Emerging Patterns in Adaptive, Distributed Real-Time, Embedded Middleware

Joseph P. Loyall, Paul Rubel, Richard Schantz, Michael Atighetchi, John Zinky
BBN Technologies

Abstract

Distributed real-time embedded (DRE) applications often have strict quality of service (QoS) requirements and are frequently deployed in environments in which resources are severely constrained, hostile conditions are prevalent, and resource contention is dynamic and unpredictable. For DRE applications to reliably operate in these environments, they must be able to measure the conditions of the system and adapt to recover from undesirable situations and to best utilize available resources. This paper presents two patterns that describe solutions appropriate to the problems of QoS adaptive applications: a QoS contract *pattern for managing adaptive decisions and tradeoffs and a* snapshot *pattern for grabbing a useful approximation of the current state of a system.*

QoS Contract

The QoS Contract pattern decouples quality of service (QoS) measurement, adaptation, and management from a functional application. It provides an architectural construct for representing the QoS needed and available in a system.

Example

Distributed Object Computing (DOC) middleware, depicted in Figure 1 for a CORBA-based implementation, has emerged and gained acceptance as the preferred paradigm for the development and implementation of a wide variety of applications as well as within a wide variety of environments. However, traditional DOC middleware has not supported the QoS needs of DRE applications well, because of its feature of hiding platform, transport, and operating environment specific details behind functional interfaces.

DOC Middleware that exposes QoS interfaces in addition to functional interfaces is starting to emerge. For example, the Quality Objects (QuO) [9] middleware supports adaptive quality-of-service (QoS) specification, measurement, and control. QuO supports distributed applications

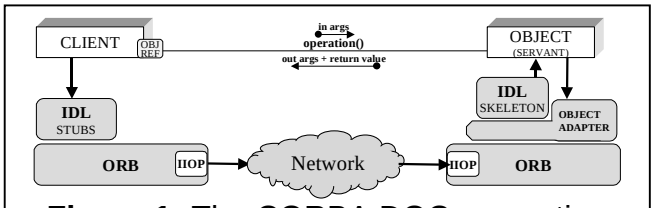


Figure 1: The CORBA DOC computing model

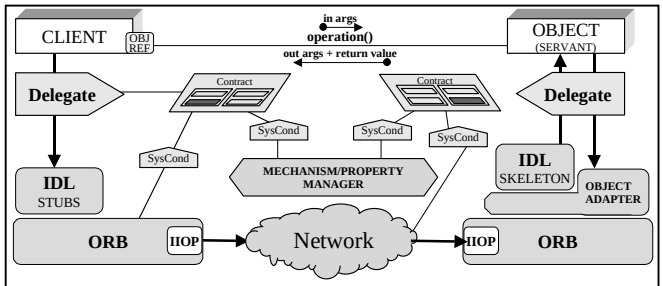
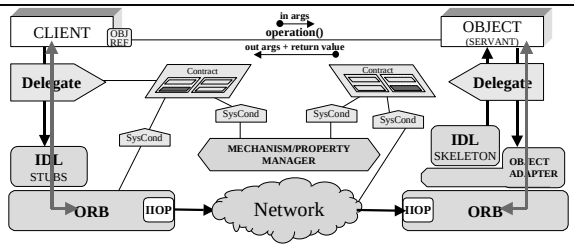


Figure 2: QuO adds components to control, measure, and adapt to QoS aspects of an application to the traditional DOC model

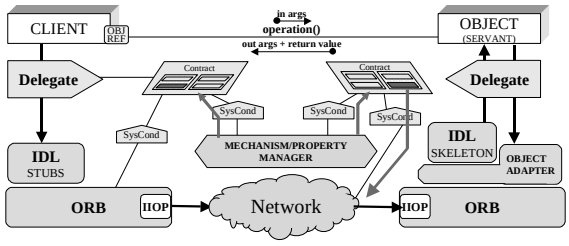
that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that may occur at run-time. By providing these features, QuO opens up distributed object implementations to control an application's functional aspects and implementation strategies that are encapsulated within its functional interfaces.

