AIDING THE DEPLOYMENT AND CONFIGURATION OF COMPONENT MIDDLEWARE IN DISTRIBURED, REAL-TIME AND EMBEDDED SYSTEMS

by

Stoyan G. Paunov

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2006

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

ACKNOWLEDGEMENTS

I would like to thank Dr. Douglas C. Schmidt, who is my advisor, for his insightful ideas, encouragement and guidance. I further want to thank Dr. Schmidt for his major contribution to my professional growth and for my newly acquired insights into our field as it stands today and where it is likely to go in the future.

I would also like to thank my mother Zoya, my father Georgi and my sister Elena for their support and belief in me through the years. Finally, I would like to thank Vanderbilt University and the Institute of Software Integrated Systems (ISIS) for providing the setting for my work and for supporting my efforts morally and financially.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS ... ii

LIST OF FIGURES ... v

ABBREVIATIONS ... vi

Chapter

I INTRODUCTION ... 1

1.1 Component Middleware ... 1
1.2 Real-time CORBA Component Model (RT CCM): ... 2
1.3 Thesis Focus: Filling some deployment and configuration gaps ... 4
1.4 Case Study: The DARPA Multi-Layer Resource Manager Project ... 5

II A COMPONENT REPOSITORY ... 8

2.1 Problem Description ... 8
2.2 Context: Use of RepoMan in the MLRM Architecture ... 9
2.3 Design of RepoMan ... 10
2.3.1 Structure of RepoMan ... 11
2.3.2 Functionality of RepoMan ... 13
2.4 Resolving RepoMan Design Challenges ... 15
Challenge 1: Effectively Integrating CORBA with an HTTP Server ... 15
Challenge 2: Lowering the Cost of Data Movement and XML Parsing ... 17
Challenge 3: Organizing and Managing Data ... 19
Challenge 4: Managing the Complexity of *PackageConfiguration* Elements 21
Challenge 5: Scalable Implementation and Lightweight Synchronization ... 23
2.5 Lessons Learned ... 24

III CONFIGURING COMPONENT MIDDLEWARE FOR QoS ... 26

3.1 QoS Configuration and evaluation challenges ... 26
3.2 QoS Configuration Challenges in the context of the MLRM ... 28
3.3 Overview of QoSPML ... 30
3.3.1 Motivation ... 30

3.3.2 Structure of QoSPML 32
3.3.3 Functionality of QoSPML 32
3.4 Resolving ARMS MLRM Challenges with QoSPML 36
3.4.1 Configuring Infrastructure and Application Components for QoS 36
3.4.2 Meeting the QoS Needs of the Various MLRM Subsystems 37
3.4.3 Using MDD Tools to Generate XML Metadata 38
3.4.4 Managing and Refining the System Configuration Space 39
3.5 Benefits of MDD tools to the component-based ARMS applications 40

APPENDIX A 43

REFERENCES 44

LIST OF FIGURES

Page

Figure 1: Component-based Architecture of the ARMS MLRM 6

Figure 2: The RepoMan Architecture 11

Figure 3: RepoMan in Action 16

Figure 4: Evaluating the QoS of a Shipboard Computing Enterprise DRE System......... 27

Figure 5: GME Metamodel of QoSPML 33

Figure 6: QoS Configuration Snippet of a Model and its Interpretation 35

ABBREVIATIONS

API – Application Programming Interface

ARMS – Adaptive and Reflective Management System

CIAO – Component Integrated ACE ORB

CCM – CORBA Component Model

D&C – Deployment and Configuration

DOC – Distributed Object Computing

DRE – Distributed, Real-time and Embedded

DSML – Domain Specific Modeling Language

MDD – Model-Driven Development

MLRM – Multi-Layer Resource Manager

QoS – Quality of Service

QoSPML – Quality of Service Policy Modeling Language

TSCE – Total Ship Computing Environment

CHAPTER I

INTRODUCTION

1.1 Component Middleware

Historically distributed systems were developed atop of operating systems and protocols. These traditional methods were however replaced by stacks of middleware technologies – a shift largely triggered by the necessity to achieve systematic reuse of existing architectural and design principles in order to avoid reinventing and reimplementing core distributed infrastructure capabilities and services and decrease development time. The most recent wave of middleware technologies offers higher-level abstractions, such as component models (for example the CORBA Component Model (CCM) and J2EE), web services (such as SOAP), and model-driven middleware (e.g. Cadena and CoSMIC) [26].

Component middleware aims to address deficiencies of previous middleware technologies by (1) clearly defining the unit of reusability in the form of a component, which includes supported interfaces and functional requirements and capabilities (2) offering standard application assembly mechanisms, (3) providing standard deployment, life-cycle management and configuration mechanisms in the context of a component application server, (4) integrating standard services into the infrastructure and (5) enabling the dynamic evolution and upgrade of deployed components.

In large-scale distributed real-time and embedded (DRE) systems, such as shipboard computing environments, inventory tracking systems, and intelligence, surveillance and

reconnaissance systems, component middleware can make the software more flexible by separating application functionality from system lifecycle activities, such as component deployment and configuration [2]. These DRE systems have stringent quality of service (QoS) requirements such as the low latency and jitter expected in conventional real-time and embedded systems, as well as the high throughput, scalability, and reliability expected in conventional enterprise distributed systems. Ordinary component middleware technologies, such as J2EE and .NET, do not provide real-time QoS support and are therefore not well-suited for the task of developing DRE systems. QoS-enabled middleware, such as the *Component-Integrated ACE ORB* (CIAO) [23], Qedo, and PRiSm, have emerged to address these limitations by marrying the flexibility of component middleware with the predictability of Real-time CORBA [2]. All of them are implementations of the CORBA Component Model (CCM) [14] supporting Real-time CORBA [15].

1.2 Real-time CORBA Component Model (RT CCM):

The general CCM contains a number of standard features such as (1) a *component server*, which is a generic server process for hosting component implementations and enabling them to access common middleware services and runtime policies, (2) a *component implementation framework*, which automates the implementation of many component features, (3) *component packaging tools*, which compose implementation and configuration artifacts into deployable assemblies, and (4) *component deployment and configuration tools*, which automate the deployment and configuration of applications

[25]. Real-time CCM combines these mechanisms with Real-time CORBA mechanisms such, as thread pools and priority preservation policies, to enable the configuration of application components in DRE systems for end-to-end QoS.

Real-time CCM implementations, such as CIAO, provide an effective way to organize software into loosely-coupled reusable components and export their functionality by means of one or more *interfaces*. An interface is an implementation-independent contract specifying the operations that can be performed on a reusable unit of code, along with their input/output parameters and return type. Components can form relationships with other components by means of standard interfaces named *ports*, including (1) *facets* that expose a piece of functionality that the component offers, (2) *receptacles* that indicate dependencies on functionality provided via facets by other components, and (3) *event sources and sinks* that enable publish/subscribe event-driven communication between components.

Real-time CCM also provides mechanisms for aggregating related *monolithic* components into component *assemblies* by connecting together their ports. Component implementations are bundled into *packages* that contain (1) binary implementations of the encapsulated components, possibly for multiple programming languages, operating systems, and hardware platforms, and (2) XML metadata that describes the contents of the package, including the interfaces, requirements and capabilities of individual components and how they are connected to form an assembly. Packages in Real-time CCM are created by a *component packager*, which is an actor that wraps multiple

implementations of the same component interface into a component package and ensures its consistency.

1.3 Thesis Focus: Filling some deployment and configuration gaps

Although component middleware technologies solve many of the problems associated with previous generations of inflexible, monolithic, functionally-designed, and "stove-piped" enterprise DRE systems, they also introduce new challenges associated with the higher flexibility and configurability of the system, the manageability of the large number of deployment and configuration artifacts and the evolution of the system in response to improved understanding of the domain or feedback from testing and emulation of end-to-end QoS performance. The rest of this document discusses some of these challenges and shows how they are solved in the context of the DARPA Adaptive and Reflective Management System's (ARMS) *Multi-Layer Resource Manager* (MLRM) [20] project. The MLRM architecture is discussed next. Chapter II then discusses how component repositories can be used to address many of the newly arisen deployment and configuration complexities. Chapter III concentrates on some of the complexities associated with configuring component middleware for QoS and shows how *Model-Driven Development* (MDD) technologies can be applied to mitigate the problem.

1.4 Case Study: The DARPA Multi-Layer Resource Manager Project

The work described in this thesis was motivated by our experience with the DARPA ARMS MLRM framework for naval shipboard computing systems and the challenges encountered while developing and evaluating it.

The MLRM services developed in ARMS are designed to support *total ship computing environments* (TSCEs), which form the basis for next-generation naval programs. A TSCE is a coordinated grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations. To make TSCE an effective platform requires coordinated MLRM services that can support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization.

