

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

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

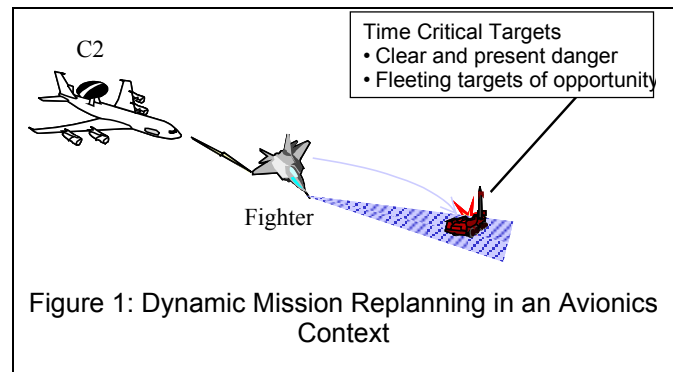
Abstract

This paper presents two patterns that describe solutions appropriate to the problems of quality of service (QoS) adaptive applications: a QoS contract pattern for managing adaptive decisions and tradeoffs and a snapshot pattern for obtaining a useful approximation of the current state of a system. These patterns appear in implementations of distributed real-time embedded (DRE) applications, which need to be QoS adaptive because of their strict QoS requirements and because they are deployed in environments with severe resource constraints, hostile conditions, and dynamic and unpredictable resource contention.

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. Consider a fighter aircraft with a preplanned mission and an associated command and control (C2) aircraft, which wants to update the fighter's mission parameters while the fighter is enroute to its target. Specifically, the C2 may want to direct the fighter to a new, more critical target or to alert the fighter to threats along the route to its target, as illustrated in Figure 1. However, this collaborative mission replanning from the C2 to the fighter must deal with resource constraints and contention, since the wireless network between the aircraft is limited in its bandwidth and the processors on the fighter are servicing other, possibly mission-critical, tasks. Statically preallocating sufficient resources to support the collaboration is difficult or impossible because of the dynamic nature of the mission replanning and the data involved. If the system relies upon resources being always available and predictable, it might not be able to exchange the important mission planning information in time to act upon it.



One way to deal with the issues of constrained and shared resources is to compress the information being sent, in this case images. When the bandwidth becomes low the images can be compressed, thereby saving network resources. This, however, will increase the amount of work required of the processor, impacting other jobs and forcing the system to decide which jobs will receive processing

time. Compressing images may also lower the quality of the information that the pilot relies upon to complete the mission.

Tiling the images, splitting a single image into a collection of smaller images that are stitched together when received, is one possible solution to gracefully degrade image quality. The important portions of the image are delivered with high quality while the less important regions are degraded. This makes better use of the available resources at the cost of complexity in the software.

Placing knowledge about tradeoffs and ways to resolve them in the application adds complexity to the application. Where before the programmers only needed to worry about sending and receiving an image, they now worry about not only the functional aspects of the system but also the QoS aspects. The application is much more complex due to the QoS aspects intertwined with the functionality, aspects concerned with the monitoring of bandwidth, and trading off compression versus data quality, CPU versus bandwidth, allocation of processing in the fighter, and tiling of images.

Context. An application that can operate (with different qualities of service) in a variety of levels of resource availability.

Problem. DRE applications have competing QoS goals and must operate with stringent resource constraints. Hard coding QoS awareness and control in an application leads to complex, non-portable, difficult to maintain applications and can lead to problems as a system evolves. Networked embedded applications compete for shared resources and this competition needs to be mediated so that the system as a whole can meet its requirements. When there are insufficient resources to fully provision for all applications and in dynamic environments, some applications are going to have to operate with less than an optimal amount of resources. In addition, there is a recurring need in DRE contexts for customizing general application behavior for a particular target environment or configuration. This customization includes runtime adaptation at the local level guided by knowledge of the wider system. Guided runtime adaptation 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. 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) than non-adaptive applications, and do not have a single point of failure (a central resource manager).

