RepoMan: A Component Repository Manager for Enterprise Distributed Real-time and Embedded Systems

Stoyan Paunov
Vanderbilt University
Nashville, TN
spaunov@isis.vanderbilt.edu

Douglas C. Schmidt
Vanderbilt University
Nashville, TN
d.schmidt@vanderbilt.edu

Abstract

Repository managers keep track of software versions, implementations, and configuration metadata in distributed computing environment to enable the (re)deployment and (re)configuration of applications and facilitate online component upgrades. This paper provides two contributions to the study of repository managers for component-based enterprise distributed real-time and embedded (DRE) systems. First it describes how we overcame the design challenges associated with developing RepoMan, which is a repository manager targeted for heterogeneous enterprise DRE systems. Second, it explains how we applied RepoMan to an enterprise DRE shipboard computing system.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: *Client/Server, Distributed Applications*.

General Terms

Management, Performance, Design, Standardization

Keywords

Distributed systems, Component architectures

1. Introduction

Meeting distribution challenges of component-based enterprise DRE systems. Enterprise systems in many domains are being developed using applications composed of distributed components running on feature–rich middleware. Examples of such component middleware platforms include J2EE, .NET, and the CORBA Component Model (CCM). In these platforms, software components are assembled and composed to provide various types of reusable services to a range of application domains.

Certain types of component middleware, such as Real-time CCM [10], are being applied to the domain of *enterprise distributed real-time and embedded (DRE) systems*, such as total ship computing environments or electrical power control systems. These systems are designed to provide users with quality of service (QoS) support to process the right data in the right place at the right time over a grid of computers. QoS properties required by enterprise DRE systems include the low latency and jitter expected in conventional real-time and embedded systems, as well as high throughput, scalability, and reliability expected in conventional enterprise distributed systems.

Although component-based enterprise DRE systems help address the problems with prior generations of inflexible, monolithic, functionally-designed, and "stove-piped" 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 *repository managers* that (1) keep track of software versions, implementations for component implementations in heterogeneous environments and (2) supply configuration metadata to facilitate the online upgrades, reconfiguration, and redeployment of components. Developing repository managers for enterprise DRE systems is hard, however. Key challenges include the need to support cross-platform portability, ensure responsiveness and scalability, and enable dynamic updates within time constraints.

This paper discusses the design and implementation of *RepoMan*, which is an implementation of the OMG CCM Repository Manager specification [6] tailored to 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 *Component-Integrated ACE ORB (CIAO)* open-source implementation of Real-time CCM, and applied to several enterprise DRE systems, including Naval shipboard computing systems and NASA earth science missions.

The remainder of this paper is organized as follows: Section 2 presents a case study that motivates the need for a component repository manager in enterprise DRE systems; Section 3 discusses the structure and functionality of our RepoMan CCM-based repository manager; Section 4 explains the design challenges that we overcame while developing RepoMan and applying it to shipboard computing; Section 5 compares our work with related work; and Section 6 presents concluding remarks and outlines our lessons learned during this project.

2. Case Study: An Enterprise DRE System for Shipboard Computing

We motivate our work on RepoMan using the DARPA ARMS (dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6) *Multi-*

ACM SE'06, March, 10-12, 2006, Melbourne, Florida, USA

Layer Resource Management (MLRM) middleware, which is described as a running example throughout the paper. The ARMS MLRM services are designed to support *Total Ship Computing Environments* (TSCEs), which form the basis for next-generation Naval programs [9]. A TSCE is a coordinated grid of computers organized into multiple data centers that manage many aspects of a ship's power, navigation, command and control, and tactical operations. To make a 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 [10] Real-time CCM middleware. Real-time CCM combines *Lightweight CCM* [5] mechanisms (such as standards for specifying, implementing, packaging, assembling, and deploying components) and *Real-time CORBA* [7] mechanisms (such as thread pools and priority preservation policies) to simplify and automate the (re)deployment and (re)configuration of application components in DRE systems.

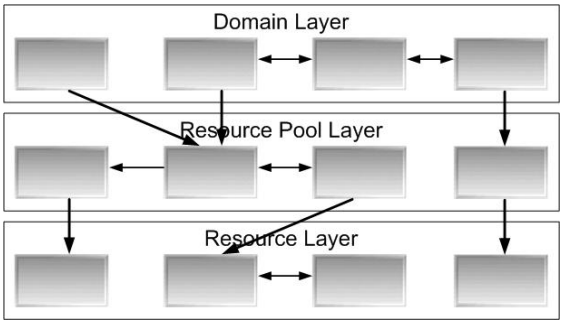

Figure 1. Component-based Architecture of ARMS MLRM

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 have infrastructure components called *node provisioners* that receive commands to spawn applications in every node from a pool manager. The *operational 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 five teams at different locations.

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 in C++ and OpenCCM in Java, (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.

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 [1], 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.

3. The 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 component 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.

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 reside in dynamic link libraries (DLLs).

One way to design a component repository would 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 an mediator [3] 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 4.2 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 for every client instance, so RepoMan's design is much more efficient and scalable.