The ARMS MLRM integrates multiple resource management and control algorithms based on the CIAO [23] Lightweight CORBA Component Model (CCM) [14] and Real-time CORBA [15] mechanisms for (re)deploying and (re)configuring application components in DRE systems. As shown in Figure 1, the ARMS MLRM top *domain layer* contains infrastructure components that interact with the mission manager of TSCE by receiving command and policy inputs and passing them to the *resource pool layer*. The resource pool layer is an abstraction for a set of computer nodes managed by a *pool manager*. The pool manager is an infrastructure component that interacts with the *resource allocator* in the resource pool layer to run algorithms that deploy application components to various nodes within a resource pool. The actual computing resources reside in the third layer called the *resource layer*, which has infrastructure components

called *node provisioners* that receive commands to spawn applications in every node from a pool manager. The *application string manager* is an infrastructure component that controls the resource utilization for a group of applications through the node provisioners. The ARMS MLRM services have hundreds of different types and instances of infrastructure components written in ~300,000 lines of C++ code and residing in ~750 files developed by different teams at different locations.

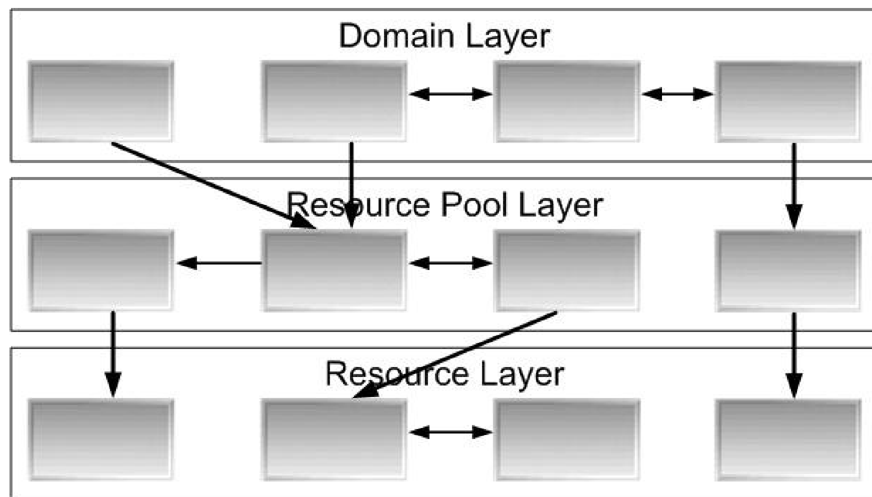

Figure 1: Component-based Architecture of the ARMS MLRM

The component-based MLRM infrastructure for a TSCE is designed to support the highly heterogeneous environment in which long-lived shipboard computing systems operate. For example, the TSCE that provides the operational context for the ARMS MLRM services is designed to support different versions of (1) component middleware, such as CIAO and OpenCCM, (2) general-purpose operating systems, such as Linux and Solaris, (3) real-time operating systems, such as VxWorks and LynxOS, (4) hardware

chipsets, such as x86, PowerPC, and SPARC processors, (5) a wide range of high-speed wired interconnects, such as Gigabit Ethernet and VME backplanes, and (6) different transport protocols, such as TCP/IP and SCTP [16], [17].

CHAPTER II

A COMPONENT REPOSITORY

2.1 Problem Description

Although component-based enterprise DRE systems help address the problems with prior generations of systems, they introduce a number of new challenges, such as the need to shield component behavior, deployment, and configuration logic from the complexities of heterogeneous hardware/software environments and runtime failure recovery. Due to these heterogeneity and reliability requirements, enterprise DRE systems often need to defer the installation of software onto target nodes until late in the life-cycle, e.g., at startup or run-time. Moreover, to cope with the continually evolving environments in which they run, these systems need mechanisms, such as online software upgrades and component reconfiguration/redeployment services, to provide the right implementation under the right circumstances.

A promising way to address these new challenges is to create *component repository managers* that (1) keep track of software implementation artifacts and configuration metadata in heterogeneous environments and (2) facilitate the online upgrades, reconfiguration and redeployment of components. Developing repository managers for enterprise DRE systems is however hard. Key challenges include the need to support

cross-platform portability, ensure efficiency, responsiveness and scalability, and enable dynamic updates within time constraints.

This chapter discusses the design and implementation of *RepoMan*, which is an implementation of the OMG CCM Repository Manager specification [13] tailored to the needs of enterprise DRE systems. In particular, RepoMan optimizes its CPU and I/O usage to provide fast/predictable access to component data for enterprise DRE systems with a range of QoS requirements. The RepoMan C++ framework contains ~5,300 lines of code in over 45 classes. It has been bundled with the CIAO open-source implementation of Real-time CCM [17].

2.2 Context: Use of RepoMan in the MLRM Architecture

The scale, complexity, and longevity of TSCEs necessitates that their components be organized and accessed in a common and standard manner. RepoMan provides this functionality for ARMS and helps ensure the continuous availability of components and their associated metadata throughout the system lifetime. For example, RepoMan is used during initial system deployment when MLRM resource allocators instruct node provisioners to spawn a specific set of applications. The node provisioners contact RepoMan to download the component implementations they need to deploy via CIAO's implementation of the OMG D&C specification [2], which standardizes many aspects of deployment and configuration for component-based distributed systems, including component configuration, component assembly, component packaging, package configuration/deployment, and target domain resource management. RepoMan is also

used at runtime to update component implementations dynamically, e.g., in response to battle damage or to handle changing workload levels.

A particularly important function of the resource allocation and control algorithms in the ARMS MLRM is the (re)deployment and (re)configuration of components based on their operational context. For example, the TSCE can switch from crew entertainment mode to ship defense mode, which necessitates updating and/or migrating many computing services. RepoMan provides mechanisms to retrieve the configuration data associated with specific component implementations and enables the dynamic updating of various configuration parameters. Resource allocators and node provisoners communicate with RepoMan to choose the best available implementations and to ensure that these implementations conform to the characteristics of each node's hardware, OS, middleware, and programming language(s), which can be highly diverse.

2.3 Design of RepoMan

RepoMan is designed to enable software developers and enterprise DRE systems to (1) organize various offline and online configurations of component packages (which include component implementations and their associated metadata, known as *PackageConfiguration,* that describe the contents of a component package by encapsulating the interface definitions of the components, their requirements and capabilities, their implementation descriptions, and their dependencies on other implementation artifacts), (2) resolve references to component implementations at deployment time, (3) retrieve metadata information to configure the components properly, (4) reconfigure the compo-

nent implementations within a package by updating their associated metadata, and (5) dynamically update components at run-time. This section describes how the structure and functionality of RepoMan supports these capabilities.

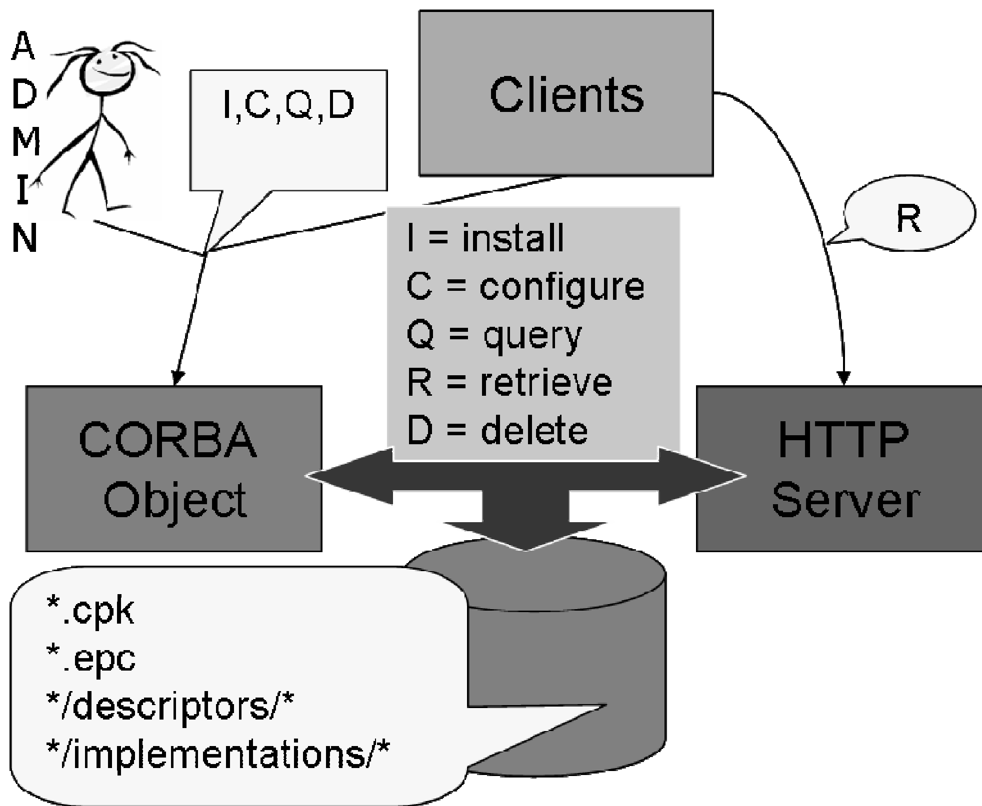

Figure 2: The RepoMan Architecture

2.3.1 Structure of RepoMan

Figure 2 illustrates the RepoMan architecture, which consists of a CORBA object encapsulating ~15 classes implementing different aspects of its functionality and a collocated HTTP server encapsulating over 30 classes. The CORBA object supports a

standard set of operations (shown as abbreviations in Figure 2) that enable applications and other CCM services to manipulate data in the repository, retrieve configuration metadata in the form of *PackageConfigurations*, and update component configurations. The collocated HTTP server enables the retrieval of implementation artifacts, which typically consist of dynamic link libraries (DLLs).