Solving this problem requires the resolution of 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 regions that these conditions 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 coupling and makes the system as a whole more difficult to extend.
- Functional behavior and applications both need to be configured and customized for use in different environments.
- Decisions should be able to be made based on local or global information, and should be able to cooperate with other applications in making distributed decisions.

Solution. Apply the *QoS contract* pattern to decouple QoS awareness and adaptation from the functionality of an application.

The QoS contract specifies the QoS *regions* of interest to an application, including the QoS levels with which the application can operate, whether with full or with degraded functionality. *QoS regions* are specified using *predicates* over measurements of current *system conditions*. *Transitions* specify behavior that is triggered as the contract moves into and out of regions. The movement of a contract between regions reflects changes in system conditions at runtime and how they affect the resources desired by the application and the resources with which the application can work.

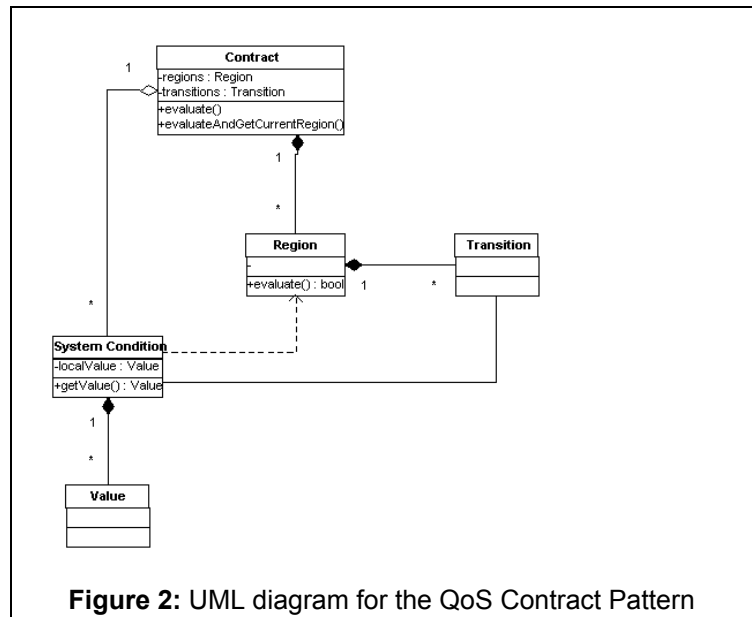


Figure 2: UML diagram for the QoS Contract Pattern

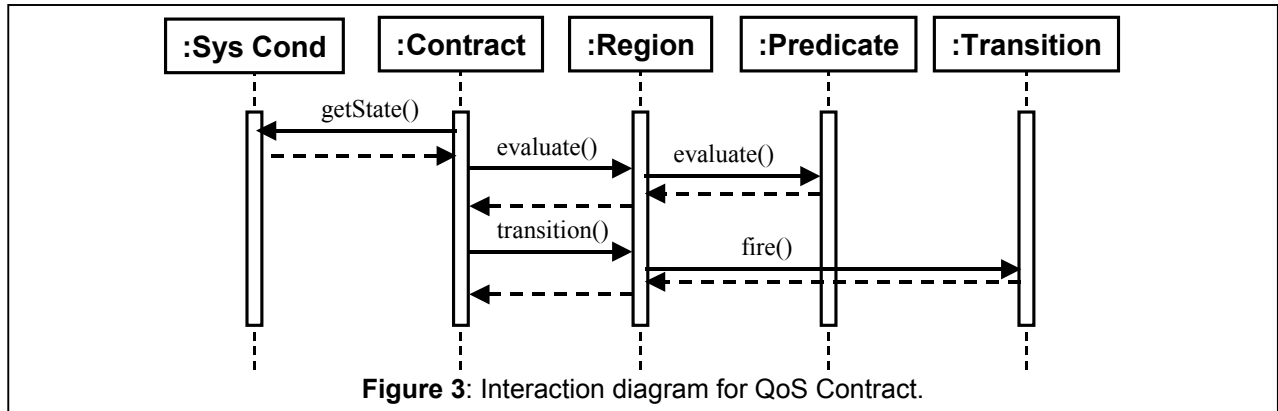
Structure. There are six key participants in the QoS Contract pattern, shown in Figure 2:

