Dynamic QoS Management in Distributed Real-time Embedded Systems

By Joe Loyall and Rick Schantz

Chapter for
Handbook of Real-Time Embedded Systems
(in Experiences with Real-Time and Embedded Systems)

1 Introduction

Increasingly, embedded systems are part of larger distributed real-time embedded (DRE) systems in a wide variety of domains, including military command and control (C2), avionics and air traffic control, and medicine and emergency response. DRE systems combine the stringent quality of service (QoS) requirements of traditional closed embedded systems with the challenges of the dynamic conditions associated with being widely distributed across an often volatile network environment. Traditionally, embedded systems have been able to rely on their closed environments and self-contained bus architectures to limit the dynamic inputs possible and could rely on static resource management techniques to provide the QoS and reliable performance they need. The environment of distributed, networked systems is more open with heterogeneous platforms, where inputs can come from external devices and platforms, and dynamic, in which conditions, resource availability, and interactions can change. Because of this, achieving the necessary predictable real-time behavior in these DRE systems relies on the ability to manage resources end-to-end, map system level requirements to platform level controls, aggregate conflicting requirements, and adapt and reconfigure to changing conditions.

Middleware, such as CORBA, is being applied to these types of applications because of its ability to abstract issues of distribution, heterogeneity, and programming language from the design of systems. CORBA, specifically, has spearheaded this trend because of its development of standards supporting the needs of DRE systems, such as RTCORBA [14], FT-CORBA [12], and Minimum CORBA [13].

This chapter describes research efforts to develop middleware for providing dynamic, adaptive QoS management in DRE systems. We describe some of the issues in providing a middleware platform for QoS adaptive systems, the middleware solutions we have developed as part of our research, and case studies applying them to the DRE system context.

2 Issues in Providing QoS Management in DRE Systems

In DRE applications, quality of the service provided is as important as functionality, i.e., how well an application performs its function is as important as what it does. Many DRE applications control physical, chemical, biological, or defense processes and devices in real-time, so that degraded performance or reliability could have catastrophic consequences.

As these traditionally closed embedded systems have become networked, they have formed DRE systems that consist of

- Multiple competing end-to-end streams of processing and information;
- Changing numbers and types of participants, with changing roles and relative importances in the system’s overall mission or goal; and
- Heterogeneous, shared, and constrained resources.
QoS management is a key element of the design and runtime behavior of DRE systems, but it is often defined in terms of management of individual resources, e.g., the admission control provided by network management or CPU scheduling mechanisms or services. While individual resource management is necessary, it is not sufficient in DRE systems because:

- Effective QoS management spans individual resources. The consumer of information determines the QoS requirements, which might change over time, while the information source (frequently remote from the consumer and therefore using different resources) and transport determine the quality and form of information.
- Management of individual resources implies a local view, while effective resource management must take into account the mission requirements, the relative importance of resource users to a mission, and what constitutes effective resource usage to achieve mission goals.
- Effective QoS management must mediate the contention for resources from many simultaneously operating applications, streams of information, and systems.
- There might be multiple, simultaneous bottlenecks (i.e., the most constrained resources) and the bottlenecks might change over time.

QoS management for DRE systems must therefore capture the QoS requirements from the mission requirements, manage all the resources that could be bottlenecks, mediate conflicting demands for resources, effectively utilize allocated resources, and dynamically reallocate as conditions change.

Based on the above bullet items, we break the discussion of issues in providing QoS management in DRE systems into the following four major sub-sections, each of which we will discuss in turn:

- **End-to-end QoS management** – The management of QoS for an individual end-to-end information stream, from information sources to information consumers. That is, managing the resources associated with information collection, processing, and delivery to satisfy a particular use of information.
- **Multi-layer QoS management** – The management of QoS for a mission or set of high-level operational goals, which includes the mapping of high-level, system-wide concepts into policies driving QoS at the lower levels, followed by enforcement at the lowest level.
- **Aggregate QoS management** – The mediation of demands and negotiation for resources between multiple end-to-end streams that are competing for resources.
- **Dynamic QoS management** – Adapting to changes in resource availability, mission and application needs, and environmental conditions (e.g., number of elements under QoS management, failures, attacks) to maintain, improve, or gracefully degrade QoS.

### 2.1 The Need for End-to-end QoS Management

QoS is often defined in terms of management of individual resources, e.g., the admission control provided by network management or CPU scheduling mechanisms or services. While managing the resources at a bottleneck might be sufficient at any given time, it is important to understand the consequences of managing the resources only at a specific point. Eliminating a bottleneck by providing additional resources might simply expose a different bottleneck elsewhere that must also be managed. For example, allocating more bandwidth (e.g., using bandwidth reservation, differentiated services, alternate paths, or reducing the other load on the network) might simply expose that there isn’t enough available CPU at a node along the end-to-end path to process the now plentiful data.

Furthermore, shaping application or data usage to fit a specific bottleneck can also have consequences that change, but do not eliminate, the bottleneck. For example, an application facing a constrained network can use data compression to consume less bandwidth. However, in doing so the application is consuming more CPU, which might or might not be available. There might be multiple, simultaneous bottlenecks, the bottlenecks might change over time, and effective QoS management spans individual resources.

The end user of information and the purpose for which it is used determine the QoS requirements for the information, but the information source and transport determine what quality can be provided. Therefore
QoS management for DRE systems must be end-to-end, i.e., it involves capturing the QoS requirements of the information user; managing the quality of the information production, processing, and delivery; managing the resources from end-to-end that could be bottlenecks; and shaping application or data usage to effectively use the managed resources.

2.2 The Need for Multi-Layer QoS Management

Effective QoS management requires resolving multiple views of QoS and resource management, as illustrated in Figure 1, ranging from higher-layer views closer to total system or mission goals and expressed in higher-level terms down to lowest layer views closer to individual resources and their controls. Although there can be many layers, we identify three in particular:

- Mission/System layer view. At the system layer, there is the knowledge of the mission goals, the applications in the system, and the available resources. This is also the layer at which there is an understanding of the relative importance of applications to mission goals, resource allocation strategies for each goal, and the policies for mediating conflicting application resource needs.

- Application/Application string layer view. An application view of resource management involves acquiring whatever resources are needed to meet a single end-to-end application string’s (i.e., a distributed application from the information source to information consumer) requirements and to effectively utilize the resources available to them. If there are not enough resources available, then an application needs to be flexible enough to adjust its resource needs (with corresponding graceful degradation of its functionality) or it will fail, e.g., throw an exception. Applications greedily acquiring all the resources they need does not scale in dynamic and resource constrained environments, but applications cooperating to share resources requires sophisticated coordination and control.

