tasks with real-time deadlines (traditionally all are *hard* real-time deadlines, however hybrid hard real-time/soft real-time scheduling is becoming more prevalent in embedded real-time systems). These tasks are scheduled at a particular rate and allocated the CPU at that rate. The *deadline* for each task is chosen to allot enough time for the task to perform its function (e.g., process sensor data, compute navigation heading). The *period* of tasks are chosen to ensure that all tasks can be scheduled.

The traditional DOC benefits, e.g., the hiding of implementation details behind functional interfaces and a common data transport protocol, may ease the programming of such embedded real-time applications. However, modularization and decomposition are still the primary benefits, because these embedded real-time applications do not utilize a variety of implementations, platforms, and languages. Furthermore, the real-time embedded software industry has not yet widely adopted the DOC computing paradigm [3].

Because of this, and to extend the current state-of-the-practice in real-time embedded computing, DOC services are emerging that support event-driven, periodic tasking models. Two examples of these are the real-time CORBA Event Service [10] and the real-time CORBA Scheduling Service [9] in TAO [20], a real-time CORBA compliant ORB. Another example is the real-time specification for Java (RTSJ) [2]. The use of TAO's real-time CORBA Event Service and real-time CORBA Scheduling Service in an avionics application is described in Section 5.2.

3.4 Adaptation at many levels

QuO's contracts and delegates support adaptation at many levels, from managers mediating adaptation for many applications, to adaptation within an application, to adaptive resource control mechanisms, to adaptation at the transport layer. QuO's contracts and delegates provide the adaptation that can be used within a single application and also within system managers. QuO's system condition objects provide a uniform interface to system resources,

mechanisms, and managers to translate between application-level concepts, such as operating modes, to resource and mechanism-level concepts, such as scheduling methods and real-time attributes.

Finally, QuO provides a *gateway* component, which allows low-level communication mechanisms and special-purpose transport-level adaptation to be *plugged into* an application [18]. The QuO gateway resides between the client and server ORBs. It is a mediator [7] that intercepts IIOP messages sent from the client-side ORB and delivers IIOP messages to the server-side ORB (on the message return the roles are reversed). On the way, the gateway translates the IIOP messages into a custom transport protocol, such as group multicast in a replicated, dependable system.

The gateway also provides an API that allows adaptive behavior or processing control to be configured below the ORB layer. For example, the gateway can select between alternate transport mechanisms based on low-level message filtering or shaping, as well as the overall system's state and condition objects. Likewise, the gateway can be used to integrate security measures, such as authenticating the sender and verifying access rights to the destination object.

3.5 Synchronous and asynchronous adaptation

QuO contracts and delegates support two means for triggering manager-level, middleware-level, and application-level adaptation. The delegate triggers *in-band* adaptation by making choices upon method calls and returns. The contract triggers *out-of-band* adaptation when changes in observed system condition objects cause region transitions.

Figure 3 illustrates QuO's in-band and out-of-band adaptation. The QuO delegate supports in-band adaptation (Figure 3a) whenever a client makes a method call and whenever a called method returns. The delegates (on the client and server side) check the state of the relevant contracts and choose behaviors based upon the state of the system. These behaviors can include shaping or filtering the method data, choosing alternate methods or server objects, performing local functionality, and so on.

QuO contracts and system condition objects support out-of-band adaptation (Figure 3b) by monitoring conditions in the system, whether they are the states of resources, mechanisms, or managers. Whenever the monitored conditions change (or whenever they change beyond a specified threshold), the system condition object triggers an asynchronous evaluation of the relevant contracts. If this results in a change in contract region (i.e., state), it in turn triggers adaptive behavior that occurs asynchronous to any object interactions.
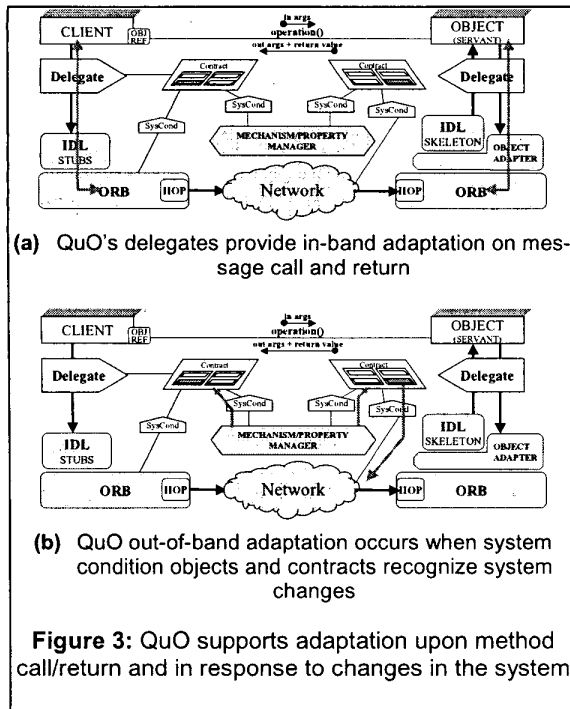
(a) QuO's delegates provide in-band adaptation on message call and return