One way to design a component repository would be to just use an HTTP server to provide access to component packages. Although this approach is simple to implement, it does not scale well because (1) it requires clients to download entire packages to obtain their contents, which is inefficient, and (2) each client would need explicit knowledge of how to parse the metadata in a component package, which would needlessly complicate client code. RepoMan alleviates these drawbacks by serving as a mediator [4] that handles package content organization and metadata manipulation to provide a standard way of storing, locating, and querying the available component packages and the relationships among them. By centralizing *PackageConfiguration* parsing, RepoMan also simplifies client code. Section 2.3.4 describes an optimization technique that shows how metadata parsing centralization allows RepoMan to parse metadata only once per component package. In contrast, using a simple HTTP server would require parsing the metadata many times, i.e., once at every client instance location, so RepoMan's design is much more efficient and scalable.

RepoMan helps minimize unnecessary CPU and network processing by using *PackageConfigurations* as an intermediary step between clients and the HTTP server. This design helps developers and administrators determine if an implementation meets

their requirements before downloading the actual binaries. For example, if a client is unsure which implementation is best suited to its needs, it can (1) retrieve the *PackageConfiguration* metadata that describes the properties of a specific component, (2) analyze that metadata to determine which implementation is appropriate, and (3) then download just the desired component implementation(s). This capability is particularly useful in enterprise DRE systems, such as TSCEs, where online upgrades change the set of available components during the lifetime of the system.

2.3.2 Functionality of RepoMan

The CCM Repository Manager maintains a collection of *PackageConfiguration* elements, each named with a universally unique identifier (UUID). The descriptive power of *PackageConfigutations* enhances RepoMan's flexibility, e.g., by encapsulating the location of artifacts that implement a component. This encapsulation allows RepoMan to act as a component discovery service, thereby alleviating the need for client applications to hard-code information about component implementation locations. It also provides a standard way to access components. RepoMan provides the following operations that can be invoked by clients:

Installation. Developers or administrative applications can install a component package under a particular name, e.g., "NodeProvisioner." The `installPackage()` operation installs the package either from a specified location on the local disk or from a remote location accessible via HTTP. The metadata in the package is parsed and the encapsulated *PackageConfiguration* is associated with the installation name. Rather than

installing a package directly, a *PackageConfiguration* can also be installed via the **`createPackage()`** operation, where the installed *PackageConfiguration* refers to an external package whose location is interpreted via a base location. RepoMan is responsible for resolving all references to external packages. Both operations ensure the uniqueness of the installation names, raising exceptions if this precondition is violated.

Deletion. The inverse of the install operations is the **`deletePackage()`** operation, which is used to remove component packages from the repository. If the specified name does not exist in the repository an exception is raised.

Retrieving configuration data. Available *PackageConfigurations* can be retrieved by name or by UUID at any time. If the *PackageConfiguration* corresponding to the supplied name is not currently in the repository, RepoMan raises an exception.

Querying the contents. If a client has no prior knowledge of the existence of any specific installation, it can retrieve all available ones by name or by type. Every component conforms to a specific interface described by *Component Interface Descriptors*, which are identified by their UUIDs and specify the operations that can be performed on the component, along with their input/output parameters and return type. RepoMan can return all installation names that implement a specific type of interface. Clients can also request a list of all component types an instance of RepoMan is managing.

Retrieving implementations. The CCM Repository Manager standard specifies that component implementations are retrieved via HTTP. Upon installation, RepoMan

updates the *PackageConfiguration* describing the package to reflect the correct locations of implementation artifacts so that they are accessible via the collocated HTTP server.

2.4 Resolving RepoMan Design Challenges

Although the CCM specification defines the interface and the functionality of the Repository Manager service, it does not prescribe any design details. We were therefore faced with a number of design challenges when implementing RepoMan. This section describes the key design challenges we encountered, presents our solutions, and outlines how we applied these solutions to the TSCE applications supported by the ARMS MLRM.

Challenge 1: Effectively Integrating CORBA with an HTTP Server

Context. As described in Section 2.2.1 and shown in Figure 3, RepoMan's architecture has (1) a CORBA object that installs/removes packages in the repository and provides component configuration data and (2) an HTTP server that provides access to the implementation artifacts.

Problem → Effectively integrating CORBA with an HTTP server. One approach to integrate CORBA and an HTTP server would enable them to communicate via a shared memory segment, but this would tightly couple the HTTP server with the CORBA implementation and preclude the use of other web servers. Another approach would be to extend the interface of the RepoMan to support HTTP, but this would require

implementing HTTP as a pluggable protocol under CORBA, which is complicated, non-portable, and also precludes the use of other ORBs and web servers.

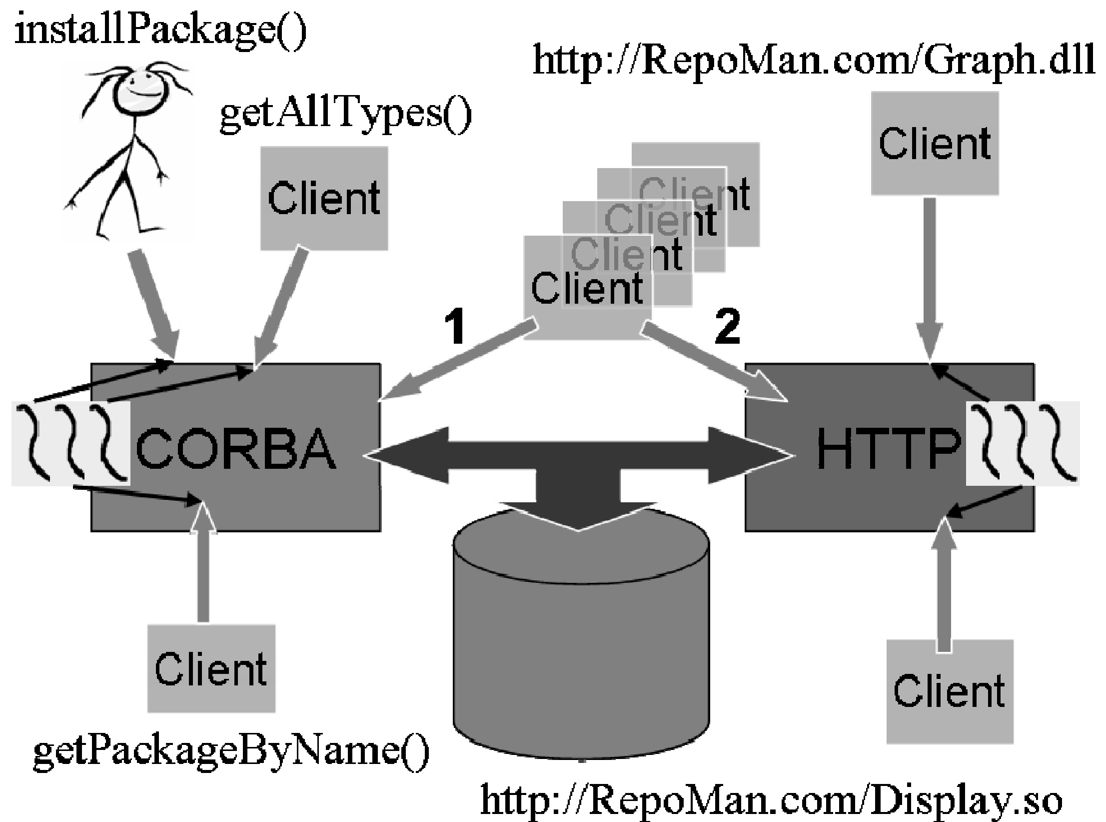

Figure 3: RepoMan in Action

Solution → Loose coupling between the CORBA object and the HTTP server. RepoMan's CORBA object and HTTP server are collocated on the same host, but have no explicit knowledge of each other and share no internal state information. Instead, they use a loosely coupled relationship that shares a common filesystem. The document root of the HTTP server points to the directory where the RepoMan caches copies of

component packages. Packages are also uncompressed in that directory at installation to avoid complicating the logic of the HTTP server with request filters (httpd.apache.org) and to minimize data movement, as discussed in Challenge 2.