The QuO framework adds the following components to the traditional DOC model, as shown in Figure 2:

- *Contracts* which specify the level of service desired by a client, the level of service an object expects to provide, operating *regions* or *states* indicating possible measured QoS, and actions to take when the level of QoS changes.
- *Delegates* which 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 in the path of the remote method call based upon the current state of QoS in the system, as measured by the contract.
- *System condition objects* which provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO contracts.



(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

QuO contracts and delegates support two means for triggering manager-level, middleware-level, and application-level adaptation as illustrated in Figure 3. The QuO delegate supports *in-band* adaptation 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. QuO contracts support *out-of-band* adaptation by monitoring and controlling attributes of the system, such as system interfaces, resources, mechanisms, or managers, asynchronous to the application method calls.

Context. A DRE system in which resource availability is dynamically changing and in which at times there may not be enough resources to completely accommodate the desires of all activities. Competing QoS requests need to be mediated so that the use of resources by the system as a whole provides maximum value and is optimized. Applications must be able to operate with, or at least deal with, the level of service that the system can provide at any given time, even if it is less than they desire. The system must also be able to incorporate new applications and ways of moderating interactions between the new applications and the system already in place.

Problem. DRE applications have competing QoS goals. These goals are often incompatible with one another and need to be mediated so the system as a whole is productive. There is a recurring need in DRE contexts for customizing general application behavior for a particular target environment or con-

figuration. This includes runtime adaptation at the local level guided by knowledge of the wider system. This enables finer granularity tasks and applications to gracefully degrade, to relinquish resources that aren't needed, and to request additional resources when they are needed. This enables DRE applications that can continue to run effectively under a variety of operating conditions, are more robust in the face of outages and failures, are more dynamic (reducing the need for over-provisioning of resources), and do not have a single point of failure (the manager).

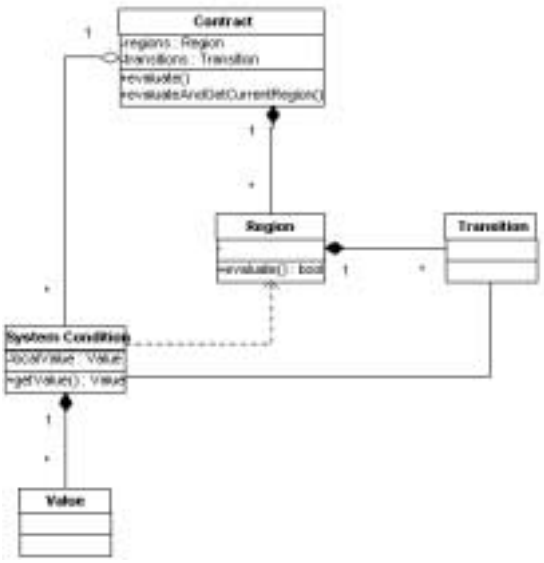


Figure 4: Object diagram for the QoS Contract Pattern

This includes the following forces:

- It is difficult to model the complete state of a complex system, under all possible operating conditions, and with respect to all possible parameters – computational, physical, and logical; and internal and external. Therefore, decisions must be able to consider conditions of interest, while abstracting away, approximating, or ignoring other conditions.
- The conditions of interest and the system states that they define might be related or orthogonal. Therefore the decision must be able to consider combinations of states, such as hierarchies or compositions.
- Adding information directly into each application about the whole system and the reactions to take increases tight coupling and makes the system as a whole more difficult to extend.
- Functional behavior and applications need to be configured and customized to use them in different environments.
- Implementing monitoring and response directly within each application tends to be at a lower level and therefore harder to re-use.

Solution. Apply the middleware-based *QoS contract* pattern to create a distributed decision, resource management, and control engine. This splits the decision making from the actions that implement the decisions. The contract takes care of receiving inputs, deciding what trade-offs should be made, and triggering adaptive behavior when appropriate.

Contracts should be able to be distributed, have many instances associated with varying granularities of objects, components, or clusters of objects and components; should be able to make decisions based on local or global information; and should be able to cooperate with other contracts in making distributed decisions.