(b) QuO out-of-band adaptation occurs when system condition objects and contracts recognize system changes

Figure 3: QuO supports adaptation upon method call/return and in response to changes in the system

System condition objects can interface to other, lower-level system condition objects, and can be either *observed* or *non-observed*. Changes in the values measured by observed system conditions trigger contract evaluation, possibly resulting in region transitions and triggering out-of-band adaptive behavior. Observed system condition objects are suitable for measuring conditions that either change infrequently or for which a measured change can indicate an event of notice to the application or system. Non-observed system condition objects represent the current value of whatever condition they are measuring, but do not trigger an event whenever the value changes. Instead, they provide the value upon demand, i.e., whenever the contract is evaluated due to a method call or return or due to an event from an observed system condition object.

This combination of observed and non-observed system condition objects, along with the nesting of system condition objects, provides flexibility to support a wide variety of in-band and out-of-band adaptation, while providing needed support to avoid instability problems and *hysteresis* effects. Observed system condition objects can measure frequently changing system conditions by *smoothing out* continuous changes (*e.g.,* by measuring statistical average of changes over time) or by reporting only when the system condition crosses a threshold. This can be implemented by a single system condition object or an observed system condition that periodically polls a non-observed system condition object monitoring the fre-

quently changing condition. The threshold can be dynamically supplied by another system condition object.

4. Implementation choices for QuO middleware

The initial prototype implementation of QuO middleware, covered briefly in Section 4.1, has been described in earlier papers [13, 17, 18, 21]. In addition, we have developed other implementations and services supporting specific use-cases of QuO. These are described in Sections 4.2 and 4.3 and have led to the following *variants* of QuO that are more suitable for particular applications. These variants advance QuO in a complementary direction to other DOC middleware, such as TAO, CORBA, and Java.

4.1 Java QuO with threading

The initial prototype of QuO had two goals that led to specific implementation decisions: (1) rapid prototyping for early baseline functionality and (2) maximum flexibility. To achieve these goals, the baseline version of QuO is written in Java, is multi-threaded, and takes maximum advantage of Java's meta-object support.

This baseline version of QuO supports multiple languages (Java and C++ clients and objects) and multiple ORBs (Visibroker and TAO). The QuO kernel, system condition objects, and contracts are implemented in Java and the QuO kernel and many system condition objects run in their own thread. Contracts are *scheduled for evaluation* by placing them on a queue and a QuO kernel thread runs in a tight loop that pulls one contract at a time from the queue and evaluates it. Contracts, regions, transitions, predicates, elements of predicates, and so on are all represented as Java objects. This facilitates runtime interpretation of contract elements and keeps the QDL languages from having to implement typing features or type inference. System condition objects maintain their values and return them immediately upon demand, thereby ensuring predictable execution times of contract region predicates.

All interactions with the QuO kernel and between QuO objects are through CORBA interfaces. Therefore, C++ clients would have a C++ delegate (which resembles a CORBA IDL stub) that checks the state of a contract with a CORBA call to the contract. We have also developed a configuration of this QuO prototype that supports the Java RMI inter-object protocol, in place of CORBA, for the DARPA Advanced Logistics Program (ALP). This provides the same QuO functionality but it is provided by RMI servants instead of CORBA servants.