- A *contract* specifies 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.
- *System conditions* provide interfaces to resources and mechanisms in the system that need to be measured and controlled by contracts.
- A *predicate* is a Boolean function on a collection of system conditions. If the *system conditions* of the system satisfy the *predicate*, the *predicate* will evaluate to true.
- A *region* is specified by a *predicate* and contains *transitions*. When *system conditions* match those specified by the *region's predicate*, the *contract* is said to be in a given *region*.
- A *transition* describes what to do when a contract moves from one region to another. *Transitions* can be specified for either entering or leaving a *region* and are instances of the *Command* pattern [3].
- A *value* is a measurement of the system reported by a *system condition*.

Dynamics. The evaluation of a QoS contract to measure and/or affect the state of a system includes the following dynamics illustrated in Figure 3:

1. The system takes 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. Determine the current region of the *contract* by evaluating the *predicates* to determine which are true.
3. If the current *region* is different from the last *region* trigger any behavior associated with the *transition*.

Contracts can be used for *out-of-band* or for *in-band* adaptation, i.e., asynchronous or synchronous to the distributed object interactions. Out-of-band contracts are associated with a thread of control that



can execute asynchronously to the application’s method calls. The thread of control can either periodically trigger contract evaluation or contract evaluation can be linked to events associated with changes in system conditions. In-band contracts are evaluated at relevant points in the path of the application’s remote method calls, such as remote method calls and returns. Evaluation of the contract can be inserted into the method call path using instances of the *Proxy* [3] or *Interceptor* [7] patterns.

Implementation. Implementing the *QoS Contract* pattern consists of the following steps:

1. *Determine the QoS regions of concern.* Determine the ranges of QoS in which the application can operate, from the highest level of QoS leading to an optimal application operating environment, lower ranges in which the application can operate with some degraded functionality or tradeoffs, down to the lowest level below which the application cannot operate.
2. *Define the system conditions comprising or controlling the QoS.* System resources, managers, infrastructure interfaces, etc. will provide means for measuring and controlling the QoS in the system at a given time. Describe the interfaces to these that the contract needs to access to determine the QoS in the system.
3. *Create predicates.* Each region needs to be described by a predicate over the system conditions. Define the predicates over the system conditions associated with each region.
4. *Determine transitions and actions.* Determine what needs to be done as the system QoS changes during runtime to compensate, recover, or take advantage. Encode these actions as transition behavior.
5. *Integrate the contract into the application.* Determine whether the contract is to be used in-band or out-of-band. If the contract is to be used in-band, create a proxy or interceptor to insert contract evaluation into the proper places along the application’s method call paths. If the contract is to be used out-of-band, create a separate thread for it or hook it in the path of an existing asynchronous (to the application) thread, such as that of a resource manager or monitor being observed by the contract.

Example Resolved. The fighter and the C2 can use a *QoS Contract* to separate the QoS management from the application code, as illustrated in Figure 4. The contract can define system conditions that measure processor usage, network bandwidth, and the speed of data exchange between the fighter and C2 nodes. The contract can define regions dealing with slowdowns in the exchange of data caused by constraints in network bandwidth or limitations in processing the data on the source or target. As the contract transitions between regions, the applications can adapt by requesting more resources, by trading off network and CPU (e.g., using compression), or by adapting the data (e.g., scaling or tiling images). This adaptive behavior can be implemented by object calls in transitions, proxies or interceptors.