Structure. Figure 4 illustrates a UML diagram of the QoS contract pattern. The contract describes a set of operating *regions* (or states), a set of transitions between regions, and predicates over system conditions (see the snapshot pattern next) that define the regions or transitions. Predicates can be associated with regions, defining the conditions under which a contract is in particular regions, or with transitions, defining the conditions under which a contract should transition between states.

Dynamics. The evaluation of a QoS contract to measure and/or affect the state of a system includes the following dynamics:

1. Grab a snapshot of the relevant system state (see the *Snapshot* pattern). This is important so that evaluation of all predicates is based upon the same values of system information.
2. If predicates are associated with regions, determine the current region (state) of the contract by evaluating the predicates to determine which are true (possibly starting with the last true one based upon the contract implementation).
3. If predicates are associated with transitions, start at the current state and evaluate the predicates to determine which transitions to follow.
4. If there was a transition, i.e., the current region or state is different from the last one, trigger any behavior associated with the transition.

Consequences.

- Separating actions from the process of deciding to use actions allows them to change independently.
- Separating contracts from the applications that they mediate and from the system conditions that they measure and control limit the extent to which applications need to be modified.
- Control may move from an application to the contract if a proactive contract is used, further separating application functionality from system QoS.
- Separating the collecting of data from actions to take promotes the creation of system conditions that are designed from the beginning to be reusable.
- Contracts can be supported via common middleware.

Related Patterns.

- Mediator [3]
- Strategy [3]
- Statecharts [8]

Snapshot

The Snapshot pattern provides a useful approximation of a consistent view of a system's state, with respect to the attributes of interest, at a given point in time. It is a pattern that facilitates runtime adaptive decisions.

Example. Making a decision affecting adaptive behavior in a system requires a reasonably accurate view of the system state at decision time with regard to relevant system QoS parameters and conditions. This often requires gathering and aggregating information from throughout the system, similar to the distributed snapshot problem documented in [1]. Examples of this type of information include the current state of resource availability and capacity or the current value of system or application parameters.

Context. Distributed systems, including both wide-area and embedded systems, have constituent parts that might be remote from one another, whose location might be unpredictable, and whose state might be available only through constrained resources. State information might similarly be widely distributed, available only through narrow pipes of resources, or available only at certain times.

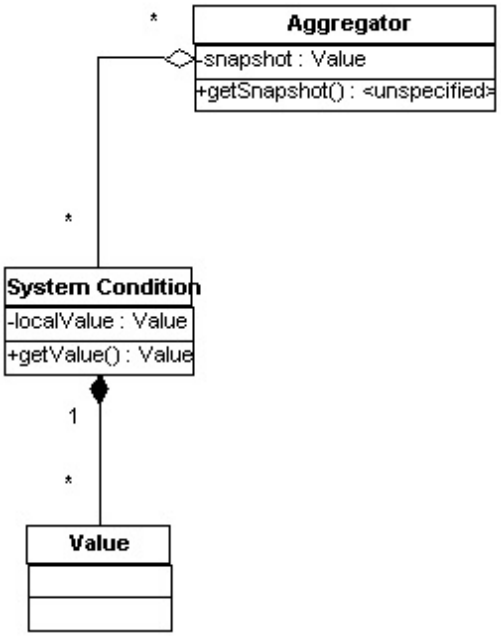


Figure 5: Object diagram for the Snapshot Pattern

Problem. An adaptive distributed system needs to have a reasonably accurate view of the relevant distributed system state available in a timely manner. This entails the following forces:

- The view of the system state should be available in a predictable, bounded amount of time once it is requested, in order for it to be useful in making a decision. This means that the call to assess the state of the system must return either immediately or with only minor, bounded calculation time.
- The snapshot should be a useful approximation of a consistent view of the system state, meaning that it should be composed of measurements gathered, calculated, or predicted no later than a small threshold of time ago and within a small threshold of time from one another.
- A *loose snapshot* of the system state may contain inconsistencies if it utilizes more than one measurement derived from a common system parameter and that parameter changes while the snapshot is being produced. A *transactional snapshot* prevents this by prohibiting system parameters from changing while a snapshot is being produced, but requires transactional control of system state and typically introduces additional delay.
- Multiple, distributed, consumers may be interested in the same data for different purposes and need not be aware of each other.
- Gathering and aggregating data should be disjoint from making it available.
- A snapshot of the system state will frequently include several values of system parameters with different properties. Some will change infrequently while others – such as a system clock – will change frequently. Some will be simple measures to collect, while others might take some processing or calculation to reach.