- Resource layer view. Resource specific mechanisms control access to resources, deciding whether and how a request for a resource allocation should be granted. Typical CPU and network resource allocation is based on priorities or reservations. Typical resource allocation mechanisms have little or no knowledge of the applications and missions using them or their requirements, although some limited information can be propagated to the resource level in the form of relative priorities and reservation requests.

Trying to manage QoS at any one of these layers is insufficient. Managing QoS only at the system layer results in specification without enforcement; only at the application layer results in limited context and ability to enforce; and only at the resource layer results in control without context and localized decisions without understanding of their global consequences. It is necessary to combine the system, application and resource views to effectively manage and provision QoS. The system layer determines effective allocations; the application layer chooses how to effectively use the allocations; and resource mechanisms manage access to shared and constrained resources.

2.3 Need for Aggregate QoS Management

In a distributed system, resources are frequently shared among end-to-end application strings. The sharing might be as simple as a network backbone through which all traffic from end-to-end travels, but can also include computation resources (at either end or in the middle) and less obvious contended resources such as screen real estate or attention from a human operator.

![Figure 1. Multi-layered view of QoS management.](image-url)
Free-for-all unmanaged environments, such as the Internet, purposefully provide no QoS guarantees, in the form of best effort for all. In contrast, to get the predictable performance and control that DRE systems need, aggregate QoS management must be provided, i.e., QoS management must mediate the conflicting demands for resources, enforce starvation and fairness policies, and place constraints on individual applications’ usage of shared resources.

Application layer QoS control without aggregate QoS management leads to a situation called the tragedy of the commons [6]. In this phenomenon, there is no incentive for applications to share resources and, in fact, there is a disincentive to do. Each application grabs as many resources as it can and the entire system degrades to best effort. Aggregate QoS management is necessary to control this and ensure controllable QoS across applications. Approaches to aggregate QoS management include

- **Negotiation** – Applications negotiate for access to resources and then use only those that they have been assigned through agreement with other applications. An example is a commerce or free market approach, in which applications get a certain amount of currency to spend on resource allocations and QoS guarantees.

- **Hierarchical** – As described in the multi-layer section above, a management entity with a higher level view determines which applications are relatively more important, or more in need of resources, than others. The higher layer manager imposes policies and constraints to which the applications must adhere.

- **Static** – The traditional approach in embedded systems uses off-line analysis of the system to determine an allocation of resources across the applications. This approach has limitations in DRE systems, as described in the next section.

### 2.4 Need for Dynamic QoS Management

The move from self-enclosed embedded systems to distributed embedded systems is also one of moving from static to dynamic environments. DRE systems are frequently parts of larger systems of systems, with participants that come and go, deployment in hostile environments, changing resource availability and contention, failures and cyberattacks, and other dynamic conditions. Static resource allocations might be appropriate for a situation at a single point in time, e.g., initial deployment, but quickly become insufficient as conditions change.

Designing and implementing QoS management into a system therefore requires building in the control and flexibility to manage changes in resource availability and mission requirements. In other words, effective QoS management relies on recognizing where bottlenecks exist at any given point in time and effectively managing resources to remove the bottlenecks; dynamically adapting application functionality to compensate for the bottlenecks and meet system requirements; or both.

### 3 Solutions for Providing QoS Management in DRE Systems

#### 3.1 Middleware for Dynamic QoS Management

Although it is possible to provide end-to-end QoS management by embedding QoS control and monitoring statements throughout a software system, such an approach leads to additional code complexity, reduced maintainability, and non-reusable software. A better approach is to separate the QoS concerns from the functional concerns of an application and combine the two into a QoS-managed software system through integration at a middleware layer.

An approach that we have taken to do this is providing extensions to standards-based middleware that allow aspects of dynamic QoS management to be programmed separately and then integrated into distributed object or component-based systems.
### 3.1.1 Quality Objects

Quality Objects (QuO) is a distributed object framework that supports the separate programming of (1) QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for controlling and providing QoS and for adapting to QoS variations that occur at run-time. By providing these features, QuO separates the role of functional application development from the role of developing the QoS behavior of the system.

As shown in Figure 2, a QuO application inserts additional steps in the remote method invocation (RMI) of distributed object applications. The QuO runtime monitors the state of QoS before each remote invocation through the use of System Condition Objects that provide a standard interface to observable and controllable parameters in a platform, such as CPU utilization or bandwidth usage. Delegates intercept remote calls and a contract decides the appropriate behavior to apply. The contract defines the set of possible states of QoS in the system using predicates of system condition object values. Based upon the current QoS state, the contract could (1) specify additional processing to perform; or (2) allow the method call to proceed as is; (3) redirect the invocation to a different method; or (4) invoke a callback on the application to alter its execution.

QuO supports in-band and out-of-band QoS control and adaptation, as illustrated in Figure 3. QuO delegates trigger in-band adaptation by making choices upon method calls and returns. Contracts trigger out-of-band adaptation when changes in observed system condition objects cause state transitions. The QuO middleware is described in more detail in [9][10][16][17][18].

### 3.1.2 Qoskets and Qosket Components

One goal of QuO is to separate the role of QoS programmer from that of application programmer. A complementary goal of this separation of programming roles is that QoS management code can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to an application program. To support this goal, we have defined Qoskets as a unit of encapsulation and reuse in QuO applications. Qoskets are used to bundle in one place...
all of the specifications and objects for controlling systemic behavior, as illustrated in Figure 4. Qoskets encapsulate the following QoS aspects:

- Adaptation and control policies – As expressed in QuO contracts and controllers
- Measurement and control interfaces – As defined by system condition objects and callback objects
- Adaptive behaviors – Some of which are partially specified until they are specialized to a functional interface.
- QoS implementation – Defined by qosket methods.

A Qosket Component is an executable unit of encapsulation of Qosket code wrapped inside standards-compliant components, such as the CORBA Component Model (CCM) [3], which can be assembled and deployed using existing tools. They expose interfaces, so they can be integrated between functional components and services to intercept and adapt the interactions between them. Each Qosket component offers interception ports that can be used to provide in-band adaptation along the functional path. Qosket components can also provide ports to support out-of-band adaptation and interact with related QoS management mechanisms, as illustrated in Figure 5.