	Bottleneck	Avionics	UAV
Performance	Dominated by IIOP data delivery, overhead of delegate small when compared to marshalling and delivery of data	Requires low overhead to be small percentage of task deadline	Streaming data, can incur no overhead per data item
Components	Distributed across WAN, heterogeneous hosts, heterogeneous languages	Two heterogeneous nodes, but single processor and language on each node	Multiple homogeneous nodes, with one common language
Resource Contention	Network and CPU usage can vary dynamically	Network is constrained, but main contention is between tasks for CPU cycles	Initial focus is CPU utilization, but later plans to address network contention
Threading, Distribution	Non-deterministic number and distribution of objects. Dynamic, changing number of threads.	Controlled, fixed number of threads and objects (and tasks).	Controlled, but not fixed, number of threads. Objects can be created and destroyed, but are deterministic.

Table 2: Comparison of the three example applications

The kernel can be *integrated* or *non-integrated*. An integrated kernel runs in the JVM of the client or servant (with a Java application), while a non-integrated kernel has its own JVM.

4.2 Non-threaded C++ and Java QuO

To support the application of QuO to an embedded dynamic mission planning avionics application, described in Section 5.2, we developed a *passive* C++ version of QuO. The existing Java version of QuO was not suitable for the avionics environment for the following reasons:

• *Java* – The embedded avionics environment does not have the spare memory and CPU to host a JVM and the extra Java ORB (Visibroker) needed by the Java QuO implementation.

• *Threading* – The embedded avionics environment has a fixed number of threads and closely controls access to these threads.

• *Overhead* – The overhead introduced by QuO delegates and contract evaluations must be minimized in the avionics environment, because of the lower latencies involved and the need to fit processing within a task's period. We have measured the threaded Java QuO kernel as imposing approximately 3 ms extra processing per method call on a 200 MHz Linux host using JDK 1.1.5, which is likely insignificant for a WAN application, but can be significant in a real-time application.

Accordingly, we implemented a version of the QuO kernel, contracts, and system condition library in C++ on top of the Adaptive Communication Environment (ACE) framework [19]. The QuO kernel, contracts, and system condition objects are all *passive*. They are implemented simply as function calls that get linked in with the application. Contract evaluation and QuO kernel services (such as system condition monitoring) execute in the thread of the calling process. In the case of in-band contract evaluation, the delegate call, the contract evaluation, and any system condition object processing used to evaluate contract regions execute in the thread of the client (or servant). In the case of out-of-band contract evaluation, the contract evaluation, any system condition object processing, and any triggered adaptation execute in the thread of the observed condition, e.g., a system resource manager or a host load monitor.

This model, where contract evaluations and all the actions spawned by them run in the thread of an existing client or manager, adds slightly different semantics to the QuO infrastructure over the previous prototype. In the previous prototype, system condition objects are defined as returning a value *immediately* when requested. The work needed to determine the values is done *continuously* in anticipation of their need. That is, the updating of system condition object values is performed asynchronously with respect to contract evaluation, delegate execution, and client execution. This enables contract evaluation to be bounded because system condition objects can run in other threads.

In the non-threaded model, the system condition object code executes to determine its value only when a contract evaluation accesses it to determine the current operating region. That is, the values of system condition objects are now computed in-band. This is necessary because there are no extra threads for system condition objects to run asynchronously. It still results in predictable contract evaluation time, however, as long as the system condition objects are written to execute within predictable time bounds.

The passive C++ version of QuO reduces the overhead of QuO adaptation through its reduced use of CORBA calls, its use of C++ native types instead of Java objects,

630

and the improved performance of compiled C++ over interpreted Java.

After implementing the passive C++ version of QuO, we used the same approach to rework the original prototype and develop an additional passive Java version of QuO, offering a Java choice to programmers that need strict control over the threads in their applications.

5. Examples of applications using these implementation choices

This section examines three demonstration applications that use QuO to perform adaptive QoS management. The three applications exhibit many of the characteristics of the use-cases described above and motivate the need for the various variants of QuO to support them. Table 2 summarizes the differences and similarities between these applications.

5.1 Case study 1: data dissemination in a wide-area network

This is one of the earliest examples that we developed using the QuO adaptive middleware. Dubbed *Bottleneck*, it consists of a client requesting still images from a remote data server and adapting to the response time it requires to receive the images. When round-trip image delivery and processing slows, the Bottleneck application examines resource and instrumentation data to determine whether the source of the slowdown is network or CPU degradation.