Challenge 3 explains how we preserve the consistency within the package hierarchy. RepoMan updates the component metadata at runtime, so the locations of implementation artifacts point to the HTTP server. Clients can therefore first retrieve and process the metadata from RepoMan and then obtain the right implementation artifacts from the HTTP server, as shown in the center of Figure 3.

RepoMan's approach is flexible and enables the use of multiple web server implementations. By default, RepoMan uses the JAWS web server [8] since it is bundled with the CIAO release. We can easily replace JAWS with the ubiquitous Apache web server, however, without affecting the CORBA portion of RepoMan.

Applying the solution to the ARMS case study. When the MLRM's node provisioners receive a command to spawn a specific component they match the requester's operational needs (e.g., operating system and hardware platform) with the available component implementations available from RepoMan. Once a node provisioner finds a match, it uses the address stored in the location field of the corresponding *PackageConfiguration* to request the implementation from RepoMan's HTTP server, which sends the corresponding artifact to the node provisioner enabling it to perform the deployment.

Challenge 2: Lowering the Cost of Data Movement and XML Parsing

Context. Component packages in CCM are files archived with the ZIP algorithm [3], which conform to a specific structure, and have a *.cpk extension. The most common

RepoMan operation requested by clients – **getPackageByName**(), as shown in the bottom left corner of Figure 3 – is used to return a *PackageConfiguration*. The information conveyed by the *PackageConfiguration* is initially only present in the XML metadata descriptors enclosed in the package. It is therefore necessary for RepoMan to parse these descriptor files to populate the in-memory *PackageConfiguration* before its contents can be marshaled and sent to the clients. RepoMan uses the XERCES XML parsing library since it is robust and performs comprehensive schema validations.

Problem → Lowering the cost of data movement and XML parsing. Manipulating component packages requires a considerable amount of processing to move data to/from disk and perform XML parsing. For example, manipulating CCM metadata in a component package involves loading the zip'd package contents into memory, uncompressing them, and then writing them back to disk again because XERCES cannot parse XML from memory directly. XERCES will then parse the uncompressed files to extract the relevant information (e.g., the interface type supported by the component or the names of the implementation artifacts), and load it into an equivalent C++ data structure that RepoMan uses to manipulate the data in memory and to transport it to clients across the network.

Solution → Minimizing data movement and XML parsing to improve CPU and I/O usage. Uncompressing packages (see Challenge 1) avoids on-access decompression and unnecessary data movement. To further decrease data movement and to minimize XML metadata parsing, the RepoMan employs the Memento pattern [4], which externalizes and records the internal state of an object at an important stage of its lifecycle to enable

its later restoration. We used the standard OMG Common Data Representation (CDR) format (which is a portable data (de)marshaling format defined by the CORBA specification [14]) to externalize the contents of the in-memory *PackageConfiguration* element at installation time after XERCES had validated the correctness of the parsed data and the *PackageConfiguration* had been populated. The result is illustrated in Figure 2. This optimization eliminates any subsequent XML parsing and enables RepoMan to load *PackageConfigurations* on-demand and forward them to clients, thereby minimizing CPU and I/O processing considerably and significantly improving the response time of package lookup operations.

Applying the solution to the ARMS case study. The operational context of a TSCE evolves continuously, e.g., it needs to satisfy changing mission requirements and adapt to transient overload and permanent battle damage. Such changes provoke a reaction in the control algorithms that drive the dynamic update or the partial or complete redeployment of the system. Minimizing data movement and XML parsing overhead (1) improves the responsiveness of the RepoMan and allows it to collaborate faster with clients (such as the ARMS MLRM and TSCE applications) and (2) helps reduce the costs associated with redeploying and updating the system, thereby enabling more CPU and I/O processing to be spent performing mission tasks and meeting system deadlines.

Challenge 3: Organizing and Managing Data

Context. The package location specified at installation time is either a path in the local filesystem or an HTTP URL pointing to a remote file. As discussed in Challenge 2, when

RepoMan installs component packages in the repository it caches them locally to minimize subsequent access time and to ensure their availability.

Problem → Organizing and managing package data. In order to function properly, RepoMan requires that the files that it manipulates (i.e., the component packages, the implementation artifacts, and the externalized *PackageConfigurations*) remain consistent across accesses. It was therefore necessary to provide the right degree of separation among files associated with different installations. It was also necessary to enable access, traversal, and clean-up of installed files. The lack of standard file system access application programming interfaces (APIs) among different operating systems makes this hard, however, because we need to ensure that RepoMan's code remains portable across OS platforms.

Solution → Ensure consistency by basing file system organization on the operational semantics. RepoMan structures the package organization hierarchy by leveraging the fact that installation names are unique within the repository. When a package is installed, RepoMan caches its contents in accordance with the configured "install path" and names the cached version based on the installation string and not the original filename. As discussed in Challenge 1 and Challenge 2, RepoMan decompresses component packages and caches them locally at installation time in a directory whose name also corresponds to the installation name. This design separates different packages and avoids clashes among files enclosed in the packages that have equivalent names. Due to the uniqueness of installation names which it ensures, RepoMan can guarantee that none of the local data will be overwritten accidentally by future installations.

We avoid the problem of non-standard file system access APIs by replicating the layout of the component package on disk (Figure 2). This design allows RepoMan to use the package layout rather than a filesystem API to guide it through subsequent clean-up of packages upon deletion. It also ensures the portability of RepoMan's file system access and traversal code.

Applying the solution to the ARMS case study. As discussed in Section 1.4, the ARMS MLRM is designed to support different general-purpose and real-time operation systems running atop diverse hardware. By using the internal package layout to guide RepoMan through its access, traversal, and clean-up operations, we avoid using any non-portable file system APIs and ensure that RepoMan can be compiled and deployed in any ARMS MLRM target environment.

Challenge 4: Managing the Complexity of *PackageConfiguration* Elements

Context. A key task of RepoMan is to update the location field of the implementation artifacts so that they can be retrieved via the collocated HTTP server, as depicted by Figure 3. This task requires RepoMan to navigate through the *PackageConfiguration* element all the way down to the implementation artifacts, which are "leaves in the implementation tree" encapsulated by the *PackageConfiguration*. The structure of the implementation tree is very flexible and allows the recursive specification of component assemblies by composing them from interconnected smaller monolithic and/or assembly-based components.

Problem → Managing the complexity of PackageConfiguration elements. The *PackageConfiguration* element encapsulates a description of the deployment

requirements for the component, the properties used to configure the component, as well as the recursive description of the component implementation tree that may consist of multiple monolithic and assembly-based components along with the description of their interconnection. The *PackageConfiguration* is therefore one of the most complex elements in the OMG CCM specification. For example, in the case of assembly-based components the field disclosing the location of any one of the artifacts implementing it is at least 11 levels deep! Updating the locations of the implementation artifacts can therefore be tedious and error-prone to program using a naïve design.

Solution → Use the Visitor pattern to manage the complexity of the PackageConfiguration. To manage the complexity of traversing and updating *PackageConfigurations*, we used the Visitor pattern [4], which separates the structure of a collection of objects from the algorithms applied to the objects. The Visitor pattern helps manipulate complicated *PackageConfiguration* hierarchies because it separates the parsing and control logic for every node in the hierarchy into separate methods, which allow RepoMan to perform its tasks one step at a time. The Visitor pattern is well suited for the recursive nature of the component implementation hierarchies targeted by the location updating procedure.

Applying the solution to the ARMS case study. The Visitor-based approach we used helps ensure that RepoMan correctly updates the location of all underlying implementation artifacts. This design is important for the MLRM because components in the same package usually belong to the same application and not updating the location field of a particular component can cause a deployment failure for the TSCE.

Challenge 5: Scalable Implementation and Lightweight Synchronization

Context. As Figure 3 illustrates, the RepoMan can be accessed by many clients in and enterprise DRE system, often under strenuous conditions, such as during the TSCE recovery process after nodes in a data center have failed.