Figure 5. Qosket components can be assembled in-band or out-of-band in a component application

Qosket components provide all the features of Qoskets and all the features of components to provide life-cycle support for design, assembly, and deployment. Each qosket component can have as many adaptive behaviors as desired. However, encoding each qosket with one and only one adaptive behavior decouples different adaptive behaviors and increases the reusability of each. The tradeoff is between the assembly time flexibility allowed by the separation of QoS behaviors versus the performance overhead of having additional components to assemble. This is the same design versus performance tradeoff that exists in functional component based applications and which can be alleviated by assembly tools and component implementations that optimize component and container instantiations. Implementations that encapsulate a single QoS behavior in each qosket component can provide an aggregate, end-to-end behavior by combining qosket components.

The ability to enable dynamic adaptive QoS behavior by assembling qosket components has several important advantages. First, it makes it possible to manage end-to-end QoS requirements without requiring the middleware to provide such a service. This is important because there is no agreed-upon QoS standard for component standards (although there are proposals being considered). Second, the assembly, packaging, and deployment tools provided by the component middleware can be used for qosket components. Third, we have been able to create qosket components that are independent of the underlying middleware by separating code specific to the particular component model from the generic code required to manage and evaluate the QoS contracts. To date we have built qosket components for the CCM-based MICO [11] and CIAO [2], the JavaBean-based Cougaar [4], and an avionics domain-specific component model based on CCM (PrISm) [15].

3.1.2.1 Examples of Qosket Components

This section provides examples of the qosket components that we have developed and applied to the QoS management of DRE systems, such as those described in Section 4.
Network Management. We have constructed qoskets that prioritize network traffic by setting DiffServ CodePoints (DSCPs) [7] and that reserve bandwidth using RSVP [19].

CPU Management. We have developed a qosket that reserves CPU, using a CPUBroker [5], in Timesys’s Linux RT operating system [8].

Application Adaptation and Data Shaping. The application needs to be able to shape its behavior to effectively use whatever resources are available to it and to gracefully handle degraded situations. This includes the ability to shape data so that whatever resources are available are used to deliver and process the data that is most important to the application’s mission. To support this, we have developed the following set of data shaping qoskets, some of them specific to particular types of data:

- **Compression.** This qosket invokes off-the-shelf compression algorithms to compress data. We have versions that compress images using PNG, JPEG, and wavelet compression algorithms. When assembled as qosket components, these need a corresponding decompression qosket at the receiving end.

- **Fragmentation.** For large size data that needs to be send over a period of time, such as a high resolution image, a large database entry, or a large XML document, this qosket breaks the data into smaller chunks (the size is configurable). When assembled as a qosket component, this needs a corresponding reassembly qosket at the receiving end.

- **Tiling.** This is an image specific version of the fragmentation qosket, which breaks an image into tiles, each of which is a displayable part of the larger image. This allows each tile to be displayed as it is received, without waiting for the entire image to be received.

- **Pacing.** This is a qosket used to control jitter and is particularly useful when combined with the fragmentation and tiling qoskets. It introduces packets into the network at a steady pace over a period of time to avoid bursty traffic.

- **Scaling.** Developed specifically for imagery, this qosket scales the size of an image, trading off resolution for content. The smaller scaled image will represent the same full picture, but will be reduced in resolution and size. A thumbnail image is an example of an image reduced in scale.

- **Cropping.** Again, developed specifically for imagery, this qosket reduces the size of an image by dropping whole parts of the image. It retains the resolution of the original, but loses part of the content and is useful when details about parts of an image are more important than others.

3.2 Software Engineering of QoS Management

In addition to middleware support for QoS management in DRE systems, it is useful to have tools and software engineering support for the construction of the QoS management features. This section describes three of these: aspect oriented languages, composition patterns, and design environments.

3.2.1 QoS Aspect Languages

The initial temptation for an application programmer when designing QoS into a program is to modify his code to examine its surroundings and take the appropriate measures. Unfortunately, this extra code tends to tangle the QoS management logic with the application logic, increases the complexity of, and reduces the maintainability of the application code. The recognition that QoS management is a concern that crosscuts the functionality of an application led us to the development of QoS aspect languages, which could be programmed separately and then woven into the application code, or encapsulated as qosket components.

The QoS aspect languages, which we developed as part of the QuO toolkit and are described in [20], are consistent with the principles of traditional AOP [21] in that it defines specific points in the runtime behavior of a program at which cross-cutting advice can be invoked. However, it extends the join point model provided by many instantiations of AOP, which have concentrated on insertion points in the functional language semantics, such as control flow, data flow, and field accesses. The QuO aspect model extends this notion to associate advice with the distribution of method calls, as illustrated in Figure 6. This complements the use of other AOP languages, which can be used to weave cross-cutting functionality into application
functional components, by providing the ability to weave in aspects between these components as they are distributed throughout a system.

In addition, QuO elements, e.g., contracts and system condition objects, provide the ability to weave code between the application and the system in which it is executing. This allows aspect-oriented insertion of cross-cutting behaviors affecting the interactions between the program execution and the execution of the rest of the elements of the system, environment, and infrastructure in which the program is executing. These interactions are outside the standard join point model definition, since the interactions can be asynchronous from the execution path of the program. For example, QuO contracts and system condition objects can recognize and respond to overload in the system, regardless of the current state of the program execution. Furthermore, the response could affect the future execution of the program, regardless of where in the execution the program is currently. This combination of systemic programming and the aspect weaving described above enables the separate programming of QoS management that cross-cuts the functional dominant decomposition of the program by influencing the application behavior only, the system only, or both the application and the system.

### 3.2.2 Composition and Composition Patterns

A DRE application might consist of many components – both functional and QoS. In the rare case in which these components are independent, the order of assembly or composition can be unimportant. However, in the usual case, the order of execution of these components must be carefully crafted to achieve the desired end-to-end, aggregate behavior. In some cases the way in which components are assembled is the difference between correct and incorrect behavior:

- Some qosket components must coordinate to implement a desired behavior. For example, a compression qosket must frequently be paired with a decompression qosket. These must be assembled in the correct order, since decompressing prior to compressing can result in undesired behavior.
- Some qosket components interfere with one another in a contradictory manner. For example, compressing or encrypting data might result in the inability to scale, crop, or tile data because it changes the data into a format that can no longer be manipulated. These qosket components must be composed in a compatible way, e.g., scale or crop prior to compression, or composed using a decision, e.g., a contract determines whether to crop or compress, but not both.
- Some qosket components can affect the dynamics of one another. For example, any qosket component that includes processing, such as compression or encryption, can affect the dynamics of a CPU management qosket.

