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

Joseph P. Loyall, Richard Schantz, Partha Pal, John Zinky, Michael Atighetchi
BBN Technologies

Introduction

We have been developing Quality Objects (QuO), an adaptive middleware framework that supports the development of distributed object applications that can measure, control, and adapt to quality of service (QoS) needs and variations in a system. We have applied QuO to the problems of distributed real-time embedded (DRE) applications, including a streaming video dissemination application and a dynamic replanning avionics application, both described in [3]. During the course of developing QuO and of applying it to DRE problems, we have both used many documented patterns and identified new patterns for network centric computing. Some of these are variations on existing, documented patterns, while others appear to be newly discovered. We have seen these patterns appear over and over in the development of our adaptive, QoS-aware systems, which suggests strongly that they are indeed new and reusable patterns. Time and additional investigation will determine whether these are also lasting and universally useful patterns.

Overview of Quality Objects (QuO) Middleware

In a traditional distributed object application, a client makes a logical method call to a remote object to initiate some distributed behavior. 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) based framework designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. 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.

A method call in the QuO framework is a superset of a traditional DOC call, and includes the following additional components, as shown in Figure 1:

- **Contracts** which 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** 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 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.

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 contracts, system condition objects, callback mechanisms, and object delegate behavior. To support the added role of QoS developer, we are developing a QuO toolkit, 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 [4, 5].

• 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 [7].

• 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 [4].

Programming Distributed Real-Time Embedded Systems using QuO

To experiment with and evaluate this technology, we have used QuO in the construction of two distributed real-time embedded applications. The first, illustrated in Figure 2, is part of a collaborative research effort with the Boeing Corporation. The application consists of a command and control (C2) aircraft and a fighter aircraft collaborating during flight to replan 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. QuO is used to adapt the soft real-time collaboration task to the dynamic constraints of the embedded system. It breaks a request for a VTF image into tiles, monitors the progress of the tile acquisition, and changes the quality level of subsequent tiles to compensate for late or early image download. QuO also interacts with the processor resource manager to request additional cycles if the image is late and there are cycles to spare and to reduce its resource demands if cycles are needed by other, more critical, avionics tasks.

The second embedded application is a simulation of shipboard dissemination of video provided by an unmanned aerial vehicle, as illustrated in Figure 3. In this application, QuO is used to maintain mission timeliness requirements by adapting for conditions, such as excess network or CPU contention, that can cause the video to fall behind. Current adaptations controlled by QuO in the UAV software are to reserve network bandwidth, drop frames (reducing video content while maintaining timeliness), and migrating processes to more lightly loaded hosts.
Both of these applications are described in more detail in [3].

**Patterns Used in the QuO Infrastructure**

The QuO middleware software uses a number of previously documented patterns in its implementation of middleware support for QoS and adaptation. The QuO kernel is a *factory* object that instantiates contracts, system condition objects, and callback objects [2].

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. Observed system condition objects are most suitable for measuring conditions that either change relatively infrequently or for which a measured change can indicate an event of more immediate notice to the application or system middleware. 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 next evaluated due to a method call or return or due to an event from an observed system condition object. Observed system condition objects are implemented using the *observer* pattern in contracts, although the observer does more than just update values in the contract, it triggers evaluation of the contract. This could also have been implemented using the *reactor* pattern. For DRE systems like the avionics system in Figure 2, in which the number of threads and processes are strictly controlled to maintain the real-time constraint behavior, it isn’t feasible to have the QuO kernel’s contract evaluator as a separate process, as it is in many of our non-embedded QuO applications. Observed system condition objects interfacing to other tasks, such as the process manager in the avionics DRE system, 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. 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. 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.

**Emerging Patterns in QuO DRE Applications**

While the development of the QuO middleware has used previously described patterns, its development and its application to DRE applications have produced a number of newly recurring potential patterns, which we describe in the following paragraphs.

**The snapshot pattern.** Evaluating a contract to determine the state of the QoS in a system or subsystem and to make a decision affecting adaptive behavior often requires the gathering of state information from throughout the system. Examples of this type of information include the current state of resource availability and capacity or the current value of system or application parameters. This newly emerging snapshot pattern consists of accessing a number of discrete values representing aspects of a system state, within a controlled threshold of time, in order to get a *snapshot* of system state, similar to the distributed snapshot problem documented in [1]. It is more important that the snapshot of the system be available in a predictable, analyzable amount of time than it is for it to be completely accurate (which in general it can never be anyway), so the value of any component of the snapshot (implemented in QuO by system condition objects) must be available either immediately or with only minor, bounded calculation time.

**The contract pattern.** Our work in QuO and its application to DRE systems has identified a new recurring pattern for a decision engine that mediates the competing needs of applications and the tradeoffs between the qualities of service desired by an application and the level of service that the system can provide at any given time. While there has been significant work in system level managers, both property managers and resource managers, to control access to resources and the properties provided to applications, there is a recurring need in DRE applications for adaptation at the local level. 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). In QuO, the distributed decision, management, and control engine is provided by the QuO contracts, which can be distributed, have many instances associated with objects, components, or clusters of objects and components, and can make decisions based on local or global information and cooperate with other contracts in making distributed decisions.

The connection pattern. The Pattern-Oriented Software Architecture book [6] describes a number of patterns centered around the connection of objects, including the Acceptor-Connector, Synchronous Connector, Asynchronous Connector, and the Publisher/Subscriber patterns. We are starting to explore the existence of a higher-level notion of a connection, that can be described as a collaboration between objects in a system. Many types of connections, such as client-server, publisher-subscriber, event channel, sensor-actuator, and sender-receiver, have distinct similarities centered on the notions of roles and connections. The roles define the objects or components that provide the data (the source) and the objects or components that receive the data (the sinks). The connection describes how the data gets from a set of sources to the set of sinks. Specific instantiations of the connections capture the specifics of whether sources and sinks are anonymous (as in an event channel) or not (publisher-subscribe); whether the connection is a push model or pull model; and so on.

Conclusions

We have developed the QuO adaptive middleware framework and have begun applying it to the development of DRE applications in a number of contexts. This has led to the refinement and re-implementation of QuO middleware components suitable for DRE applications, using a number of documented patterns. In addition, we have identified a number of newly emerging patterns associated with adaptive QoS management, both in QuO and related middleware, and in the DRE applications which incorporate adaptive QoS management techniques.

References