Problem → Providing a scalable implementation and ensuring correct synchronization and low response time. Minimizing the response time of RepoMan is hard because it can receive different requests from multiple clients simultaneously. Although multi-threading is commonly used to improve application response time, it also yields several design problems, such as selecting the appropriate concurrency model, e.g., thread-per-request vs. thread pool. Although a thread-per-request model can potentially adapt better to increasing demand, it can also exhaust the system resources in response to bursty client requests. While a thread pool model can be used instead to prevent the latter scenario, this model is not as adaptive. Another design problem involves selecting the synchronization mechanisms to prevent race conditions when multiple threads are accessing shared resources. Since synchronization mechanisms incur mutual exclusion overhead and can severely limit the opportunity for concurrent operation of multiple threads due to their sequential processing enforcement nature, their use should be limited only where they are absolutely needed.

Solution → Use a variable-size thread pool with lightweight synchronization. RepoMan uses a thread pool to prevent bursty clients from depleting system resources. The size of RepoMan's thread pool is configurable at startup since the number of spawned threads depends on the characteristics of the target host on which it is deployed.

RepoMan uses three hash tables to store its internal state information, such as associations of installation names with package contents on disk. We avoid synchronizing each operation performed by RepoMan in its entirety by only synchronizing access to these hash tables. This lightweight synchronization design is more efficient than the alternatives (such as the Monitor Object or Active Object patterns [21]) by limiting the concurrent access to a fraction of the code and allowing multiple threads to handle the same type of requests from different clients concurrently.

Applying the solution to the ARMS case study. RepoMan is a key part of the (re)deployment and (re)configuration activities performed by the ARMS MLRM. Using a multi-threading and lightweight synchronization design along with the optimizations discussed in Challenge 2, helped us minimize RepoMan's response time, thereby contributing to the minimization of the overall cost of redeployment, reconfiguration, and component update activities.

2.5 Lessons Learned

We discussed the design challenges faced when developing and applying RepoMan to a shipboard computing enterprise DRE system and showed how our solutions help resolve these challenges. The following are lessons learned during our work on RepoMan and its application to the ARMS Multi-Layer Resource Manager (MLRM):

- Building enterprise DRE systems whose operational semantics change frequently necessitates the dynamic update of components and requires a component repository

to enable the automated (re)deployment and (re)configuration of heterogeneous components throughout the system.

- The CCM Repository Manager specification strikes an effective balance between flexibility and efficiency by keeping client code considerably simpler and supporting dynamic updates and system (re)deployment and (re)configuration.
- Applying software patterns to RepoMan helped ensure that its design uses the best practices associated with solving some recurring problems and leveraged the experience of experienced developers. Patterns applied to RepoMan include Iterator, Memento, Null object, and Visitor in the COBRA object and Bridge, Service Configurator, Singleton, Strategy, Wrapper Facade in the HTTP server.
- Amortizing certain costs over lifetime of RepoMan helped to improve its performance. Although externalizing the *PackageConfiguration* slows down the installation, it enabled us to optimize the performance over the lifetime of the system since subsequent retrieval operations are much more frequent than initial installation operations.

The implementation of RepoMan is open-source and can be downloaded along with the CIAO Real-time CCM middleware.

CHAPTER III

CONFIGURING COMPONENT MIDDLEWARE FOR QoS

3.1 QoS Configuration and evaluation challenges

A particularly vexing problem facing researchers and developers of large and layered enterprise DRE systems, such as major defense, aerospace, and commercial programs, is that the inadequacies of system architectures may not be ascertained until years into development. At the heart of this problem is the *serialized phasing* of layered system development, in which the application components are not created until *after* their underlying system infrastructure components. A side effect of serialized phasing is that design flaws that affect system QoS are not discovered until late in the lifecycle because the implementations, configurations, and deployments of infrastructure components are often not tested adequately under realistic workloads.

[22] describes an interesting component workload emulation approach which can be used to exercise the infrastructure middleware much before application components are complete. We used this technology in the context of the ARMS MLRM to conduct "what if" scenario analysis in order to figure out how well the implementations, configurations, and deployments of infrastructure components will satisfy key system QoS properties, such as the maximum number of clients the system can handle before it saturates and the effects of average and worst-case response time for various workloads. Figure 4 shows

the stages in the component workload emulation approach. The feedback gained in the process is used as basis for system reconfiguration and improvement.

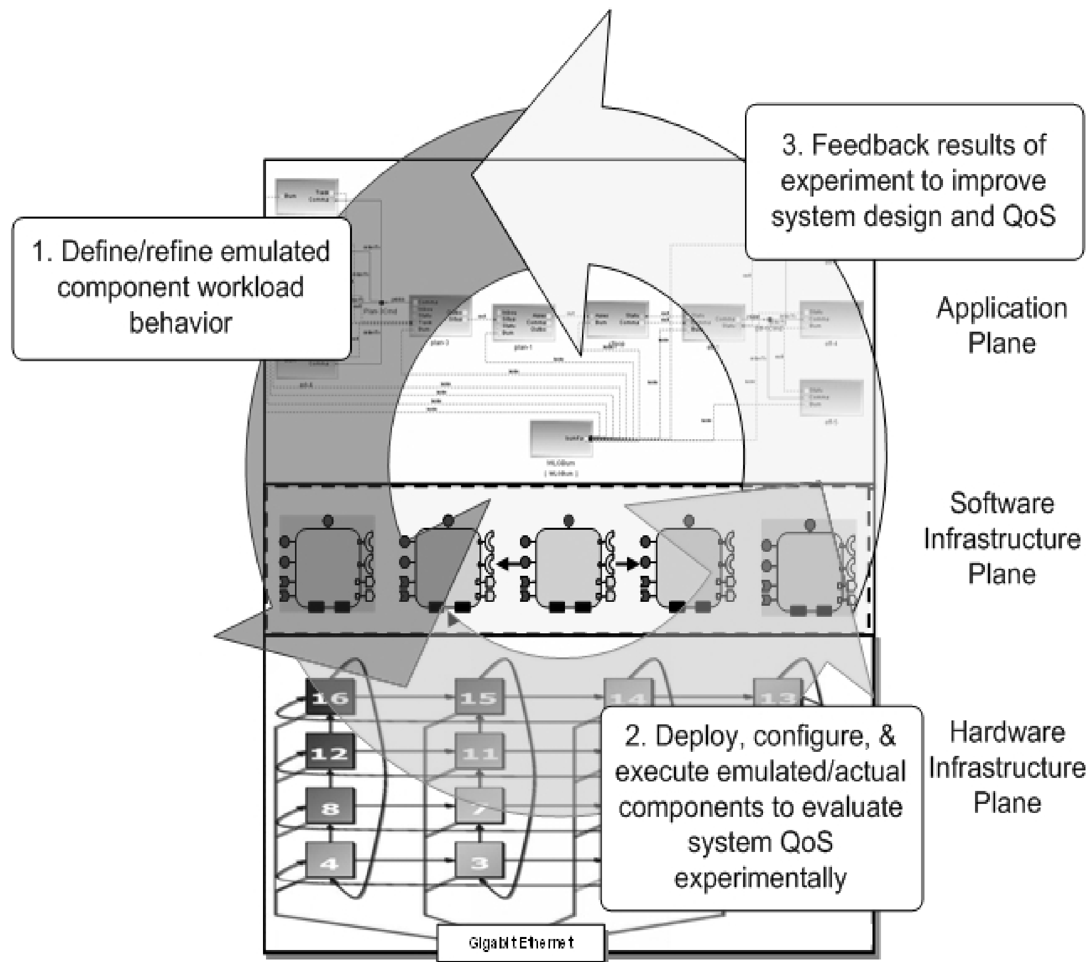

Figure 4: Evaluating the QoS of a Shipboard Computing Enterprise DRE System

While evaluating and reconfiguring the QoS characteristics of the ARMS MLRM services we encountered a number of challenges which are discussed in the following section. To address these problems we leveraged some model-driven development

(MDD) methodologies [10] and created a higher level *Quality of Service Policy Modeling Language* (QoSPML) in order to raise the level of abstraction and shield the (re)configuration developers from the accidental complexities associated with component middleware QoS provisioning [16].

3.2 QoS Configuration Challenges in the context of the MLRM

We encountered the following challenges while developing, evaluating and configuring the ARMS MLRM services to meet their QoS requirements.

Challenge 1: Using standard Real-time CORBA APIs to configure the QoS of ARMS components. One way to ensure that ARMS application and MLRM infrastructure components exhibit the necessary QoS properties is to tightly couple the necessary QoS mechanisms into them *imperatively*. While this approach is common, it requires that developers be intimately familiar with Real-time CORBA to handle its accidental complexities. Moreover, hand-coding QoS properties into components imperatively can yield convoluted and inflexible implementations that are hard to evolve.