While the case studies in Section 4 describe specific compositions that we used to get the desired end-to-end QoS behavior in specific instances, there are a few general composition techniques that we have extracted that serve as patterns of composition, illustrated in Figure 7:

- **Layered Composition** – In this pattern, qosket components that make higher level decisions (such as a System Resource Manager, SRM) are layered upon qosket components that enforce these decisions (such as Local Resource Managers, LRM), which in turn are layered upon mechanism qosket components that control resources and QoS behaviors.
- **Parallel Composition** – When qosket components receive data simultaneously and perform their QoS behaviors independently, they can be composed in parallel.
• Sequential Composition – In some cases, a set of components must be tightly integrated such that a set of QoS behaviors are performed sequentially, with the output of each component becoming the input to the next component.

3.2.3 Design Environments

The middleware and software engineering technologies that we have described help improve the programming of DRE systems. Another critical area of software engineering support is help for designing these systems. Previous work in design environments have made designing and developing the functional logic of a distributed system much easier; however, the same support is not there for designing the QoS features. As part of the research described in this chapter, we created a prototype QoS design tool and applied it and functional design tools to the design of the case studies in Section 4.

Figure 7. Composition patterns recognized in our uses of qosket components.

Figure 8. DQME enables the design of QoS Management
The Distributed QoS Modeling Environment (DQME), illustrated in Figure 8, was developed by BBN and Vanderbilt University and focuses on a QoS centric view of a distributed system. It captures the following essential elements of QoS and their interactions:

- **Mission requirements** – The functional and QoS goals that must be met by the application; relative importance of tasks; and minimum requirements on performance or fidelity. These help determine the relative importance of QoS controls, adaptation strategies and tradeoffs.

- **Observable parameters** – Before any QoS control action can be taken, the system has to know its current state with regard to the QoS of interest. These parameters determine the application’s current state and are also important to help determine and/or predict the system’s next state.

- **Controllable parameters and adaptation behaviors** – These are the knobs available to the application for QoS control and adaptation. These can be in the form of interfaces to mechanisms, managers or resources, or in the form of packaged adaptations, such as those provided by QuO’s Qosket encapsulation capability (Section 3.1.2).

- **System dynamics** – Based on the current state of the system (observable parameters) and the set of knobs available (controllable parameters and adaptation behaviors), there could be several options for adaptation. The interactions between these are reflected in the system dynamics and can help define the set of possible trajectories that an application can take.

- **Control strategies** – Specification of the control actions employed and the tradeoffs made in response to dynamic system conditions in order to maintain an acceptable mission behavior. When making a decision, the control strategies take into account the mission requirements, current system state, and system dynamics. DQME supports the following types of controllers: state machines, supervisory control, feedback control, search spaces, region spaces, and classical compensators.

DQME supports hierarchical models of the above elements, with the higher levels representing the elements of end-to-end QoS adaptations in support of system wide goals. The model developer can descend into these models to create models of local adaptations, based upon local parameters, but coordinated with the other subordinate models in support of the higher level goals.

## 4 Case Studies of Providing QoS Management for DRE Systems of Increasing Complexity

### 4.1 HiPer-D, Reactive QoS Management in a US Navy Testbed

Our first case study applied the principles of dynamic QoS management to a US Navy context and the problem of working with Unmanned Aerial Vehicles (UAVs). A UAV is an autonomous or remote-controlled, unpiloted aircraft that is launched in order to obtain a current view of an area of concern. A UAV can receive remote-control commands from a control platform in order to perform actions such as changing its area of interest or tracking a target. To support UAV operations, a video camera on the UAV produces an imagery stream that, under certain modes of operation, must be displayed with requirements for minimal and predictable real-time delay and fidelity, on consoles throughout a ship. As Figure 9 illustrates, there are several steps to this process:

1. Video capture and feed from the off-board source (UAV) to the ship.
2. A process on the ship receives the video feed and distributes it to control stations on the ship's network.
3. Users' hosts receive video and display it.
4. Users analyze the data and (at certain control stations) send commands back to the UAV to help control it.
Our prototype simulates the first three of these steps. The command phase of the fourth step is formulated as a requirement to be able to control the QoS of the video distribution properly. The video must be displayed on the end user’s video monitor in a timely manner – if the data is too stale, it will not represent the current situation of the physical UAV and the scene it is observing – and with at least the minimum fidelity needed for visual cues – the user must see enough of the important imagery and motion. Otherwise, the user cannot control the vehicle appropriately or detect important features from the video stream. This means, among other things, that we cannot rely on the “best-effort” characteristics of common network transport protocols. The retransmission of dropped packets by TCP introduces unacceptable delay (violating the timeliness requirement) while UDP is not discriminative in its packet dropping – it is as likely to drop packets from important images as from less important ones. We choose to control the QoS in the QuO middleware layer where the application’s requirements can directly influence the QoS control and adaptation policies that are available and used for this use case. For example, it is not acceptable to suspend the display during a period of network congestion and resume the display from the same point in the video flow when bandwidth is restored. It is also not acceptable to simply drop arbitrary frames or to continuously attempt to retransmit lost frames. Tradeoffs must often be made of one property (e.g., timeliness) against another property (e.g., fidelity) based on the particular requirements of the end-user at that moment.

Figure 8 illustrates the architecture of the prototype. It is a three-stage pipeline, with simulated UAVs or live UAV surrogates sending imagery to processes (distributors), which distribute the imagery to the simulated C2 stations, which display, collect, or process the imagery (e.g., with an automated target recognition process). Various versions of the UAV image dissemination prototype have been used for evaluation within the US Navy’s HiPER-D platform, as a basis for US Army and US Air Force prototype systems, and as an open experimental platform for DARPA’s Program Composition of Embedded Systems (PCES) program. The UAV prototype uses the QuO middleware to integrate several QoS management strategies, services, and mechanisms:

- End-to-end priority-based resource management, using RTCORBA and scheduling services for CPU resource management and Differentiated Services (DiffServ) [7] for network priority management.
- End-to-end reservation-based resource management, using the CPU reservation capabilities of Timesys Linux and RSVP network reservation [19].
- Application-level management, including data management strategies (e.g., filtering, tiling, compression, scaling, and rate shaping) and process migration.