If the source of the slowdown is the network, the QuO middleware triggers adaptation to attempt to reserve bandwidth, using RSVP [23] or Darwin [4], if either is available. If this is not successful, the QuO middleware triggers application adaptation, in which the application trades off its data quantity or data quality requirements for its timing requirement, by requesting smaller images (lower data *quantity*) or lower resolution images (lower data *quality*) to reduce the amount of network traffic.

If the source of the slowdown is the CPU, the application responds by requesting unprocessed images. This reduces the load on the CPU used to process the images and enables them to be received faster, but reduces the quality of the display or analysis of the images, as illustrated in Figure 4.

The flexibility of the threaded Java QuO implementation is the best choice for this implementation. The QuO components are separate objects and have separate threads, so they are subject to dynamic configuration and modification, without relinking the whole application, which can be distributed across many remote hosts. Furthermore, this version of QuO supports more rapid prototyping and easier modification of Bottleneck, since

the QuO objects and threads are decoupled from the application objects and threads.

5.2 Case study 2: dynamic mission planning in an avionics platform

As part of a collaborative research effort, we have been using QuO as part of a dynamic mission planning avionics application. The application, illustrated in Figure 5, consists of a command and control (C2) aircraft and a fighter aircraft collaborating during flight to redirect the fighter's mission parameters. The C2 aircraft sends virtual target folders (VTFs), consisting of image data (as in case study 1), to the fighter aircraft, where they are processed to update the fighter's mission.

This has aspects of the WAN use-case, in that there is a (wireless) connection between the C2 and the fighter nodes, across which VTF image data is sent. This application uses QuO for in-band and out-of-band adaptation on the fighter side, as illustrated in Figure 6. During VTF image download QuO manages the tradeoffs of timeliness versus image quality. This is accomplished through image compression, image tiling, processor resource management, and network resource management.

When the fighter node requests an image from the C2 node, a QuO delegate breaks the image request into a sequence of smaller tile requests. The number of tiles that the delegate requests is based upon the image size while the compression level of an individual tile is based upon the deadline for receiving the full image and the expected download time for the tile. During image downloading a contract monitors the progress of receiving the tiles and influences the compression level of subsequent tiles based upon whether the image is behind schedule, on schedule, or ahead of schedule. The image is tiled from the point of interest first, with the early tiles containing the most important target data, so that decreased content of the later
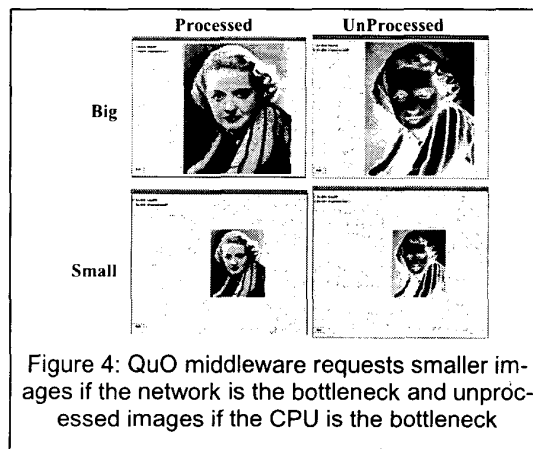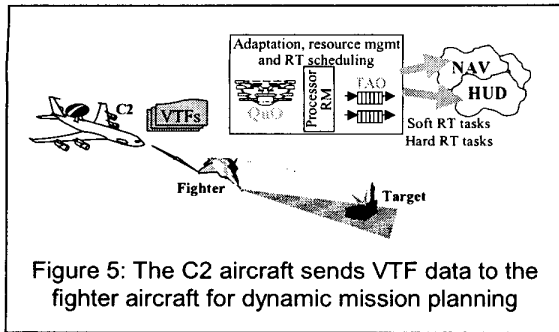

Figure 4: QuO middleware requests smaller images if the network is the bottleneck and unprocessed images if the CPU is the bottleneck

Figure 5: The C2 aircraft sends VTF data to the fighter aircraft for dynamic mission planning

tiles will have minimal impact on the dynamic planning capabilities.