Challenge 2: Ensuring the right granularity of QoS. The ARMS application and infrastructure components have diverse characteristics and QoS requirements including, but not limited to, high throughput of continuously refreshed data, hard real-time deadlines associated with periodic processing, well-defined computational paths traversing multiple components, soft real-time processing of many tasks, and operator display and control requirements. Specifying the right granularity of QoS for these components imperatively using Real-time CORBA APIs is hard.

Challenge 3: Managing large scale system configurations. In enterprise DRE systems like ARMS with many components, manually tracking every configuration for every component and assembly of components is hard. Hand-coding QoS properties into component implementations provides a way to track component configurations, but makes it hard for developers to review component specifications quickly. Even worse, if testing and benchmarking yields weak points in the system design, or functional requirements change, developers must manually read the code, find all relevant code snippets, and update each accordingly to reconfigure the necessary components, which is tedious and error-prone.

Challenge 4: Using metadata to configure components for QoS and to define behavioral components. Many middleware platforms, such as EJB, CCM, and .NET, have chosen XML as their configuration language since it enables different (1) application developers to create interoperable subsystems and (2) middleware developers to evolve different layers of their frameworks independently. Although XML is expressive, it is hard to manually read and write due to its accidental complexities. For example, although its elements are organized in a hierarchical form specified by the schema to which they conform, XML documents have a flat structure, are highly verbose, and lack intuitive relationships to the domain they represent. Evolving and debugging XML code manually is therefore extremely cumbersome, which makes it hard to reuse XML-based configurations.

Challenge 5: Refining system QoS properties. Enterprise DRE systems inevitably evolve due to changing functional requirements and specifications, deeper understanding

of the domain, or hardware/software platform refresh. As a result, the associated QoS properties defined for a particular version of the system must also evolve. Hand-coding QoS properties therefore creates systems that scale poorly and fail to evolve rapidly to reflect new requirements and specifications.

The next section shows how we developed and applied a MDD tool to address these challenges.

3.3 Overview of QoSPML

Although component-based DRE systems are more flexible and easier to develop, a new level of complexities has surfaced, such as the automatic configuration of application and infrastructure QoS policies. A promising way to address these complexities is to use MDD tools to create *Domain Specific Modeling Languages* (DSMLs) that automate key portions of QoS-enabled component middleware configuration, deployment, and evaluation. The following sections describe QoSPML which is a DSML developed using the *Generic Modeling Environment* (GME) [12] to model Real-time CORBA policies and to enable the automatic generation of configuration metadata.

3.3.1 Motivation

Standard *distributed object computing* (DOC) middleware provides application programming interfaces (APIs) that developers use to configure infrastructure and application components *imperatively* to provide predictability, satisfy timing constraints,

and preserve prioritized access to shared resources. Standards-compliant [15] Real-time CORBA DOC middleware provides standard APIs and policies that allow enterprise DRE systems to configure and control various resources, such as (1) *processor resources* via priority mechanisms, thread pools, and synchronizers, for real-time applications with fixed priorities, (2) *communication resources* via protocol properties and explicit bindings to server objects using priority bands and private connections, and (3) *memory resources* via bounding the size of request buffers and thread pools.

The standard APIs for programming QoS policies in Real-time CORBA, however, are complicated. Moreover, the imperative model for programming these features requires application developers to have detailed knowledge of the underlying semantics and implementation in order to configure these policies correctly. Over the past several years, however, QoS-enabled component middleware, such as CIAO [23], Qedo [18], and Prism [19], has evolved to support QoS configuration via standard XML descriptors that are specified declaratively and processed automatically by the middleware deployment and configuration runtime environments [2].

Although using XML descriptors to configure the QoS properties of the system reduces the amount of code written imperatively, it also introduces new complexities, such as verbose syntax, lack of readability at scale, and a high degree of accidental complexity and fallibility. QoSPML was developed to alleviate these complexities and to enable the seamless configuration of key QoS properties of Real-time CCM [23] components.

3.3.2 Structure of QoSPML

The Real-time CORBA specification provides many QoS policies for controlling application behavior. To comply with the standards, and as illustrated in Figure 5, the following QoS policy types can be modeled in QoSPML: a *priority model policy*, a *thread pool policy*, and a *connection policy*. QoSPML organizes policies into logical groups named *policy sets*, which enable the specification of alternative configurations in the same QoS model. The connection and thread pool policies are modeled as references to actual resources to permit resource sharing among separate policy sets. For example, the same thread pool policy can be shared between two different policy sets, while both policy sets define a different connection and priority policy.

3.3.3 Functionality of QoSPML

QoSPML enables developers of enterprise DRE systems to specify and control the following Real-time CORBA QoS policies via visual models:

Propagation of priorities: Real-time CORBA defines two ways to propagate end-to-end priorities: *server-declared* and *client-propagated*. In the server-declared model the priority at which requests run is determined by the server, whereas in the client-propagated model the server honors the request priority assigned by the client. These priority propagation schemes are modeled in the *PriorityModelPolicy* element. The type of propagation scheme is selected via the *PriorityModel* enumeration attribute and the priority is specified with the priority attribute.

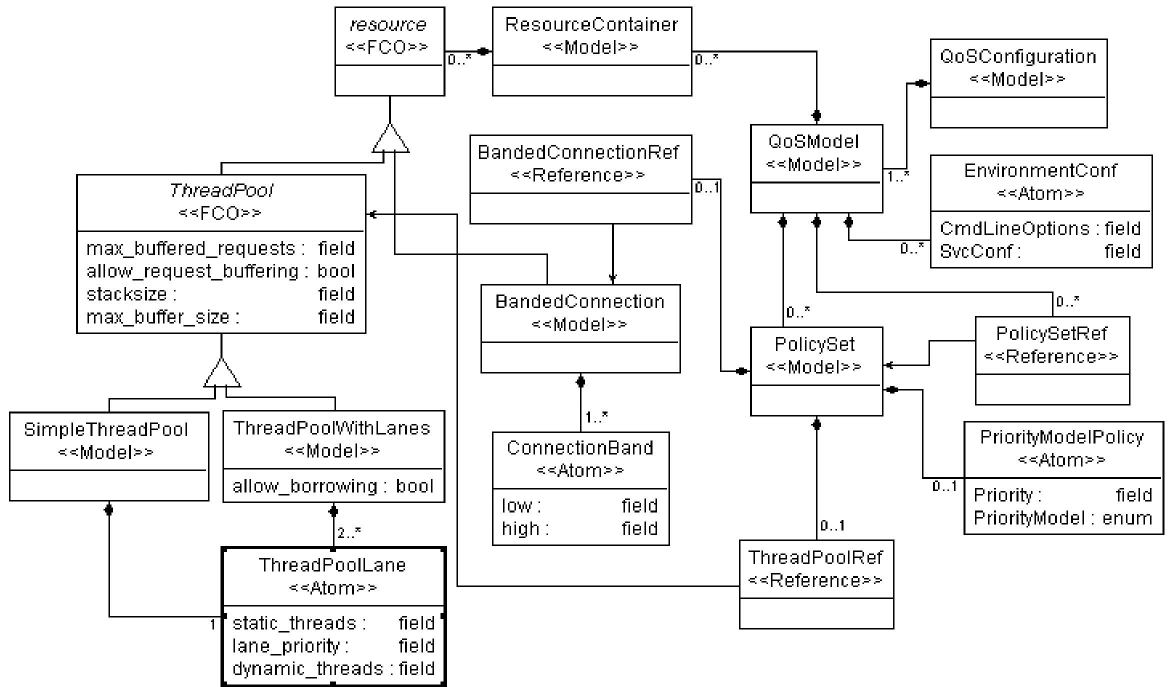

Figure 5: GME Metamodel of QoSPML

Specification of threading model: Each *ThreadPool* model encapsulates data that specifies the properties of a thread pool in Real-time CORBA. For example, developers can set the stack size associated with the thread pool, allow/disallow request buffering and set the maximum number of requests to be buffered and the corresponding buffer size. A thread pool has a set number of pre-spawned *static threads* and up to a maximum limit of *dynamic threads* spawned on-demand only if all static threads are in use. QoSPML supports two types of thread pools: (1) the *SimpleThreadPool* model, which has a single priority lane and allows lower priority client-propagated requests to exhaust all the static and dynamic threads and starve higher priority requests and (2) the *ThreadPoolWithLanes* model, which creates multiple lanes for different priorities to

prevent lower priority client-propagated requests from exhausting all pool's threads. If thread borrowing is enabled, higher priority requests can temporarily promote a thread from a lower priority pool to run the request at the higher priority.