Solution. Apply the *snapshot* pattern to create a set of *system condition objects* and an *aggregator*. A system condition object is an object with a *getValue* interface through which its value can be accessed. This value can be any piece of system state that may be useful. The aggregator is responsible for gathering all these pieces of data and consolidating them into a view of the system.

Structure. The snapshot pattern, shown in Figure 5, is composed of two main classes, a collection of system condition objects representing state, and an aggregator which collects the values held by the system condition objects into a snapshot of the state of the system.

Dynamics. Grabbing a snapshot consists of the following dynamics:

1. Query each system condition object for its current value
2. Collecting the values and transforming them into an aggregate snapshot collection (e.g., a list or array).

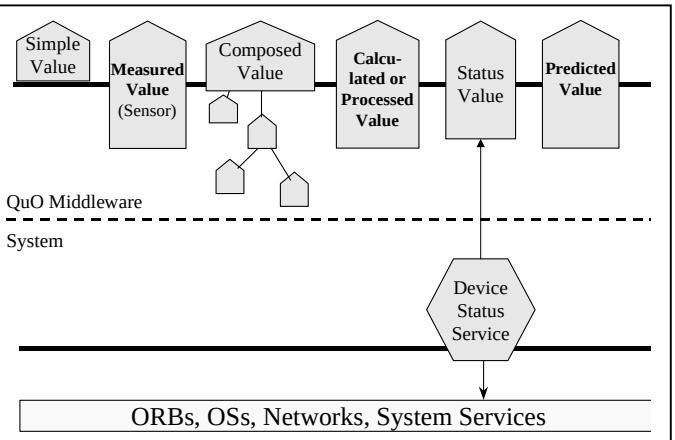


Figure 6: Different kinds of system condition objects

When *getValue* is called on a system condition, the system condition returns a value immediately or within a small, bounded measure of time. The implementation of the system condition object can be arbitrarily complicated, as long as a value is always ready to be returned and as long as replacing the system condition object's value is an atomic action. A system condition may cache values so that it is always ready with a response that can be returned in a timely manner. If loose snapshots are used the fact that some extra processing might yield a more up to date answer is tolerated in exchange for an expedient answer. A transactional snapshot may also support cached values but only if all other participants are also using the "stale" data to compute their values.

A design and implementation tradeoff must be made to decide whether it is more appropriate to place system condition objects on the same host as the aggregator or on the host of the system parameter they are measuring, when these are remotely distributed. The decision should consider which placement provides a more accurate and timely snapshot of system state.

When asked for state via the *getSnapshot* method, the aggregator queries the system condition(s) it is watching, in series or parallel, and returns the relevant system state.

Figure 6 illustrates examples of different kinds of system condition objects. From left to right they are:

- A simple value, something that just maintains a value set by a client or some other entity.
- A measured value, e.g., network throughput or a system's load average.
- A composed value, which may aggregate a value from a number of sources, which might also be system condition objects.
- A calculated or processed value
- A status value, maintained or collected elsewhere such as in the ORB, OS or other source.
- A predicted value based on past behavior.

Consequences.

- By separating the tasks of calculating system state from querying the value of the state new aggregators can be added easily and efficiently.
- The system condition interface allows new system conditions to be measured in a consistent way regardless of differences in their data format, availability, and other characteristics.
- State may be calculated needlessly, if no one is interested in it.
- The snapshot of system state is by necessity only accurate within a threshold of time.