The UAV image dissemination prototype employs QoS management all along the pipeline, with CPU management at each node, network management at each link, and in-band and out-of-band application QoS management at several locations. As illustrated in Figure 11, we used aspect-oriented programming techniques to separate the QoS and adaptive control aspects from the functional aspects of the program.

We performed experiments to test and evaluate the effectiveness of the frame-dropping adaptation in the UAV application. The three stages were run on three Linux boxes, each with a 200MHz processor and 128MB of memory using TCP.
At time $t=0$, the distributor started. Shortly after this, the video began to flow. At $t=60$ seconds, $t=62$ seconds, and $t=64$ seconds, three load-simulating processes were started on the same host as the distributor, each attempting to use 20 percent of the maximum processing load (a total of 60 percent additional processing load). This reduced the distributor's share of processing power below what it needed to transmit video at 30 frames per second. At $t=124$ seconds, the load was removed. At time $t=300$ seconds (approximately), the experiment terminated. The basic premise is that the full load was applied for one minute, starting after the pipeline had time to “settle in,” and ending a few minutes before the end of measurement so we could observe any trailing effects.

This scenario was run twice, once without QuO attached and without any adaptation (the control case) and once with a QuO contract controlling adaptation (the experimental case). For the purposes of this experiment, the only adaptation enabled was to reduce bandwidth by selectively dropping frames. Figure 12 shows the effect of the increased load on the latency of the video stream. In this graph, the x-axis represents the passage of time in the scene being viewed by the video source and the y-axis represents the “lateness” of each image, i.e., the additional latency (in delivery to the viewer) caused by the system load. If all images were delivered with the same latency, the graph would be a constant zero. The label “Interval of Load” indicates the period of time during which there was excessive contention for the processor. Without QuO adaptation, the video images fall progressively further behind starting when the contention first occurs, and the video does not fully recover until some time after the contention disappears.

The outcome of this experiment demonstrates that adaptation can provide improved performance of the application in the form of smoothing critical performance metrics over changing conditions. The added latency caused by adverse system conditions (in this case, excessive CPU load) occurs in a sharply reduced magnitude and duration when adaptation is enabled, and the video image is continuously usable for its intended real-time purpose despite the fluctuation.

### 4.2 WSOA, Hybrid Control and Reactive QoS Management in a Dynamic Mission Replanning US Air Force Flight Demonstration

The Weapon System Open Architecture (WSOA) program was a US Air Force project to develop and prototype capabilities for aircraft to do mission replanning while airborne en route to a target (as opposed to on the ground prior to a mission). WSOA prototyped a dynamic collaborative mission planning capability that could be used in-flight between a C2 aircraft and a fighter aircraft. As illustrated in Figure 13, the fighter aircraft and the C2 aircraft establish a collaboration to exchange virtual target folders, consisting of images and plain text data to update the fighter’s mission. The end goals are to shorten the mission planning time (from hours to minutes) and increase the confidence in mission execution.
WSOA was built upon a layered architecture of domain-specific, QoS adaptive, and distribution middleware; as well as QoS mechanisms, services, and managers. Middleware on the fighter node consisted of

- The Bold Stroke avionics middleware [22]
- QuO QoS adaptive middleware
- TAO distribution middleware

The C2 node utilized ORBexpress, a CORBA ORB supporting the C2 legacy software applications written in Ada, and Visibroker. QoS mechanisms, services, and managers in WSOA consist of

- RT-ARM [23], a CPU admission control service, on the fighter node
- Kokyu [24], a real-time scheduling and dispatching service, on the fighter node
- Tactical communication links, using Link 16 [ref].

Processing on the fighter node involves hard and soft real-time tasks. Hard real-time tasks are statically scheduled to assign rates and priorities. WSOA used QuO, RT-ARM, and Kokyu to schedule soft real-time tasks in the extra CPU available for the worst case, but unused in the usual case, and to adapt processing and task rates dynamically to maintain the operation of hard and soft real-time tasks on the fighter’s mission computer.

The layered architecture of the WSOA is illustrated in Figure 14. The nature of the Link 16 tactical network means that bandwidth is statically allocated, so dynamic management of the network has to be on the edges, i.e., adjusting the amount and nature of the data being transported. WSOA used a QuO contract, illustrated in Figure 15, and a delegate (wrapping a `get_image()` method) to download imagery as a sequence of tiles of varying quality, starting with a point of interest in the image. The QuO contract monitored download progress and the delegate adjusted image tile compression reactively based on the current contract state (i.e., early, on time, or late) to meet the required image transmission deadline. Image tiles near the point of interest – which are the first to be transmitted – were sent at as low compression as possible to improve image quality, but surrounding images can still be useful at lower resolution. In addition, the QuO contract interfaced with RT-ARM to...
adjust the rates of execution for the decompression operation.

On December 11, 2002, Boeing and the USAF conducted a flight demonstration of the WSOA software in Missouri. An F-15 aircraft equipped with the WSOA software took off from Lambert-St. Louis International Airport with a preplanned mission and target – an exposed aircraft just Northwest of Hannibal, Missouri, over 100 miles away. An airborne simulated AWACS, hosted in a Boeing 737 flying laboratory, received a simulated alert from a Joint Tactical Terminal (JTT) of a time critical target, in this case an SA10 surface to air missile being set up due west of Hannibal. The F-15 and AWACS established an airborne collaboration, exchanged intelligence, imagery, and target information and the F-15 redirected its mission to simulate prosecution of the SA10. Within 20 seconds of establishing the collaboration, the F-15 Weapon System officer was receiving useful imagery through the Link-16 tactical link between the two aircraft.

We evaluated image download times from the C2 node to F-15 compared to deadlines of 38, 42, 46, 50, 54, and 58 seconds. As Figure 16 illustrates, the 38 second deadline is lower than the measured latency for any image downloaded without adaptation at the highest compression ratio of 100:1, so that meeting it is infeasible. Similarly, a deadline of 58 seconds exceeds the maximum latency of any image at the lowest compression ratio of 50:1, and thus does not require any adaptation.