Specification of connection bands: Another Real-time CORBA feature supported by QoSPML is banded connections, which are specified by the *BandedConnections* policy element. These connections are logically divided into *ConnectionBands*, which have a *low* and *high* attribute for specifying the range of priorities of the requests traveling on that band.

Constraint-Checking and Model Interpretation: The GME in which QoSPML was developed provides a powerful constraint-checking mechanism which can be utilized by tool developers in order to ensure the correctness of the models created with their tools. GME synthesizes the basic constraints based on the DSML structure and allow the manual specification of further constraints which cannot be automatically deduced.

Another important capability that GME provides is the ability to develop and associate model interpreters with a particular tool. The model interpreter can access and manipulate the in-memory representations of models created by the particular DSML which it is able to interpret. This allows the model interpreter to do anything from model-transformations to code or configuration generation. In the case of QoSPML, we extract the information captured by the models and map it to a semantically equivalent XML document.

Advantages: Figure 6 illustrates portions of an example QoS configuration using QoSPML. The highlighted region in the figure illustrates the priority model policy and defines references to the connection bands and the thread pool with lanes elements. The

XML below shows a snippet of the generated via model interpretation configuration metadata.

QoSPML addresses the challenges discussed in Section 3.1. In particular, it allows developers to avoid writing applications that use the convoluted Real-time CORBA imperative APIs directly, while still providing control over QoS policies. QoSPML also enables application developers and performance engineers to provision the QoS of applications in enterprise DRE systems via higher-level models that QoSPML converts automatically into lower-level Real-time CORBA QoS policies expressed using XML.

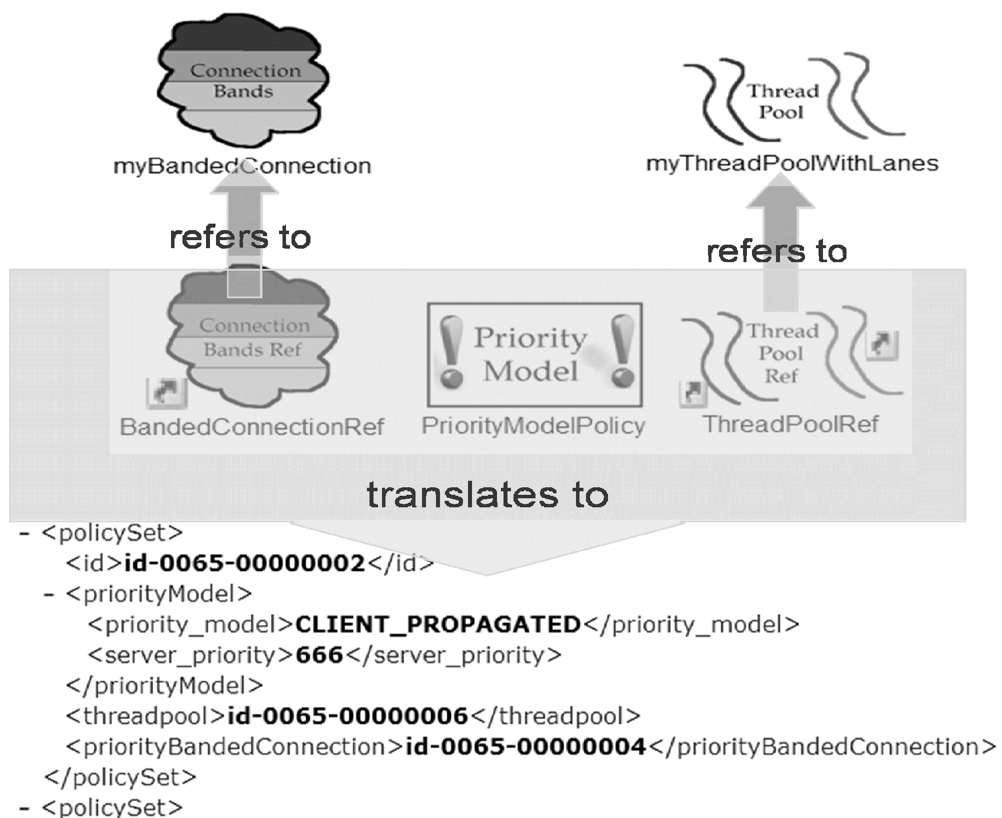

Figure 6: QoS Configuration Snippet of a Model and its Interpretation

3.4 Resolving ARMS MLRM Challenges with QoSPML

We now examine how the QoSPML DSML described in the previous section can be applied to address the challenges discussed in Section 3.1 which arose when developing, evolving, and evaluating the ARMS MLRM case study for shipboard computing enterprise DRE systems.

3.4.1 Configuring Infrastructure and Application Components for QoS

Challenge 1 in Section 3.1 described the difficulties associated with writing applications using the Real-time CORBA API imperatively. QoSPML provides a more scalable and robust approach to configuring the QoS properties of the CCM components being developed, or reused, by enabling developers to specify these properties declaratively and visually. Developers use QoSPML to specify the QoS policies that determine the threading, connection, and priority propagation mechanisms used for a particular component and group these policies into policy sets. The specified mechanisms are modeled in terms of the actual system resources that implement them, which makes it possible for different policy sets to share the same instance of a resource at the middleware layer. QoSPML also enables developers to verify the correctness of their models by providing constraint checking mechanisms embedded in the language. The QoS models can be interpreted by means of a model interpreter that generates correct metadata descriptors understood by the Real-time CCM middleware runtime.

In the context of ARMS, QoSPML facilitates the seamless configuration of components for QoS in each layer of MLRM because it allows application developers to

bypass the tedious tasks of hard-coding the Real-time CORBA code or hand-crafting the XML descriptors that can be used to describe the QoS configuration.

3.4.2 Meeting the QoS Needs of the Various MLRM Subsystems

Challenge 2 in Section 3.1 discussed the diversity of services and QoS requirements supported by the ARMS MLRM infrastructure. Each of these QoS requirements is hard to achieve separately and even harder to achieve in combination. Fortunately, QoSPML detaches configuration developers from the inherent complexities of the configuration code and allows them to concentrate on the general logic of the application components.

In the context of ARMS, a major cause of missed deadlines is priority inversions, where lower priority requests access a resource at the expense of higher priority requests. Priority inversions must be prevented or bounded since they can cause the ARMS applications to miss their deadlines. QoSPML's ThreadPoolWithLanes element can be used in conjunction with the BandedConnection and the PriorityModelPolicy elements to configure MLRM properly and reduce priority inversions.

The ThreadPoolWithLanes feature of QoSPML can be used to meet some of the QoS needs of ARMS. By using this feature, the MLRM will be configured so that lower priority requests cannot exhaust threads allocated for higher priority requests when a request is executed. Long-running requests in MLRM can also exhaust the maximum number of static threads, causing the system to miss deadlines. QoSPML therefore allows ARMS MLRM developers to specify the maximum number of dynamically spawned threads to better manage long running requests and periodic high loads.

The BandedConnection element in QoSPML allows MLRM developers to control network resources effectively by separating lower and higher priority requests so they do not share the same multiplexed connection. In multiplexed connections, requests are queued and serviced on a FIFO basis, where low priority requests could be scheduled first. By using priority bands, developers can partition the communication links between application and MLRM components based on a range of priority values. This QoS policy ensures that low priority requests travel on separate paths from high priority requests, therefore preventing priority inversions. A beneficial side-effect of this partitioning mechanism is that it decreases latency and improves response time.

It is also important to ensure the portability of priorities in cases when ARMS application and MLRM component run atop different OS platforms with different priority ranges. Once the necessary priority mappings have been defined, QoSPML's PriorityModelPolicy feature can be used to preserve the end-to-end priorities and to define the priority propagation scheme used to configure Real-time CORBA policies. As discussed in Section 3.2.3, there are currently two types of policies: server declared and client propagated.

3.4.3 Using MDD Tools to Generate XML Metadata

Challenge 4 in Section 3.1 described the complexities introduced by applying XML metadata to configure DRE systems. We used QoSPML to bypass the XML coding necessary to configure application and middleware components declaratively, which raised the level of abstraction by means of a visual DSML. We used this MDD tools to formally

model the configuration space and enable the automatic generation of configuration code. QoSPML therefore allows developers to concentrate on the actual design of the enterprise DRE system, while shielding them from the accidental complexities of the configuration artifacts. It also makes rapid (re)configuration possible, thus allowing developers to evolve the system more conveniently.

In the context of ARMS, we had initially used validation tools, such as XML SPY, to verify the syntactic correctness of the XML metadata against the schema to which it conforms. Unfortunately, these validation tools miss many problems with handcrafted XML. In contrast, QoSPML provides a more effective solution because it uses GME's powerful constraint-checking facility to ensure that models are correct-by-construction. The generated XML descriptors are therefore also correct as long as the output of the QoSPML interpreter conforms to the XML schema that describes the documents.