- Transactional snapshots are often more costly than loose snapshots. They may require extra communication overhead that a loose snapshot does not need. However, a loose snapshot's view of the system may not always be consistent.
- The snapshot provides a visible focus on the conditions that affect system behavior.

Related Patterns.

- Observer [3]
- Reactor [7]

Known Uses

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, described in [2, 6]. The application, illustrated in Figure 7, 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, to the fighter aircraft, where they are processed to update the fighter's mission.

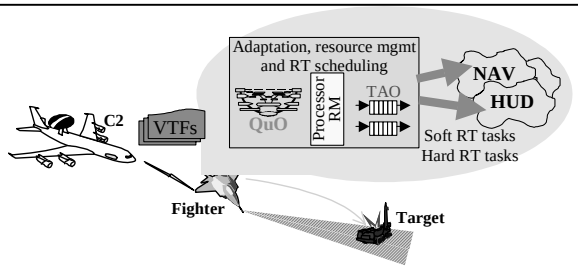


Figure 7: The C2 aircraft sends VTF data to the fighter aircraft for dynamic mission planning

Because of the constrained nature of the wireless link between the C2 and fighter nodes and because of the contention for resources with the other, more flight and mission critical, tasks on the aircraft, there are not always going to be enough resources to meet all the needs. The collaboration task needs to measure the resources available to it and adapt to effectively use the available resources for continuing to accomplish its goal under varying conditions.

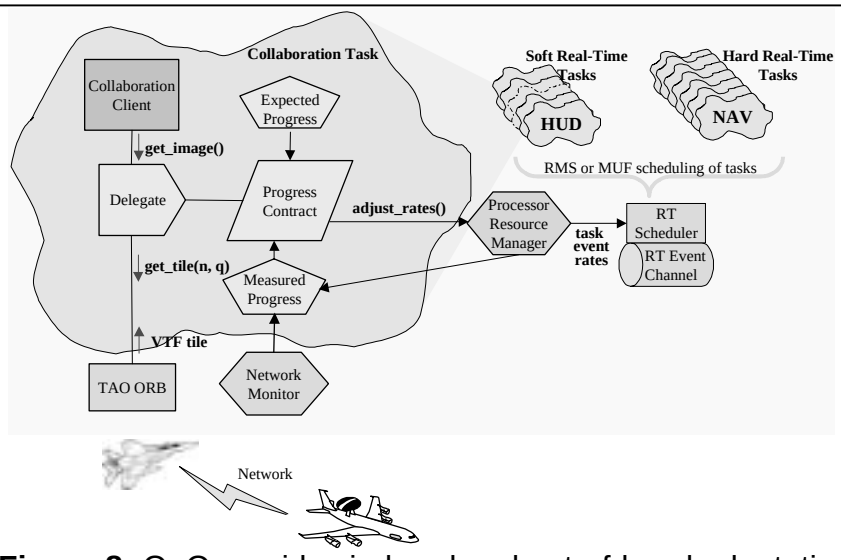


Figure 8: QuO provides in-band and out-of-band adaptation in the fighter node

This application uses QuO for in-band and out-of-band measurement and adaptation on the fighter side, as illustrated in Figure 8. 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 QuO contract, which is an implementation of the *QoS contract* pattern, 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. System condition objects are used to monitor the image download progress and to grab a *snapshot* of the state of the image download.

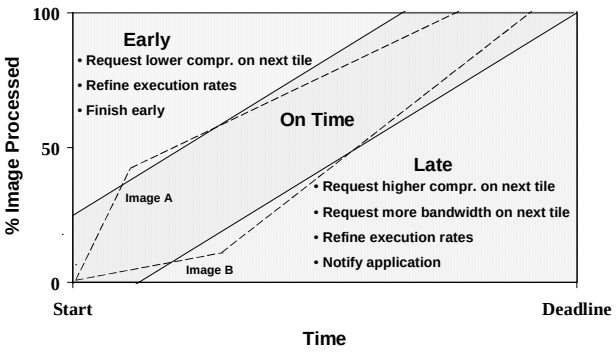


Figure 9: Receipt of VTF images can be either early, on time, or late

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 more effectively utilize the CPU. The *QoS contract* monitors the progress of the image download by grabbing a *snapshot* 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 9 illustrates the regions of the *contract* and the available adaptation options when the *contract* indicates that image receipt is *early* or *late*.