Figure 17 shows the end-to-end image download latencies with reactive adaptation of compression ratios and adaptation of operation invocation rates. The adaptive QoS management enabled the image transfer within its deadline, in all cases but the tightest (42 seconds) of the feasible deadlines. In the 42 second case, the limited slack in the deadline made the reactive decision to adapt the image fail to meet the deadline in one case.
4.3 PCES, End-To-End and Multi-Layered Controlled QoS Management in a Multi-Platform Live-Flight and Live-Fire Joint USAF and Army Demonstration

As part of DARPA’s Program Composition for Embedded Systems (PCES) program, BBN, Boeing, and Lockheed Martin developed a capstone flight demonstration of advanced capabilities for time critical target (TCT) surveillance, tracking, and engagement. The PCES capstone demonstration, illustrated in Figure 18, was a medium scale DRE application, consisting of several communicating airborne and ground-based heterogeneous nodes in a dynamic environment with changing mission modes, requirements, and conditions. It consisted of a set of UAVs performing theater-wide surveillance and target tracking and sending imagery to, and under the control of, a C2 Center. Specific UAVs could be commanded to concentrate their surveillance on areas of interest. When a positive identification of a threat was made, the commander directed engagement by ground or air combat units. Specific surveillance units then gathered battle damage indication (BDI) imagery and the process was repeated as needed.

The capstone demonstration was conducted at White Sands Missile Range (WSMR) in Spring of 2005. On the north end of the demonstration, we had two live ScanEagle UAVs and four simulated ScanEagles. 100 miles to the south was the C2 Center, which consisted of multiple machines, including situational assessment (SA) displays, command and control processing, and displays of the reconnaissance and BDI imagery. Network connectivity from north to south was through a resource constrained, shared network.

The PCES capstone demonstration application exhibits challenges and issues that are indicative of medium-scale DRE applications with similar characteristics. The following sections discuss how we applied the concepts of Section 3 to address these challenges and build the demonstration.

4.3.1 Mission Modes and Roles Drive System Requirements

The ScanEagle camera transmits imagery through a 2.4 GHz analog television downlink. We captured the imagery from the ScanEagle ground station receiver and piped it through an analog to digital converter, so we could control the resolution, rate, and size of the imagery, based on how the imagery was to be used, the amount of resources available to transmit and process the imagery (i.e., network bandwidth and CPU), and the number of UAVs sharing the resources. The way in which information was to be used in the mission determined the requirements on the system for the delivery, format, and content of surveillance and C2 information. In the PCES capstone demonstration, there were three roles that UAVs play:

- Surveillance
- Target tracking and engagement
- BDI

When the demonstration began all the UAVs were performing surveillance. As the scenario unfolded, the C2 node would direct a UAV to reroute in response to a potential TCT, prompting a change by the UAV to the target tracking role. After target engagement, a UAV would be directed to perform the BDI role.
The TCT mission provided a relative priority to each of these roles, target tracking and engagement was the most critical, BDI second most critical, and surveillance was the least critical. Each role, likewise, included attributes for its desired QoS and the tradeoffs it could afford. A UAV performing target tracking needs to provide high rate and high resolution imagery so that positive target or threat identification can be made. If the target was moving, this translated into full motion MPEG video. If the target was still, then the imagery rate could be lower.

For BDI, the UAV needs to provide high resolution imagery until a human operator determines that he has sufficient detail to discern battle damage. Imagery does not need to start immediately, since dust and smoke will obscure the scene immediately after engagement, but once imagery has started, high resolution imagery must be delivered regularly, although not necessarily at high rate, until a commander decides it is sufficient.

Finally, for UAVs performing surveillance, the primary mission is to maximize the surveilled area with sufficient resolution for a commander to determine an item of interest. This means that imagery from each surveillance UAV must be sent at a sufficient rate to ensure there are no gaps in surveillance coverage and at sufficient size and resolution for a commander to discern command level detail.

The demonstration exhibits the challenges we described in Section XX:

- Multi-layered mapping of the mission- and role-based requirements to QoS enforcement,
- Managing QoS from end-to-end for each UAV sensor to C2 node stream,
- Mediating the aggregate QoS across competing streams, and
- Handling the QoS dynamically as the roles of participants change over time.

To manage these dimensions of QoS in the PCES capstone demonstration, we developed a multi-layered, dynamic QoS management architecture, illustrated in Figure 19. The System Resource Manager (SRM) is a supervisory controller responsible for allocating resources among the system participants and for disseminating system and mission wide policies to local resource managers. These policies include the resource allocation, the relevant mission requirements and parameters, and tradeoffs.

A Local Resource Manager (LRM) is associated with each demonstration participant, such as a UAV sensor. The LRM receives the policy from the system resource manager and translates it into local management and control actions. The LRM is a feedback controller, using the mission requirements, tradeoff information, and allocated resources part of the policy provided to it to determine which QoS behaviors (e.g., CPU management, network management, data shaping, application adaptation) should be employed, in what order, and to what degree. The LRM also monitors the actual behaviors and adjusts as needed to maintain the QoS level.

In order to determine which QoS behaviors to employ, the LRM needs to use a system dynamics model to predict the effect of employing each QoS behavior and combination of QoS behaviors. In Figure 19, we
separately indicate the control and prediction parts of the LRM, the former illustrated as a Controller and the latter as a QoS Predictor. The system dynamics (i.e., effect) of some QoS behaviors can be determined analytically, e.g., the results of cropping an image (i.e., the amount of data in the resulting image) or reserving an amount of bandwidth (i.e., the amount of bandwidth available to the application). Other behaviors have no analytical model (or less accurate ones), e.g., some compression algorithms or setting a network priority (the results of which are difficult to determine analytically without global knowledge of many other external factors). With the former, the QoS predictor contains the model, equation, or formula to predict the behavior. With the latter, the QoS predictor is initialized with experimental data produced in test runs, and updated at runtime with more accurate monitored information.

The QoS mechanism layer consists of encapsulated QoS behaviors that control and monitor the following:

- **Resources**, such as memory, power, or CPU, which can be monitored and controlled through knobs exposed by the resource.
- **Specific QoS mechanisms**, such as network reservation or network priority services that expose interfaces to resource monitoring and control; or QoS managers, such as bandwidth brokers [25] or CPU brokers [5], that provide higher level management abstractions.
- **Application or data adaptation**, such as changing the rate of tasks, algorithms or parameters of functional routines, or shaping the data used or produced by application components.

### 4.3.2 Construction of DRE Systems

The PCES capstone application illustrates challenges in the manner in which medium and large scale DRE systems are, and will be, built. Traditionally, these types of systems are built as one of a kind, stovepiped systems. In contrast, we constructed the multi-layered QoS management and the end-to-end UAV to C2 imagery streams by composing it from reusable QoS and functional components as described in Section 3.2.2.