In addition to the in-band adaptation of tiling and compression, QuO provides out-of-band adaptation in conjunction with the processor resource manager and dynamic scheduler components of the system. The processor resource manager selects task event rates from the ranges available for different tasks to optimally utilize the CPU. The contract monitors the progress of the image download through system condition objects interfacing to the network and CPU monitors. If the processing of the image tiles falls behind schedule, the contract prompts the processor resource manager to attempt to adjust the rates to allocate more CPU cycles to the decompression routine. This is in addition to, and orthogonal to, the in-band adaptation to adjust the compression level of the next tile.

If these adaptation attempts are not successful the QuO middleware triggers application adaptation. The application adjusts its timeliness or image quality requirements, by requesting longer deadlines or lower image resolution to reduce the urgency or amount of processing needed. Figure 7 illustrates the regions of the contract and the available adaptation options when the contract indicates that image receipt is *early* or *late*.

This application uses the passive C++ version of the QuO middleware for the reasons described in Section 4.2, *i.e.*, the avionics software uses a fixed number of threads, has no JVM, and demands minimal overhead.

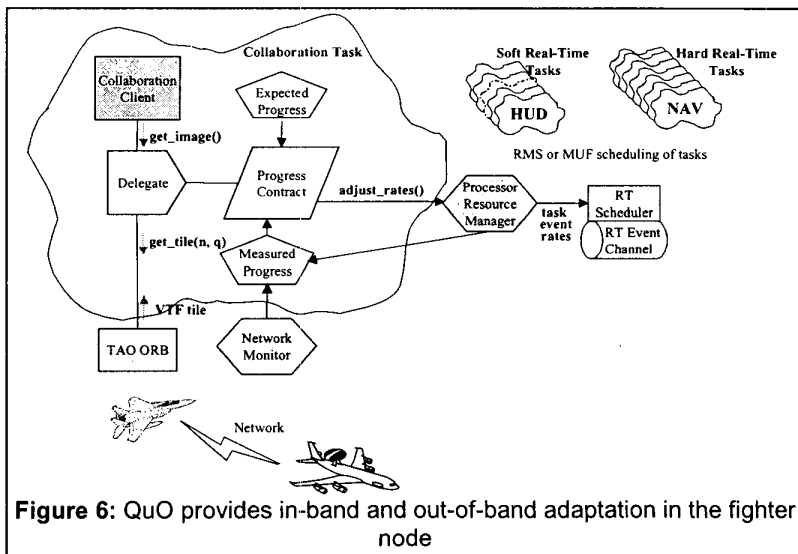## 5.3 Case study 3: shipboard dissemination of UAV video

As part of an activity for the US Navy, we have been developing a demonstration application utilizing QuO to control the dissemination of Unmanned Air Vehicle (UAV) data throughout a ship. Figure 8 illustrates
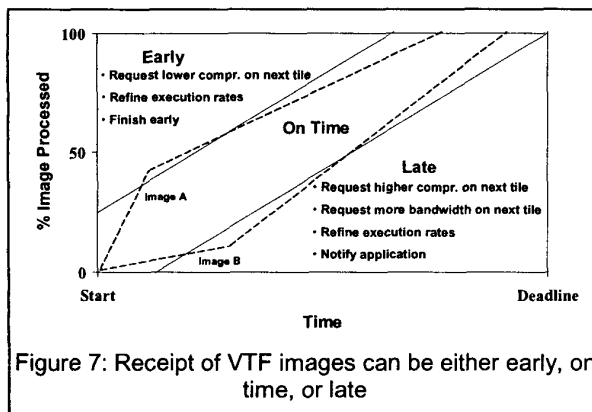
the initial architecture of the demonstration. It is a three-stage pipeline, with an off-board UAV sending MPEG video to an on-board video distribution process. The off-board UAV is simulated in early prototypes by a process that continually reads an MPEG file and sends it to the distribution process. The video distribution process sends the video frames to video display processes throughout the ship, each with their own mission requirements.