Video Dissemination in a Simulated UAV Context

Under DARPA's Program Composition for Embedded Systems (PCES) program and for the US Navy, we have been developing a simulated Unmanned Air Vehicle (UAV) system. The application, described in [4, 5] concentrates on the delivery of video imagery captured by a UAV to distributed control stations and the delivery of control signals back to the UAV. Figure 10 illustrates the architecture of the demonstration system. It is a three-stage pipeline, with multiple UAV sending video (in multiple formats, e.g., compressed MPEG and uncompressed PPM) to video distribution processes. The UAVs are simulated in our application by processes that continuously read video files and by live camera feeds, some of which are attached to robot vehicles. We have wireless and wired Ethernet connections to enable experimentation with a variety of network resource conditions. The target control stations and video displays will have different mission requirements: some require low latency video, others require high resolution, while another – serving an automated target recognition (ATR) process – requires important video content.

QuO adaptation is used as part of an overall system concept to provide *load-invariant performance*. The video displays of the distributed control stations must display images with acceptable timeliness and fidelity, regardless of the network and host load, in order for the UAV operators to achieve their missions (e.g., flying a UAV or tracking a target). To accomplish this, there are *QoS contracts* throughout the system, as illustrated in Figure 10, managing bandwidth (using RSVP, DiffServ, and application adaptation), throughput (managing video frame rate), the video source (managing filtering, scaling, and compression), and control signals (responding to ATR alerts). To support this, there are system condition objects that monitor the frame rate, host load, ATR status, network load, etc. and that the various contracts use to grab a *snapshot* of the relevant system state to drive the adaptations illustrated in Figure 11.

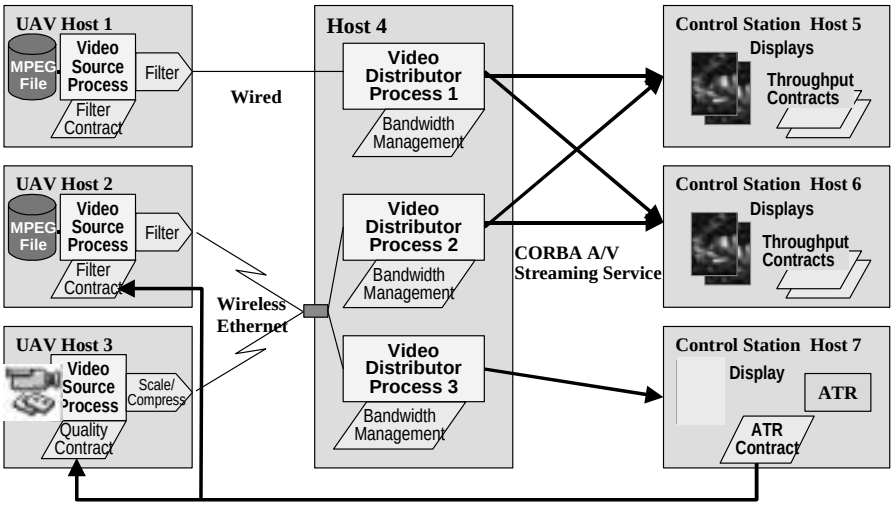


Figure 10: Architecture of the UAV Demonstration

See Also

In addition to the emerging patterns described above, the adaptive DRE applications that we are developing using QuO middleware use a number of instantiations of, or variations of, previously documented patterns. The QuO kernel is a *factory* [3] object that instantiates contracts, system condition objects, and callback objects.