We implemented the elements of our end-to-end QoS management architecture as qosket components so that they can be assembled with the components of the functional application, as illustrated in Figure 20. The SRM qosket component includes decision making code to decide how resources should be allocated among participants and wrap that allocation into policy, with some monitoring code to determine the number of current participants, the amount and type of shared resources, and other information affecting the policy decision, such as mission states, requirements, and conditions.

The LRM qosket components include decision making code to decide local actions based on the policy, monitoring code to measure the effects of the QoS management, and control code to adjust levels to satisfy the policy. The LRM’s control code is typically limited to setting the proper attributes on the QoS behavior qosket components and invoking them in the proper order.

The assembly also includes as many QoS behavior qosket components as necessary. In the example in Figure 20, we illustrate two types of QoS behavior qosket components, one that does data shaping and another that interfaces to a QoS mechanism.

The base functionality of each end-to-end imagery stream consists of image capture (i.e., the UAV’s camera sensor and associated processing) and image sending (i.e., communicating the imagery off-board) on the UAV; and the image receipt, display, and processing on the C2 node. The image generation rate is a configurable parameter that indicates how often an image should be pushed out, which is determined by the usage requirements of the imagery, and can be different than the rate at which it is collected. We used both
CIAO [2], an implementation of the CORBA Component Model, and PRIStm [15], an avionics specific component model and component successor to the Bold Stroke middleware we used in the WSOA demonstration described in Section 4.2. The full scope of the demonstration system includes a combination of live flight vehicles, ground vehicles, and simulated participants, and is described in [26].

We augment the functional stream with qosket components as described in Section 3 to get end-to-end QoS management. The full assembly for a representative imagery stream is illustrated in Figure 21. There is one SRM component, which we locate at the C2 node, so it is near the receivers and the command authority, both of which provide information needed to determine the mission requirements. In PCES, the situation assessment (SA) command component keeps track of the number and role of participants and serves as the system repository. When something in the system state changes, such as the number or role of participants, the SA component notifies the SRM component. The SRM uses the relative weights of the roles, the importance of each UAV within a role, the number of UAVs, and the amount of resources available, in order to compute a resource allocation for each UAV. It creates a policy structure for each participant consisting of the following:

- The UAV’s role (surveillance, target tracking, or BDI)
- The UAV’s importance relative to others in the role
- Allocations of resources (bandwidth, CPU, network priority)
- Minimum and maximum allowable qualities (frame rate, cropping, scale, compression, and CPU reservation)

This policy event is pushed to each of the LRM components.

There is an LRM component associated with the sender assembly and another one associated with the receiver assembly for each end-to-end stream. Each receives the policy sent by the SRM and updates the relevant QoS Predictors with the minimum and maximum cropping, scaling and compression levels. The LRM then queries the QoS Predictors to get the proper levels to set for each of the data shaping components to fit the allocated bandwidth and CPU. The adaptation strategy and tradeoffs for each role is captured in a model of the system [27] and is used to determine the order of assembly and invocation of components. For example, the strategy we use for the surveillance role is to reduce the rate until the minimum (the slowest rate that does not cause gaps in surveillance); compress until the maximum allowed compression; and scale the image as a last resort if needed. The LRM then sets each of the following QoS mechanism qosket components with the proper settings from the policy and QoS predictors.

![Figure 21. Full assembly for one end-to-end image delivery stream in the PCES Multi-UAV Surveillance and Target Tracking Demonstration](image-url)

There is an LRM component associated with the sender assembly and another one associated with the receiver assembly for each end-to-end stream. Each receives the policy sent by the SRM and updates the relevant QoS Predictors with the minimum and maximum cropping, scaling and compression levels. The LRM then queries the QoS Predictors to get the proper levels to set for each of the data shaping components to fit the allocated bandwidth and CPU. The adaptation strategy and tradeoffs for each role is captured in a model of the system [27] and is used to determine the order of assembly and invocation of components. For example, the strategy we use for the surveillance role is to reduce the rate until the minimum (the slowest rate that does not cause gaps in surveillance); compress until the maximum allowed compression; and scale the image as a last resort if needed. The LRM then sets each of the following QoS mechanism qosket components with the proper settings from the policy and QoS predictors.
The **Diffserv qosket component** is responsible for setting DiffServ codepoints (DSCPs) on component containers. The LRM uses the network priority from the SRM policy to configure the Diffserv component ensure that all packets going out have their DSCP correctly set. Routers configured to support DiffServ ensure that the packets get queued according to their DSCP priorities.

The **CPU Broker qosket component** is responsible for reserving CPU cycles over a period of time for a component container. The LRM uses the minimum and maximum CPU reservation and the relative importance from the SRM policy to configure the CPU Broker component. The underlying CPU mechanisms (CPU Broker [5] and TimeSys Linux) guarantee that the container gets the minimum CPU cycles it needs. In the case of CPU contention, no more than the maximum CPU cycles are allocated to the container.

**Data Shaping Qosket Components.** Once the available CPU and network resources have been allocated across UAV streams, each stream must shape its data to use the allocated resources effectively. We assemble several data shaping qoskets that the LRM uses to accomplish this:

- **Fragmentation, Pacing, and Defragment qosket components** are combined to reduce jitter in the network by spreading the transmission of data evenly over the interval specified by its rate. The LRM configures them with the allocated bandwidth and a fragment size (the maximum transmission unit of the network is a logical choice). The fragmentation component breaks an incoming image into fixed sized fragments and the pacing component sends the fragments over the network at regular intervals. Fragmentation on the sender side is accompanied by assembly (or defragmenting) on the receiver side. The defragment component receives fragments and, once it has received all the fragments of an image, reconstructs the image.
- The **Compress qosket component** is responsible for compressing an image. The level of compression is set by the LRM as specified by the QoS Predictor.
- The **Crop qosket component** removes a specified amount of the image from a set place in the image. The amount of the image that is cropped is set by the LRM as specified by the QoS Predictor. In the current prototype, we crop from the center of the image, but could be cropped around the AOI no matter where it is in the image.
- The **Scale qosket component** reduces the size of an image. The LRM sets the amount that the image is scaled as specified by the QoS Predictor.

Assuming there are enough resources, the SRM ensures that every end-to-end image stream gets at least the minimum it needs and that the more important streams get the majority of the resources. In the cases where there are not enough resources for all the streams, the SRM ensures that the most important streams get the resources that are available. The LRM ensure that the allocated resources for each end-to-end stream are used most effectively for the UAV’s role in the system.