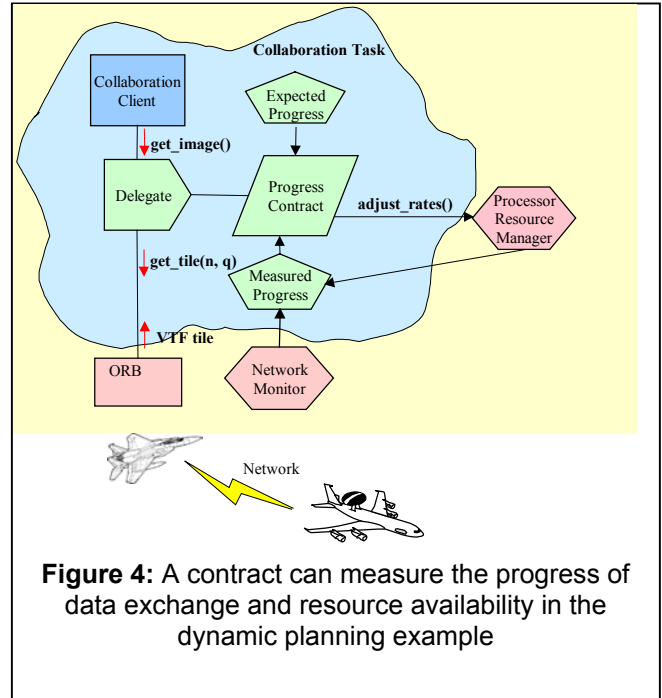


Figure 4: A contract can measure the progress of data exchange and resource availability in the dynamic planning example

Variants. States/Regions. One variant of the *QoS Contract* pattern involves using *states* instead of regions. If a contract is using regions, transitions may go from any region to any other region. States make the contract act more like a state machine[11]. From a given state, transitions are only allowed to go to other regions that have transitions from the current state.

When using states, predicates are associated with transitions instead of with regions. To compute the next state, start at the current state and evaluate the predicates associated with each transition to determine which transition to follow.

Observed/Non-Observed system conditions. System conditions 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 conditions 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.

Known Uses.

Dynamic Mission Planning In an Avionics Platform. Weapon System Open Architecture

(WSOA) [2,6] is a dynamic mission planning avionics application consisting of a command and control (C2) aircraft and a fighter aircraft collaborating during flight to update the fighter’s mission parameters, as illustrated in Figure 5. 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. Because of the constrained nature of the wireless link between the C2 and fighter nodes and

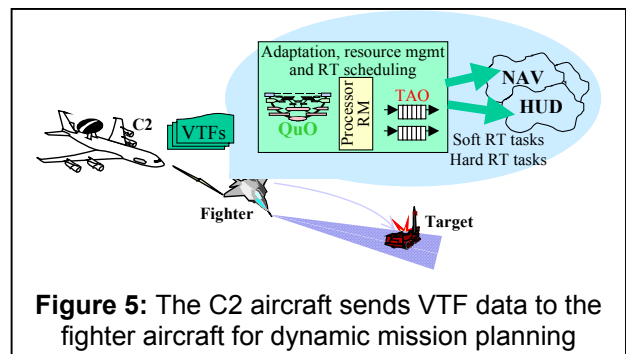


Figure 5: The C2 aircraft sends VTF data to the fighter aircraft for dynamic mission planning

because of the contention for resources with the other, more flight and mission critical, tasks on the fighter, 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.

This application uses an instance of the contract pattern for in-band and out-of-band measurement and adaptation on the fighter side.

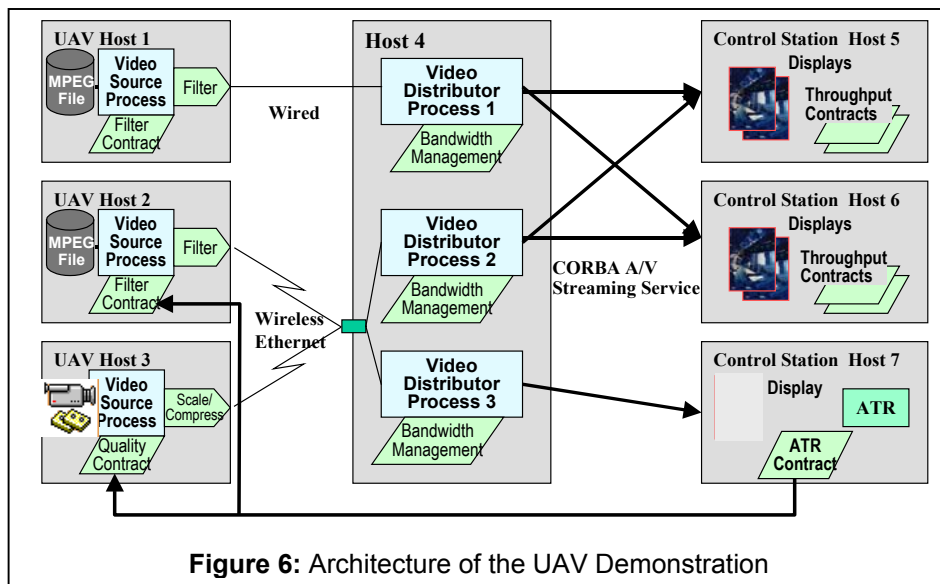
During VTF image download the contract manages the tradeoffs of timeliness versus image quality, by measuring download progress and adapting with image compression, image tiling, processor resource management, and network resource management.