System condition objects come in two flavors, *observed* and *non-observed*. Changes in the values measured by observed system conditions trigger contract evaluation, possibly resulting in region transitions and engaging out-of-band adaptive behavior. 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 whenever the contract is next evaluated. Observed system condition objects are implemented using the *observer* pattern [3] in contracts, although the observer does more than just update val-

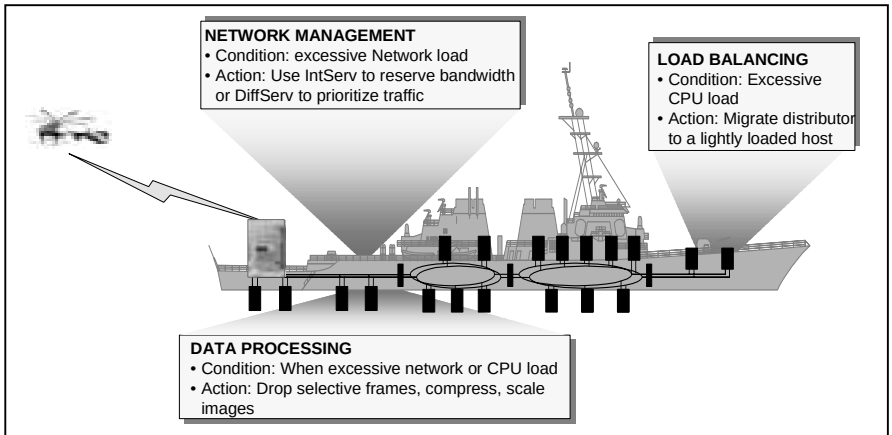


Figure 11: QuO provides adaptation to maintain timeliness of video dissemination in shipboard delivery of UAV imagery

ues in the contract. It also triggers evaluation of the contract. This could have alternatively been implemented using the *reactor* pattern [7]. For DRE systems like the avionics system in Figure 6, in which the number of threads and processes are strictly controlled to maintain the real-time constraint behavior, observed system condition objects can be used to evaluate the contract in the thread of the other tasks.

The QuO delegate is similar in many respects and may be an instance of the *Proxy* pattern [3]. It provides the same interface as the remote object stub, or the local class that it is representing, but it adds adaptive behavior and callouts to external QuO contracts and system condition objects. The QuO delegate supports in-band adaptation 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 appropriate 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.

System condition objects are instantiations of the *wrapper façade* pattern [7]. They provide a consistent set of object based interfaces to lower level mechanisms, managers, and resources, which may or may not have object interfaces themselves. These interfaces are suitable for use in QuO contracts, delegates, applications, other system condition objects, and external interfaces. They support the introduction of additional functionality, higher level views of low level information, and data fusion, smoothing, etc.

References

[1] Chandy K., Lamport L. "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, February 1985.

[2] Corman D, Gossett J. "Weapon Systems Open Architecture – Using Emerging Open System Architecture Standards to Enable Innovative Techniques for Time Critical Target Prosecution," 20th Digital Avionics Systems Conference (DASC), IEEE/AIAA, October 2001, Daytona Beach, Florida.

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

[4] Karr DA, Rodrigues C, Loyall JP, Schantz RE, Krishnamurthy Y, Pyarali I, Schmidt DC. "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," Proceedings of the International Symposium on Distributed Objects and Applications, September 18-20, 2001, Rome, Italy.

[5] Karr DA, Rodrigues C, Loyall JP, Schantz RE. "Controlling Quality-of-Service in a Distributed Video Application by an Adaptive Middleware Framework," Proceedings of ACM Multimedia 2001, September 30 - October 5, 2001, Ottawa, Ontario, Canada.

[6] Loyall JL, Gossett JM, Gill CD, Schantz RE, Zinky JA, Pal P, Shapiro R, Rodrigues C, Atighetchi M, Karr D. "Comparing and Contrasting Adaptive Middle-ware Support in Wide-Area and Embedded Distributed Object Applications". *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21)*, April 16-19, 2001, Phoenix, Arizona.

[7] Schmidt D., Stal M., Rohnert H., Buschmann F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[8] Yacoub S., Ammar H. "A Pattern Language of Statecharts", *Proceedings of the Fifth Annual Conference on the Pattern Languages of Programs, PLoP'98*, Allerton Park, Illinois, August 1998.

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