Figure 22 shows the design of the QoS management system using the DQME design environment. In addition to using DQME to design the QoS management architecture and algorithms, we used the Component Assembly and Deployment Modeling Language (CADML) [28] to design the assembly of the functional and QoS components in the demonstration and generate the XML CAD file used by the CIAO component.

![Figure 22: DQME Model of the QoS management design in the PCES capstone demonstration](image)
infrastructure. The component assembly of the PCES capstone demonstration is shown in Figure 23.

4.3.3 Composing Separately Developed Subsystems

The PCES capstone demonstration application combined US Air Force and US Army operations built by several different organizations, at different times. These included C2 subsystems, sensor and weapon systems, and simulated systems. A tight coupling of these systems to integrate them for the demonstration would have gone against our goals of avoiding a stovepiped system, even if it would have been possible. However, even more realistic reasons for avoiding the tight integration existed: several of the subsystems were legacy systems; many of them were large, complex systems; and they were written in different languages and hosted on different platforms.

In the integration of these, we had two primary complexities that we wanted to avoid:

- **Startup dependencies** – Because the subsystems were developed independently, they each had application specific ways of handling their inputs and outputs. When they were composed into the capstone demonstration, some functioned as clients to others, some as servers, and some functioned as both clients and servers. It was easy to fall into the trap of an accidental, but complicated imposed dependence on startup order, in which server elements of the subsystems had to be created and initialized in a particular order so that they existed before their clients looked for them.

- **Runtime coupling** – An ad hoc approach to integrating individual subsystems presented another danger, namely of making them too tightly coupled. Not only would doing so make the architecture of the composed system less modular, but it presented additional dangers for the development of the system. The capstone demonstration was large, with many interoperating pieces. During development, integration, and demonstration rehearsals, crashes of individual pieces were common. When one piece crashed, or was taken down so a source code change could be made, anything functioning as a client of the now missing piece would get transient errors or crash. In a tightly coupled system, this meant restarting the whole system, or large parts of it.

The traditional software engineering approach to solving the first problem is to enforce startup order using process and startup scripts. The second would traditionally be handled by introducing exception handling in each component, so that it caught and gracefully handled transient errors. However, both of these solutions introduce tighter coupling into the system than we were aiming for – a change to, absence of, and/or addition of, any component and subsystem would affect elements in other subsystems. Individual subsystems should be able to start and function on their own or as part of a composed system and should be able to start and evolve independently, with well-defined and limited effect on other subsystems.

To achieve this goal, we used the following middleware services to integrate the pieces of the composed capstone demonstration, as illustrated in Figure 24:
The **CORBA Notification Service**, which provides a CORBA standard publish/subscribe service that allows any IDL-defined structure to be published as an event.

The **Global Information Grid/Joint Battlespace Infosphere** (GIG/JBI), a US Air Force standard publish/subscribe information management system.

The **CORBA Real-Time Event Channel**, a CORBA standards-based middleware service that supports the real-time exchange of small messages (events).

Since each piece only has references to the middleware service endpoints and not direct references to other pieces, it is unaffected by the other pieces going down or coming up. In addition, it was much easier to add a new piece or communications path to the system. Finally, it was easier to test subsystems or groups of subsystems, since each subsystem could run independently or could be replaced by a test driver that co-existed with the real pieces.

### 4.3.4 Performance of the PCES Capstone Demonstration

Figure 25 illustrates the behavior of the PCES Capstone Demonstration from a resource management point of view over three hours of an execution. Figure 26 zooms in on the first hour and a half of network (Figure 26(a)) and CPU usage (Figure 26(b)) for each participant in the demonstration. The reallocation of resources upon role changes is apparent from the graphs. Another apparent observation from the graphs is the relative lack of jitter and control of network resource usage (Figure 26(a)) versus the higher jitter in the CPU usage (Figure 26(b)). This is due to the fact that in the Capstone Demonstration, we used DiffServ codepoints and DiffServ enabled routers for network control. However, stability problems in the underlying Timesys Linux RT operating system caused us to omit Timesys Linux RT for the Capstone Demonstration and use vanilla Linux OS priorities instead. Linux is not a real-time OS and, therefore, did not provide the control that a real-time OS does and resulted in higher relative jitter in the Capstone Demonstration. We still managed to maintain higher priorities for the critical tasks. However, in future versions, it is obviously...
preferable to use a real-time OS, which means resolving the stability issues with Timesys Linux RT or replacing it with another real-time OS.

5 Concluding Remarks

As the complexity and capabilities of DRE systems increase, driven both by increasing ability to network independent embedded systems and by the need to increase their capabilities and scope, the ability to construct them with predictable, reliable QoS must keep pace. Gone are the days when resource admission
control at one point sufficed to be called QoS. Today’s DRE systems are dynamic interoperating systems of systems, with the need for QoS management that is

- **Multi-layered**, basing low level allocations of resources and control of behavior on high level mission goals.
- **End-to-end**, matching end user QoS needs with the quality production and dissemination of information.
- **Aggregate**, mediating and managing the conflicting QoS needs across competing applications, systems, and users.
- **Dynamic**, adjusting QoS provision on the fly as situations, conditions, and needs change.

There is little doubt that intelligent engineers could produce a system with QoS management that fulfills each of these characteristics for a specific system. However, our goals in the work described in this chapter were more ambitious. We did not set out simply to develop QoS management for a single instance of a specific system, but to develop tools and techniques that enable QoS management to be developed in many systems repeatedly, so that it is well designed, reusable, and maintainable. We based this work in middleware, because QoS management falls in that space where the applications interact with the platforms and environments in which they are deployed. We also built upon and extended software engineering practices that support these goals:

- **Aspect-oriented programming and separation of concerns**, to support the separation of programming application code (which is the purview of a domain expert) and QoS code (which is the purview of a systems engineer).
- **Components and composition**, to support the encapsulation of QoS management code into reusable bundles and the construction of new systems by composing, specializing, and configuring existing components.
- **Middleware services**, to support the loose integration of whole subsystems, enabling large DRE systems of systems to be constructed from existing DRE systems.

Unlike many previous research endeavors in these areas, the work described in this chapter has gone beyond the laboratory setting. The three case studies described in this chapter involved actual military and industrial organizations, systems, problems, and live flight demonstrations. Applying our QoS management middleware research to these case studies has validated our research results and their utility to increasing the capabilities of the complex DRE systems emerging in real-world domains.

6 References


[28] T. Lu, CADML (Component Assembly and Deployment Modeling Language), Web Site: http://www.dre.vanderbilt.edu/~lu/CADML/