3.4.4 Managing and Refining the System Configuration Space

Challenge 3 in Section 3.1 described how managing a large amount of XML metadata is cumbersome and that extracting information from it requires significant effort. Likewise, challenge 5 in Section 3.1 discussed that it is even harder to modify XML-based configuration files in response to (1) changing system requirements, (2) better understanding of the QoS needs to the application, or (3) uncovered design weaknesses. For example, even a single typo in an XML file can compromise the document structure and cause the parsers to fail, which makes handcrafted XML files extremely hard to manage and evolve.

In the context of ARMS, by using QoSPML developers no longer have to deal with XML metadata directly. Instead, they can use visual models to perform their tasks from a domain-centric perspective. After making the necessary changes to the system configuration, they can regenerate the descriptors quickly and correctly, which scales much better for enterprise DRE systems like ARMS.

3.5 Benefits of MDD tools to the component-based ARMS applications

This chapter focused on the experience gained while integrating and applying the QoSPML DSML to the DARPA ARMS MLRM services for naval shipboard computing enterprise DRE systems. The benefits observed by applying our DSML to the component-based ARMS applications and infrastructure services thus far include:

- Using highly configurable component middleware, such as CIAO [23] and DAnCE [2], enhances software development quality and productivity. Unfortunately it also introduces extra complexities, which are hard to handle in an *ad hoc* manner for enterprise DRE systems.
- Using DSMLs can expedites application development and system QoS configuration by providing proper integration of MDD tools with the underlying component middleware infrastructure. In the ARMS MLRM case study, the QoSPML DSML was used to simplify the evaluation of many different system configurations and facilitate QoS-related “what if” scenarios prior to the integration or even the development phase. QoSPML also plays an important role in enterprise DRE system

evolution because it provides a way to evaluate alternative system configurations visually and empirically.

- QoSPML can help to reduce the learning curve for the end users. For example, in the ARMS MLRM case study, application developers needed little knowledge of the Real-time CORBA QoS policy APIs and the CIAO XML descriptors that declaratively configure these policies in Real-time CCM. Instead, they used the higher-level models of QoS policy provisioning mechanisms provided by QoSPML.

Although our use of MDD technologies solves many hard problems encountered in the ARMS program, it also leaves room for some improvement and future work:

- Despite the fact that QoSPML facilitates the QoS configuration of enterprise DRE systems based on Real-time CORBA, developers are still faced with the question of what constitutes a "good" configuration.
- Although MDD removes many complexities associated with handcrafted solutions, developers are still faced with the challenge of evolving existing models when the respective domain evolves. Although model evolution tools, such as GREAT [9], exist they are hard to use and only provide partially automated solutions.

This experience motivates further research on automated QoS configuration and deployment techniques to uncover effective heuristics to guide us in the complicated process of enterprise DRE system evaluation-driven QoS configuration, as well as further research on model migration to simplify the process evolving DSMLs as the understanding of their respective domains matures.

The GME open-source domain-specific modeling framework can be downloaded from www.isis.vanderbilt.edu/projects/GME. QoSPML was integrated with the open-source Component Synthesis with Model Integrated Computing (CoSMIC) tool chain and is available at www.dre.vanderbilt.edu/cosmic.

APPENDIX A

REPOMAN AND QOSPML ON THE WEB

http://www.cs.wustl.edu/~schmidt/CIAO.html

RepoMan is available as part of the ***C**omponent **I**ntegrated **ACE** **ORB*** (CIAO). For instructions on how to download and install CIAO please follow the link above.

http://www.dre.vanderbilt.edu/cosmic/

QoSPML has been integrated in the **Co**mponent **S**ynthesis with **M**odel **I**ntegrated **C**omputing (CoSMIC) tool chain and can be found under the RTConfig worksheet in CoSMIC. To download and install CoSMIC and obtain QoSPML go to the link above.

http://www.isis.vanderbilt.edu/projects/GME/

The Generic Modeling Environment is the meta-modeling environment used to develop both QoSPML and CoSMIC. You can learn more about GME by visiting the website above.

REFERENCES

[1] ARMS DARPA Website, dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6, Jan 2006.

[2] Deng, G., Balasubramanian, J., Otte, W., Schmidt, D. and Gokhale, A. (2005, Nov), "DAnCE: A QoS-enabled Component Deployment and Conguration Engine," *Proceedings of the 3rd Working Conference on Component Deployment*. Grenoble, France.

[3] Deutsch, P, "DEFLATE Compressed Data Format Specification version 1.3", Network Working Group, RFC 1951.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides J., "Design Patterns Elements of Reusable Object-Oriented Software," Addison-Wesley, 1994.

[5] Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G., Deng, G., Turkay, E., Parsons, J. , and Schmidt, D. (2005). "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture.* [in press].

[6] Grassi, V., Mirandola, R., and Sabetta, A., "From Design to Analysis Models: A Kernel Language for Performance and Reliability Analysis of Component-based Systems," *Fifth International Workshop on Software and Performance*, Jul 2005.

[7] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 1987.

[8] Hu, J., Pyarali, I., and Schmidt, D., ``The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks," The Parallel and Distributed Computing Practices journal, special issue on Distributed Object-Oriented Systems, Vol. 3, No. 1, March 2000.

[9] Karsai G. and Agrawal A. and Shi F. and Sprinkle J., "On the use of Graph Transformations in the Formal Specification of Computer-Based Systems," *Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems*, Huntsville, AL, Apr 2003.

[10] Karsai, G., Sztipanovits, J., Ledeczi, A. and Bapty, T. "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE,* Jan 2003.

[11] Krishna, A., Turkay, E., Gokhale, A., and Schmidt, D., "Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems", *11th IEEE Real-Time and Embedded Technology and Applications Symposium*, Mar 2005.

[12] Ledeczi, A., Maroti, M., Karsai G., and Nordstrom G., "Metaprogrammable Toolkit for Model-Integrated Computing*", Proceedings of the IEEE International Conference on the Engineering of Computer-Based Systems Conference*, Mar 1999.

[13] Object Management Group: Deployment and Configuration Adopted Submission, OMG Document ptc/03-07-08 edn. (2003).

[14] Object Group Management (2003, May), Light Weight CORBA Component Model Revised Submission, Ed. OMG Document realtime/03-05-05.

[15] Object Management Group (2002, Aug). Real-time CORBA Specification. Ed. OMG Document formal/02-08-02.

[16] Paunov, S., Hill, J., Schmidt, D., Baker, S., Slaby, J., "Domain-Specific Modeling Languages for Configuring and Evaluating Enterprise DRE System Quality of Service," *Proceedings of the 13th Annual IEEE International Conference on the Engineering of Computer Based Systems,* Potsdam, Germany, 2006.

[17] Paunov, S., Schmidt, D., "RepoMan: A Component Repository Manager for Enterprise Distributed Real-time and Embedded Systems," *Proceedings of the 44th Association of Computing Machinery (ACM) Southeast Conference,* Melbourne, Florida, USA, 2006.

[18] Ritter, T., Born, M., Unterschutz, T., and Weis, T., "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," *Proceedings of the 36th Hawaii International Conference on System Sciences,* Honolulu, Hawaii, Jan 2003.

[19] Roll, W. "Towards Model-Based and CCM-Based Applications for Real-Time Systems," *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, IEEE/IFIP, Hakodate, Hokkaido, Japan, May 2003.

[20] Schmidt, D., Schantz, R., Masters, M., Cross, J., Sharp, D., and DiPalma L., "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," CrossTalk, November, 2001.

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

[22] Slaby, J., Baker, S., Hill, J., Schmidt, D., "Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS," ISIS Technical Report ISIS-05-604, Oct 2005.

[23] Wang, N. and Gill, C. (2003, Jan), "Improving Real-time System Configuration via a QoS-aware CORBA Component Model," *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems.* Minitrack, HICSS 2003.

[24] Ye, J., Loyall, J., Shapiro, R., Schantz, R., Neema, S., Abdelwahed, S., Mahadevan, N., Koets, M., Varner, D., "Model-Based Approach to Designing QoS Adaptive Applications", *25th International Real-Time Systems Symposium*, May 2004.

[25] Natarajan, B., Schmidt, D., and Vinoski, S., The CORBA Component Model Part 3: The CCM Container Architecture and Component Implementation Framework, C/C++ Users Journal, September, 2004.

[26] Schmidt, D. and Vinoski, S., Object Interconnections: The CORBA Component Model: Part 1, Evolving Towards Component Middleware, C/C++ Users Journal, February, 2004.