

Detection and Reaction to Unplanned Operational Events in Large Scale Distributed Real-Time Embedded Systems¹

Jianming Ye, Joe Loyall, Rick Schantz, Gary Duzan
BBN Technologies, Cambridge, MA
{jye, jloyall, schantz, gduzan}@bbn.com

1. Problem overview

In recent years, we have seen an increase in the use of embedded systems in large distributed systems, often designed and optimized for critical real-time controls. These systems of systems often include multiple end-to-end application streams and share heterogeneous, constrained resources. Unlike most non-distributed, static, closed embedded systems, these systems operate in highly dynamic environments and need to be able to react in real-time to changes in missions, conditions and operating environments. Today more and more of these mission-critical distributed real-time embedded (DRE) systems are becoming Quality of Service (QoS) enabled, thus providing predictable, adaptable, and managed behaviors. Due to the dynamic and distributed nature of DRE systems, unplanned events can occur that could significantly impact the behavior and performance of these systems. Examples of unplanned events are significantly less than expected available resources, system failures, unexpected service demands, and events completely outside the system scope which have unanticipated side effects on its operation.

With potentially large numbers of unplanned events in DRE systems, designing adaptation strategies to handle these is challenging. In order to effectively manage unplanned events in DRE systems, we first need to clearly define what unplanned events are and how they differ from planned events. A planned event often has known signatures that we can probe for or monitor and is expected to happen during the operation of the system, even if it is “abnormal” behavior. When it happens, we know exactly what to do and we often know the cause(s) of the event. Unplanned events, on

the other hand, are not expected to happen during the normal operation and are unpredictable, and we don’t know when, where, and how they will happen. Thus these events have to be dealt with as they occur and cannot be specified in pre-planned mission requirements. While these events often do not have clear signatures, they can have adverse effects on the system and have symptoms that are detectable. However, each event can have multiple symptoms and each symptom can have multiple causes. We can further divide unplanned events into two subtypes: *unpredictable* events and *unexpected* events, based on whether we can provide remedies to maintain acceptable system posture. Unpredictable events are those that we can envision might happen, or are possible, but that we cannot predict, when, where, and how they will manifest. For unpredictable events, key symptoms of the events can lead us to the causes and, therefore, the right remedies to bring the system back to normal operation. Unexpected events, however, are those that cannot even be envisioned. For unexpected events, we either cannot identify the key symptoms to monitor or we don’t know how to further probe for their causes. Thus we would not be able to handle these events in a meaningful, predictable way. The point here is that we can only manage events that we have enough knowledge about, whether it is signatures or symptoms. As we gain more experience about the system and more knowledge about unexpected events, the number of this type of unplanned event should decrease, but likely will never go to zero. Thus, our discussion on QoS management of unplanned events will primarily focus on unpredictable events rather than unexpected events.

From a system point of view, each event can bring the system to a different state. From a QoS management point of view, there is little difference

¹ This work has been partially supported under DARPA contract number NBCHC030119. Approved for Public Release, Distribution Unlimited.

between a planned and an unplanned event in that our goal is to detect the current state and apply appropriate adaptation strategies so that the system performance continues to meet its mission requirements. However, there are differences in operational details in handling these two types of events, which we'll discuss in this paper.

The current practice for detecting and reacting to dynamic QoS events is often application specific and involves manual, ad hoc tailoring. The resulting systems are often rigid, expensive to maintain, and lack portability and scalability. These shortcomings are due to the following reasons:

- QoS management implementations are often entangled with functional code and are written in different languages. The tight coupling not only complicates the design but also drastically reduces reusability and portability.
- Current QoS management is often focused on single, localized resource management issues, such as network bandwidth, CPU, or time delay. While managing individual resources is essential, managing them in isolation cannot achieve system-wide QoS requirements. This is because QoS features are often interrelated and changing one could affect others in the system. For example, altering bandwidth and CPU allocation can impact time delay, and changing CPU allocation can impact bandwidth consumption, etc. As a result, localized remedial actions need to coordinate to achieve desired QoS requirements. Therefore, QoS management in DRE systems has to be a coordinated, system-wide activity to avoid having one problem cascade into another as attempts at remediation occur.

Research has been conducted to construct runtime system abstractions and mechanisms to separate QoS adaptive code, which can be used to respond to unplanned events, from functional code and to make them portable across platforms. One approach to this is represented by the Quality Objects (QuO) middleware framework. However, as yet there is still no off-the-shelf composite architecture to guide the design of system-wide QoS management involving multiple QoS concerns and their tradeoffs in large DRE systems, another prerequisite to effective response to unplanned events. In the rest of this paper, we first describe some previously developed enabling underlying concepts toward an architecture that we believe could be

effective in handling unplanned events. We then discuss unplanned event handling in this context.

2. Our approach and technology base

The QuO middleware framework provides a template implementation base for designing and instantiating adaptation strategies for both planned and unplanned events independent of functional code, and the resulting QoS management code can be intelligently weaved into the functional code for the system and its applications. We briefly review the key components of QuO and its support for QoS management.

- System condition objects provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured. These are the sensors and effectors of the system under construction. In addition, there is a Resource Status Service which helps to organize and aggregate these sensors into more useful and manageable forms.
- Contracts encapsulate decision engines for the adaptation strategies by specifying the levels of services desired or available, and manage adaptive behaviors by switching between different levels of services based on the probing results from the system condition objects.
- Callback objects provide notification interfaces for feedback paths.
- Delegates acting as local proxies for remote objects add local adaptive behaviors (enforcements) based upon the current state of QoS in the system, as specified by contracts.
- System Resource Managers provide the policies, schedulers and control strategies for organizing these elements and for directly controlling the computer system resources.