QuO adaptation is used as part of an overall system concept to provide *load-invariant performance*. The video displays 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). To accomplish this, system condition objects monitor the frame rate and the host load on the video display hosts. As the frame rate declines and/or the host load exceeds a threshold, they cause region transitions, which trigger the following adaptation:

- The video distribution process is told to drop frames going to the display on the overloaded host.
- The video display on the overloaded host is told to reduce its display frame rate to the rate at which frames are being sent it.

Simultaneously, system condition objects on the video distribution host are monitoring the host load, the input and output queues, and the frame rate. If the queues fill up or if the host load exceeds a threshold, the contract tells the video distribution process to drop frames to compensate. In this way, the adaptation attempts to maintain the video display processes displaying the current images that

Figure 6: QuO provides in-band and out-of-band adaptation in the fighter node

Figure 7: Receipt of VTF images can be either early, on time, or late

the UAV is observing with appropriate fidelity, regardless of the load on the various hosts.

The contracts on each host are simultaneously reporting the current contract region and video distribution metrics (e.g., queue lengths, frame rate, and number of dropped frames) to a system resource manager (RM). When QuO recognizes that the load on the video distribution host has become unacceptable, it notifies the RM. The RM then has the option of starting a video distribution process on another, less loaded host, and hooking it up to the UAV video source process and the video display processes. It then kills the processes on the overloaded host.

This application uses the passive C++ version of QuO and exhibits only out-of-band adaptation. The application maintains a data path, across which video frames are sent, and a separate control path, using CORBA IIOP, across which QuO adaptive control is sent. This seemed a better approach than putting a delegate in the path of the video frames because adaptation decisions do not need to be made between each frame and doing so could lead to hysteresis, rather than the desired load invariant performance.

6. Issues

Development of the variants of QuO middleware described in this paper raises a number of issues. We discuss a few of these here.

Choice of QuO implementation. For ease of prototyping, the original version of QuO is usually still the best choice. It has been used in many example applications, with different languages and different platforms. However, for applications that require a smaller footprint and fewer threads, either the passive Java or C++ version is necessary. Ultimately, our goal is a single design for QuO with an implementation in each language, each of which is configured differently for different use cases.

Unification of implementations. The original effort to develop the passive C++ version of QuO began as a porting effort from the original prototype. However, as we improved on the design to take advantage of C++ features, to take advantage of the portable ACE interface, and to improve the performance and footprint, we began to fold some of these improvements back into the original prototype. We are currently working on unifying the designs as much as possible, with the best features of each variant.

Maintenance and extending QuO. As long as there are different variants of QuO, the burden of maintaining, testing, and extending QuO will be increased. However, as we unify the variants around a single design, this burden should be reduced. We are also currently developing an extensive regression test suite to help support and ease the maintenance burden.

Combination and dynamic configuration of QuO. In theory, applications or components built around one variant of QuO or another should interoperate seamlessly. However, we have yet to develop any use cases or examples to test this. Other ideas to consider are whether the variants described in this paper are separate implementa-

Figure 8: Architecture of the UAV demonstration
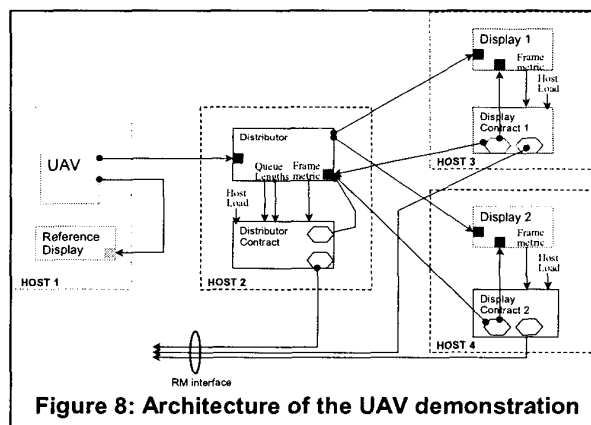
tions, whether features of them can be mixed and matched in a QuO configuration, and whether it makes sense to dynamically change from one variant to another.

7. Conclusions

During the course of transitioning our QuO adaptive middleware to real-world environments, we have had to extend and enhance it to provide new features, implementations, and services to support the specific characteristics of the target environments. These new implementations and features complement the development of other middleware supporting similar environments, such

as a real-time event service [10] and a hybrid static/dynamic scheduling service [9] for a Real-Time CORBA[16] ORB.