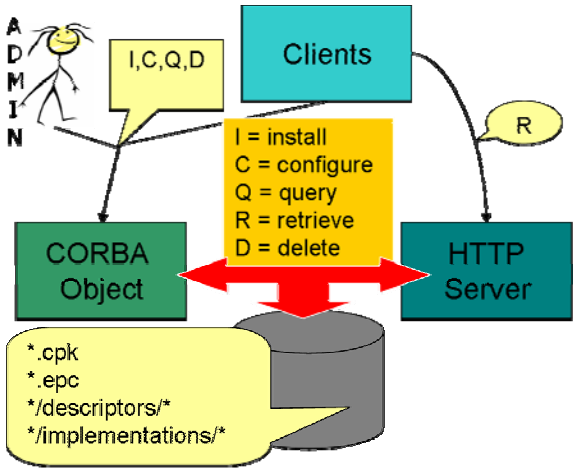

Figure 2: The RepoMan Architecture

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 a specific component's properties, (2) analyze this 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.

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 can install a package either from a specified location on a 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 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 them all by names or types. 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 that are now accessible via the collocated HTTP server.

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 3.1 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.

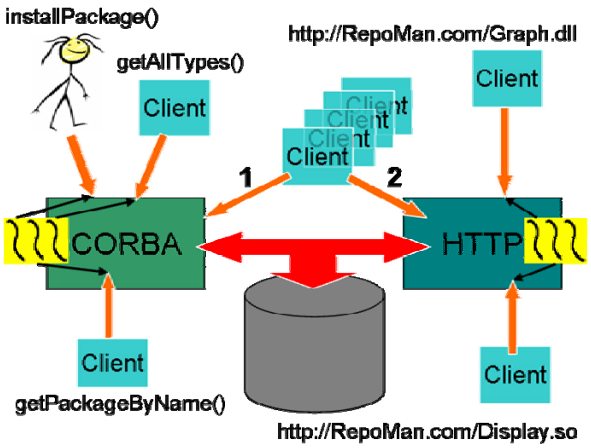

Figure 3: RepoMan in Action

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 retrieve 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 [4] 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 link in the location field from the corresponding *PackageConfiguration* to request the implementation from RepoMan's HTTP server, which downloads the corresponding artifact to the node provisioner that then performs 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 [2], 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 *PackageConfiguration* before its contents can be marshaled and downloaded to 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 (Section 4.1) 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 [GoF:04], 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 [5]) 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 down them to clients, thereby minimizing CPU and I/O processing considerably and significantly decreasing 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 proc-

essing 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 a repository it caches them locally to minimize subsequent access time and to ensure their availability.

Problem → Organizing and managing package data. RepoMan expects 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 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, RepoMan can ensure 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 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 2, 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 its collocated HTTP server, as depicted in Figure 3. This task requires RepoMan to navigate through the *PackageConfiguration* element all the way down to the implementation artifacts, which are leaves in an "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 a recursive description of the component implementation tree that may consist of multiple monolithic and assembly-based components and a description of their interconnection. *PackageConfigurations* are 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 [3], 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 allows RepoMan to perform its task 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: Providing a 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 a 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 accessing shared resources. - Since synchronization mechanisms incur mutual exclusion overhead, their use should be limited only where they are actually 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 in its entirety by only synchronizing access to the hash tables, which provides a lightweight synchronization design that is more efficient than the alternatives (such as the Monitor Object or Active Object patterns [8]) 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 multi-threading and a lightweight synchronization design, as well as the optimizations discussed in Challenge 2, helped minimize the response time of the RepoMan, thereby contributing to minimizing the overall cost of redeployment, reconfiguration, and component update activities.

5. Related Work

The SOFtware Appliances (SOFA) architecture [11] provides a framework for building applications composed of a set of dynamically downloadable and updatable components. SOFA provides a *Type Information Repository* (TIR) that manages the evolution of component interfaces and implements version control. It is complemented by another repository that stores component implementations. Rather than separating these functions, RepoMan combines them into the same service, which allows it to tie component implementations with the interfaces that they implement. Another difference is that SOFA targets Java component implementations, whereas the CCM Repository Manager and thus RepoMan can support many programming languages.

The OpenCCM Distributed Computing Infrastructure (DCI) (www.objectweb.org) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. DCI implements the Packaging and Deployment (P&D) model defined in the original CCM specification, however, which omits key aspects in the component configuration and deployment cycle, including the package repository management supported by RepoMan. We are working with the OpenCCM team to enhance their DCI so that it is compliant with the OMG D&C specification and it is interoperable with RepoMan and DAnCE [1].

6. Concluding Remarks

This paper motivated and described RepoMan, which is an implementation of the CCM Repository Manager specification that keeps track of component implementations and their respective configurations for enterprise DRE systems. 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 patterns to RepoMan helped ensure that its design used best practices associated with solving recurring problems and leveraging the experience of experienced developers. Patterns applied to RepoMan included Iterator, Memento, Null object, and Visitor in the COBRA object and Bridge, Component Configurator, Singleton, Strategy, Wrapper Facade in the HTTP server.
- Amortizing certain costs over the RepoMan lifetime helped to improve its performance. Although externalizing the *PackageConfiguration* slows down the installation, it enabled us to optimize performance over the lifetime of the system since retrieval operations are much more frequent than install operations.

The implementation of RepoMan is open-source and can be downloaded along with the CIAO Real-time CCM middleware from www.dre.vanderbilt.edu/CIAO.

References

[1] 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.

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

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

[4] 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.

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

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

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

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

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

[10] 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.

[11] Plasil, F., Balek, D., Janecek, R.. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998.