As more large-scale distributed applications are built with off-the-shelf standard-based components, we are enhancing the standard CORBA Component Model (CCM) with support for dynamic adaptation behaviors. In the current component version of QuO, design-time QoS adaptive behavior is encapsulated into distributed run-time qosket components, and qosket components can operate in-band between functional component interactions or out-of-band between the functional components and the containers, platform, and resources that make up the environment. A qosket component can be incorporated just like a functional component in the assembly of an application. QuO

based component support is addressing issues in construction by composition of both the functional components and the QoS-enabled QuO components with CCM assembly tools. The ease with which we can dynamically insert new behavior will be related to the ease in which we can respond more precisely to unplanned events.

To help localize and concentrate decision-making and detection capabilities, we employ a multi-layer architecture (scalability is another major factor driving multi-layer QoS management architectures). QoS management is divided into multiple layers. At each layer there are local QoS managers, with each layer up covering a larger and larger scope of QoS management in the system. A typical medium scale DRE system might have three resource management layers: individual node layer, application string layer and multi-string system layer. The number of QoS management layers can scale up and down based on the size and characteristics of the DRE system. Higher layer QoS requirements are mapped to more concrete lower layer QoS requirements, all the way down to the individual node/resource layer.

In order to effectively manage QoS, we should be able to detect events/conditions that could potentially impact the behavior of the system, analyze them to determine the strategies to handle them, and react (adapt) with appropriate strategies. Therefore, at each QoS management layer we can divide a QoS manager into three logical units:

- Detection unit - Monitor local conditions and events, both planned and unplanned, through probes into the system and operating environment. These conditions/events can be specified with absolute or preferred upper bounds, lower bounds, or both on the probes. Most unplanned events violate, or cause violation of, absolute bounds, and (at worst) are detectable based on the symptoms they provoke. The key here is to correctly identify and monitor these symptoms related to possible unplanned events. This is the first step toward a coordinated adaptation strategy. After all, if we can't detect, we can't react.
- Decision unit - Determine the adaptation strategies based on the information from the detection unit as well as the control/policy signals pushed down from higher layer QoS managers and feedback from the lower layer QoS managers. Feedback is generated so that the information can be communicated to the

higher layer QoS manager. The decision can be made in at least two different ways: top-down control and local autonomous control. A top-down control decision is made when there is adaptation strategy mandated by the higher layer QoS manager. In this case, higher layer strategy is translated and enforced. In the absence of higher layer control signals, however, local autonomous decisions can be made using information from the detection unit and lower layer feedback. Both of these controls have to coexist in the system and coordination is key to handling unplanned events. Once the symptoms have been detected for an unplanned event, rapid local reaction can take the form of getting the system back into a state in which it is under "control" (rather than having to solve the problem completely). More strategic decisions from higher layer QoS managers, which have more system-wide information, would focus on helping the system return to its more predictable, normal operating condition through top-down control. However, this strategic decision would normally come much slower than local reaction. For unexpected events, the strategic decision may never come (pending offline analysis and system enhancement). In this case, lower layer autonomous controls are often insufficient or unable to completely stabilize the symptoms. For planned events, the adaptation is likely more focused directly on control instead of reaction and control.

- Reaction unit - Execute and push the adaptation strategies down to the lower layer QoS manager for enforcement. The task for this unit is very simple: enforcing the adaptation strategies determined by the decision unit. Strategies against unplanned events could include direct manipulation of system behaviors and dynamic reconfiguration and deployment. New strategies would be added as experience increases.

Note that these are logical units, thus in practice they can be implemented in any number of ways with different footprint and side effects. However, we believe that keeping them separate improves the code reusability and makes larger system QoS design more manageable. In addition, it simplifies adding and amending new behaviors, as experience provides more detailed analysis of real operating conditions, but doesn't change the structural or architectural form.

Detection units at each layer probe events that are related to this while information about lower

layers is provided as feedback to the decision unit. These two sources of information give QoS managers more complete views of the subsystems under their control. Due to their potential for immediate negative impact on the correct operation of the system, it is critical that we develop local strategies and provide rapid reaction to the symptoms of unplanned events based on local information. The goal here is to stop bad things that are happening and to get into a more controllable state while higher layer manager(s) figure out a corrective strategy to bring the system back to normal operation. Diagnosing the cause and deriving a corrective remedy is anticipated to be much slower than the local reaction to the symptom and will often require further probing of the system and coordination of several QoS management layers. We have to pay special attention to the design and coordination of these control and reactive strategies for unplanned events. The local reactions have to be constrained enough to keep them from being exploited and to keep them from affecting the system negatively, but broad enough to handle a wide range of symptoms. Thrashing control is a significant issue, especially when root causes are still unknown

The described architecture minimizes the communication by distributing adaptation strategies and provides the flexibility to handle unplanned events at multiple QoS management layers. Various QoS management design approaches, from centralized to highly distributed and combined strategies in between, can be accommodated within the same architecture. Handling unplanned events in this fashion is analogous to the way exceptions are handled in various programming languages. During the design stage, we need to anticipate classes of unplanned events, decide how to detect them, and provide appropriate handlers (adaptation strategies) including default handlers of last resort, so that they can be dealt with accordingly at runtime. Like exception handling, it is key to detect the symptoms as soon as possible, followed by local and/or system-wide coordinated corrective actions. Also like exception handling in static programs, we can expect to update it frequently as new information about unplanned events is discovered, making them unpredictable, but no longer unexpected events.