Video Dissemination in a Simulated UAV Context. A simulated Uninhabited Aerial Vehicle (UAV) application developed for the US Navy and for DARPA’s Quorum and Program Composition for Embedded Systems (PCES) programs, 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 6 illustrates the architecture of the demonstration system. It is a three-stage pipeline, with multiple UAVs sending video (in multiple formats, e.g., compressed MPEG and uncompressed PPM) to video distribution processes. The UAVs are simulated by processes that continuously read video files and by live camera feeds, some of which are attached to robot vehicles. The application includes wireless and wired Ethernet connections to enable experimentation with a variety of network resource conditions. The target control stations and video displays 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.

A set of contracts throughout the distributed application is used to provide *load-invariant performance*. The contracts manage 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* (see the *Snapshot* pattern) of the relevant system state.

Consequences. The *QoS Contract* pattern offers the following **benefits**:

- *Insulates from change.* Separating actions from the process of deciding to use actions allows them to change independently.
- *Simplifies Retargeting Application.* The extent to which applications need to be modified is limited by separating contracts from the applications that they mediate and from the system conditions that they measure and control.



- *Increases reuse.* Separating the collecting of data from actions to take promotes the creation of system conditions that are designed from the beginning to be reusable.
- *Lessens Development.* Contracts can be supported via common middleware.

The *QoS Contract* also incurs the following **liabilities**:

- *Extra overhead.* The extra indirection introduced by the contract (as opposed to having QoS control encoded directly in the application) can introduce added overhead.
- *Static determination of QoS regions.* The QoS regions of concern must be predetermined, limiting the granularity of control and flexibility of the contract.

See Also.

- Quality Connector[10] The *Quality Connector* covers a similar area to the *QoS Contract*. Both deal with how to separate concerns of QoS from application concerns. The *Quality Connector* concerns itself with how to implement the interactions between applications and QoS components and assumes a way to manage changing the QoS. The *QoS Contract* on the other hand focuses on QoS specification and decision making while spending less time on the connection mechanism between components.

Snapshot

The Snapshot pattern provides a useful approximation of a consistent view of a system’s state at a given point in time. It is a pattern that facilitates runtime adaptive decisions.

Example. Consider a weather reporting system. The system is composed of a distributed collection of sensors and is responsible for creating a comprehensive picture of the system when requested. Sensors are placed in out of the way locations and have limited connectivity that can be affected intermittently by hostile conditions. Even in the best of conditions bandwidth to the sensors is limited.

When a user wants to know current conditions the system needs to contact each sensor individually and query its state. Unfortunately, due to low bandwidth and unreachable sensors generating a report on demand is too time consuming and information may be missing due to unreachable sensors.

Context. A distributed or embedded system that needs a view of the system state but whose ability to gather this information is constrained.

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.

- 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 provide indirection between the supplier of the information and the consumer. By splitting the gathering of information from the request for the information, the information can be gathered as resources allow and can be queried without interfering with the supplier.