This paper has compared and contrasted the characteristics of the wide-area and embedded applications that have led to the emergence of different variants of the QuO middleware. In addition, we described three specific example applications that use different aspects of the QuO implementations to address their respective requirements and use-cases.

References

[1]   BBN Distributed Systems Research Group, DIRM project team. "DIRM Technical Overview," Internet URL http://www.dist-systems.bbn.com/projects/DIRM, 1998.

[2]   Bollella G., Gosling J., Brosgol B., Dibble P., Furr S., Hardin D. Turnbull M. *The Real-Time Specification for Java™*. Addison Wesley Longman, 2000.

[3]   Bollella, G., Gosling, J. "The Real-Time Specification for Java," *Computer*, June 2000.

[4]   Chandra P., Fisher A., Kosak C., Ng T.S., Steenkiste P., Takahasi E., Zhang H. "Darwin: Resource Management for Value-Added Customizable Network Service," *Proceedings of the Sixth IEEE International Conference on Network Protocols (ICNP'98)*, Austin, TX, October 1998.

[5]   Cukier M., Ren J., Sabnis C., Henke D., Pistole J., Sanders W., Bakken D., Berman M., Karr D., Schantz R. "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects," *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October, 1998.

[6]   Doerr B., Venturella T., Jha R., Gill C., Schmidt D. "Adaptive Scheduling for Real-time, Embedded Information Systems," *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, St. Louis, Missouri, October 1999.

[7]   Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8]   Gill C., Levine D. "Quality of Service Management for Real-Time Embedded Information Systems", 19th IEEE/ AIAA Digital Avionics System Conference (DASC), Philadelphia, Pennsylvania, 7-13 October 2000.

[9]   Gill C., Levine D., Schmidt, D. "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, special issue on Real-Time Middleware, Kluwer Academic Publishers, Vol. 20 No. 2, to appear 2001.

[10]   Harrison T., Levine D., Schmidt D. "The Design and Performance of a Real-time CORBA Event Service," *Proceedings of OOPSLA '97*, October 1997.

[11]   Kiczales G. "Beyond the Black Box: Open Implementation," *IEEE Software*, January 1996.

[12]   Loyall J., Bakken D., Schantz R., Zinky J., Karr D., Vanegas R., Anderson K. "QoS Aspect Languages and Their Runtime Integration," *Lecture Notes in Computer Science*, 1511, Springer-Verlag. *Proceedings of the Fourth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, 28-30 May 1998.

[13]   Loyall J., Schantz R., Zinky J., Bakken D. "Specifying and Measuring Quality of Service in Distributed Object Systems", *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*, 1998.

[14]   Object Management Group, "The Common Object Request Broker: Architecture and Specification Revision 2.4, OMG Technical Document formal/00-11-07", October 2000.

[15]   Object Management Group. "Minimum CORBA," OMG Document formal/00-10-59, October 2000.

[16]   Object Management Group. "Real-Time CORBA," OMG Document formal/00-10-60, October 2000.

[17]   Pal P., Loyall J., Schantz R., Zinky J., Shapiro R., Megquier J. "Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration", *Proceedings of The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 00)*, March 2000.

[18]   Schantz R., Zinky J., Karr D., Bakken D., Megquier J., Loyall J. "An Object-level Gateway Supporting Integrated-Property Quality of Service", *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

[19]   Schmidt D. "ACE: an Object-Oriented Framework for Developing Distributed Applications," *Proceedings of the 6th USENIX C++ Technical Conference*, April 1994.

[20]   Schmidt D., Levine D., Mungee S. "The Design and Performance of the TAO Real-Time Object Request Broker", *Computer Communications Special Issue on Building Quality of Service into Distributed Systems*, 21(4), pp. 294—324, 1998.

[21]   Vanegas R., Zinky J., Loyall J., Karr D., Schantz R., Bakken D. "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.

[22]   Wollrath A., Riggs R., Waldo J. "A Distributed Object Model for the Java System", *USENIX Computing Systems*, 9(4), 1996.

[23]   Zhang L., et al. "RSVP: a New Resource Protocol," *IEEE Network*, 7(6), September 1993.

[24]   Zinky J., Bakken D., Schantz R. "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems* 3(1), 1997.