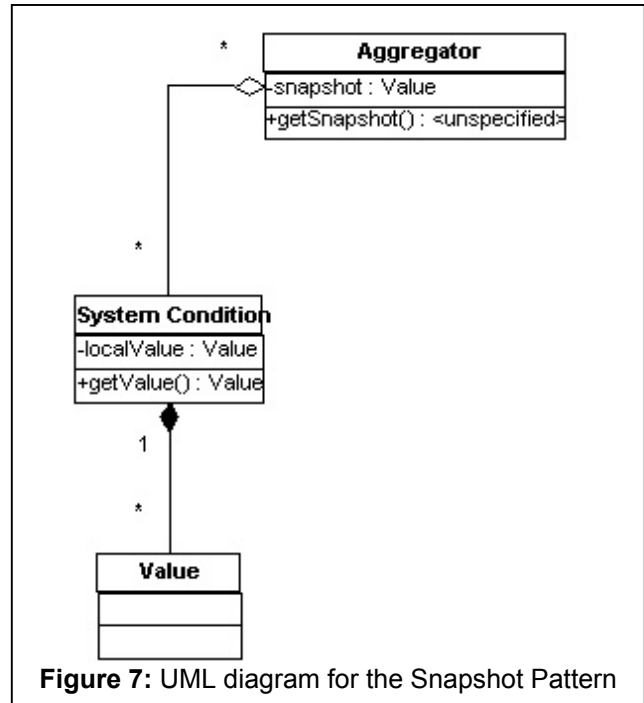


Figure 7: UML diagram for the Snapshot Pattern

In detail: create a set of *system conditions* and *aggregators*. A *system condition* is responsible for querying state and storing the result. This state 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. There are two main participants in the Snapshot pattern, shown in Figure 7.

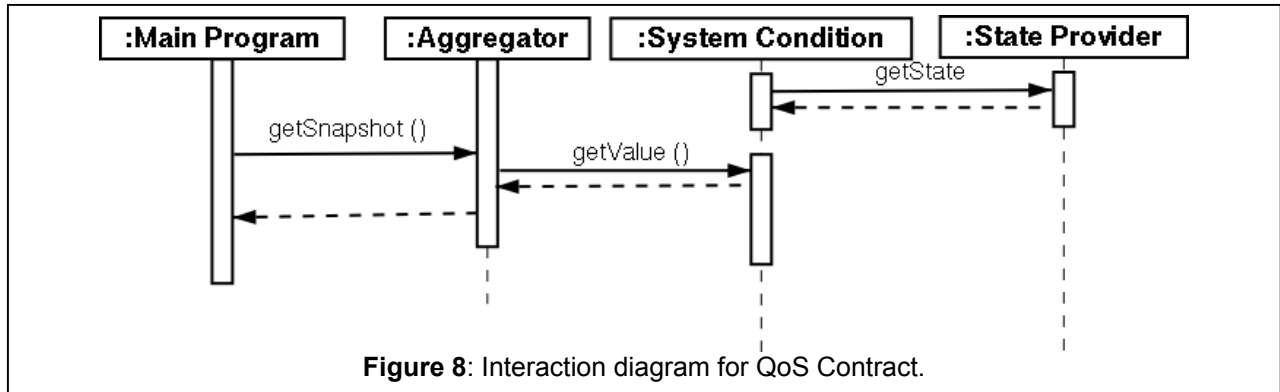
- *System Conditions* provide a Wrapper/Facade [7] for system components, called *Values*, and provide access to the state of these components through a `getValue` method.
- *Aggregators* collect and transform values held by, possibly distributed, *system conditions* into a snapshot of the state of the system. Aggregators may be system conditions themselves if they feed higher level aggregators.

Dynamics. Obtaining the state of the system consists of the following steps, illustrated in Figure 8:

- A system condition collects state from its affiliated value.
- An *aggregator* queries each *system condition* for its current value using `getValue`.
- The *aggregator* then collects the values and transforms them into an aggregate snapshot collection (e.g., a list or array) that can be returned via a call to `getSnapshot`.

Implementation: There are three main activities associated with implementing the Snapshot pattern.

- Implement *system conditions* that wrap sources of state and provide a `getValue` method. 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'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.

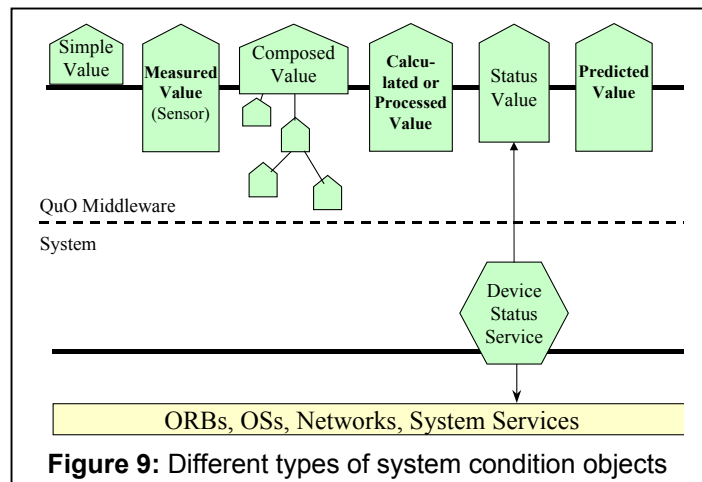
- Decide whether it is more appropriate to place system condition objects on the same host as the aggregator or on the host of the system parameter they are measuring, when these are remotely distributed. The decision should consider which placement provides a more accurate and timely snapshot of system state as well as what effect different placements have on the system.
- Implement an aggregator. When asked for state via a call to `getSnapshot`, the aggregator queries the system condition(s) it is watching, in series or parallel, and returns the relevant system state.

Figure 9 illustrates examples of different kinds of system conditions. 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.

Example Resolved. The weather system can make use of the *Snapshot* pattern. *System conditions* can be created near the display that will query the sensors for their values as resources allow. When a view of the current conditions is requested the display can query an *aggregator*, which will get the sensor values from the *system conditions* and return them in a timely manner to be displayed. Missing values can be replaced with the last known value or with a projected value.

Variants. *Consistent snapshots*[1]. Up to this point the pattern has discussed *loose snapshots*. When 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 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 *consistent snapshot* is used when this type of behavior is not acceptable. In a consistent snapshot all state to be aggregated needs to be consistent and come from the same underlying values. One way to get a consistent snapshot is to use transactions, where the system is stopped from doing processing while the state is gathered. This trades delay, while the system is stopped, for a consistent view of the system.

Push/Pull of System State. System conditions may either query the state they are watching, a pull model, or have the state provided to them by the component whose state they are watching, using a push model.

Known Uses.

Dynamic Mission Planning In an Avionics Platform. In the WSOA project described in the *QoS Contract* pattern, the QoS contract monitors the progress of the image download by obtaining a snapshot of system condition objects that measure image download progress and CPU resource allocation.

Video Dissemination in a Simulated UAV Context. In the simulated UAV system described in the QoS Contract pattern, 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 before determining what the state of system QoS is.

Staff Meetings. When staff members gather for a group meeting each member shares a high-level view of their current state. When the meeting is over the entire group knows the state of the group as a whole.

Consequences. The *snapshot* pattern has the following **benefits**:

- 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.
- Loose snapshots can be efficient to obtain.
- Consistent snapshots can accurately represent the state of the system.
- The snapshot provides a visible focus on the conditions that affect system behavior.

The *snapshot* can also incur the following **liabilities**:

- 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.
- Consistent snapshots are often more costly than loose snapshots. They may require extra communication overhead that a loose snapshot does not need.
- A loose snapshot's view of the system may not always be consistent.

See Also.

- The *Snapshot* pattern makes use of the Memento [3] pattern to encapsulate the state of the system.

- Aggregators can be seen as instances of the Observer [3] pattern.

Acknowledgements. The authors would like to thank our shepherd Joe Cross and the DRE focus groups from PLoP 2002 and OOPSLA 2002 for their help and insightful comments.

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.
- [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.
- [10] Cross, J., Schmidt, D. “Quality Connector – A Pattern Language for Provisioning and Managing Quality-Constrained Services in Distributed Real-time and Embedded Systems”, *Proceedings of the 9th Annual Conference on Pattern Languages of Programs, PloP 2002*, Allerton Park, Illinois, September 2002.
- [11] Bo I. Sandén. “The State-Machine Pattern”, *Proceedings of the conference on TRI-Ada '96: disciplined software development with Ada*, Philadelphia, Pennsylvania, 1996.