Middleware Support for Dynamic Component Updating

Jaiganesh Balasubramanian[1], Balachandran Natarajan*[2], Douglas C. Schmidt[1], Aniruddha Gokhale[1], Jeff Parsons[1], and Gan Deng[1]

[1] Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37203, USA,
(jai,schmidt,gokhale,parsons,dengg)@dre.vanderbilt.edu
[2] Veritas Software India Ltd., Pune, India
bala.natarajan@veritas.com

Abstract

Component technologies are increasingly being used to develop and deploy distributed real-time and embedded (DRE) systems. To enhance flexibility and performance, developers of DRE systems need middleware mechanisms that decouple component logic from the binding of a component to an application, i.e., they need support for dynamic updating of component implementations in response to changing modes and operational contexts. This paper presents three contributions to R&D on dynamic component updating. First, it describes an inventory tracking system (ITS) as a representative DRE system case study to motivate the challenges and requirements of updating component implementations dynamically. Second, it describes how our SwapCIAO middleware supports dynamic updating of component implementations via extensions to the server portion of the Lightweight CORBA Component Model. Third, it presents the results of experiments that systematically evaluate the performance of SwapCIAO in the context of our ITS case study. Our results show that SwapCIAO improves the flexibility and performance of DRE systems, without affecting the client programming model or client/server interoperability.

1 Introduction

Component middleware is increasingly being used to develop and deploy next-generation distributed real-time and embedded (DRE) systems, such as shipboard computing environments [1], inventory tracking systems [2], avionics mission computing systems [3], and intelligence, surveillance and reconnaissance systems [4]. These DRE systems must adapt to changing modes, operational contexts, and resource availabilities to sustain the execution of critical missions. However, conventional middleware platforms, such as J2EE, CCM, and .NET, are not yet well-suited for these types of DRE systems since they do not facilitate

* Work performed while author at Vanderbilt University.

the separation of quality of service (QoS) policies from application functionality [5].

To address limitations of conventional middleware, *QoS-enabled component middleware*, such as CIAO [6], Qedo [7], and PRiSm [8], explicitly separates QoS aspects from application functionality, thereby yielding systems that are less brittle and costly to develop, maintain, and extend [6]. Our earlier work on QoS-enabled component middleware has focused on (1) identifying patterns for composing component-based middleware [9, 10], (2) applying reflective middleware [11] techniques to enable mechanisms within the component-based middleware to support different QoS aspects [12], (3) configuring real-time aspects [6] within component middleware to support DRE systems, and (4) developing domain-specific modeling languages that provide design-time capabilities to deploy and configure component middleware applications [13]. This paper extends our prior work by *evaluating middleware techniques for updating component implementations dynamically and transparently (i.e., without incurring system downtime) to optimize system behavior under diverse operating contexts and mode changes.*

Our dynamic component updating techniques have been integrated into *SwapCIAO*, which is a QoS-enabled component middleware framework that enables application developers to create multiple implementations of a component and update (*i.e.* "swap") them dynamically. SwapCIAO extends CIAO, which is an open-source[3] implementation of the OMG Lightweight CCM [14], Deployment and Configuration (D&C) [15], and Real-time CORBA [16] specifications. Sidebar 1 outlines the features of Lightweight CCM relevant to this paper.

The key capabilities that SwapCIAO adds to CIAO include (1) mechanisms for updating component implementations dynamically without incurring system downtime and (2) mechanisms that transparently redirect clients of an existing component to the new updated component implementation. As discussed in this paper, key technical challenges associated with providing these capabilities involve updating component implementations without incurring significant overhead or losing invocations that are waiting for or being processed by the component.

The remainder of this paper is organized as follows: Section 2 describes the structure and functionality of an inventory tracking system, which is a DRE system case study that motivates the need for dynamic component implementation updating; Section 2.2 describes the key design challenges in provisioning the dynamic component implementation updating capability in QoS-enabled component middleware systems; Section 3 describes the design of SwapCIAO, which provides dynamic component implementation updating capability for Lightweight CCM; Section 4 analyzes the results from experiments that systematically evaluate the performance of SwapCIAO for various types of DRE applications in our ITS case study; Section 5 compares SwapCIAO with related work; and Section 6 presents concluding remarks.

[3] SwapCIAO and CIAO are available from `www.dre.vanderbilt.edu/CIAO`.

Sidebar 1: Overview of Lightweight CCM

The OMG Lightweight CCM [14] specification standardizes the development, configuration, and deployment of component-based applications. Applications developed with Lightweight CCM are not tied to any particular language, platform, or network. *Components* in Lightweight CCM are implemented by *executors* and collaborate with other components via *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

Assemblies of components in Lightweight CCM are deployed and configured via the OMG D&C [15] specification, which manages the deployment of an application on nodes in a target environment. The information about the component assemblies and the target environment in which the components will be deployed are captured in the form of XML descriptors defined by the D&C specification. A standard deployment framework parses XML assembly descriptors and deployment plans, extracts connection information from them, and establishes the connections between component ports. In the context of this paper, a *connection* refers to the high-level binding between an object reference and its target component, rather than a lower-level transport connection.

2 Case Study to Motivate Dynamic Component Updating Requirements

To examine SwapCIAO's capabilities in the context of a representative DRE system, we developed an *inventory tracking system* (ITS), which is a warehouse management infrastructure that monitors and controls the flow of goods and assets within a storage facility. Users of an ITS include couriers (such as UPS, DHL, and Fedex), airport baggage handling systems, and retailers (such as Walmart and Target). This section describes (1) the structure/functionality of our ITS case study and (2) the key requirements that SwapCIAO dynamic component updating framework had to address. Naturally, SwapCIAO's capabilities can be applied to many DRE systems – we focus on the ITS case study in this paper to make our design discussions and performance experiments concrete.

2.1 Overview of ITS

An ITS provides mechanisms for managing the storage and movement of goods in a timely and reliable manner. For example, an ITS should enable human operators to configure warehouse storage organization criteria, maintain the inventory throughout a highly distributed system (which may span organizational and national boundaries), and track warehouse assets using decentralized operator consoles. In conjunction with colleagues at Siemens [17], we have developed the ITS shown in Figure 1 using SwapCIAO. This figure shows how our ITS consists of the following three subsystems:

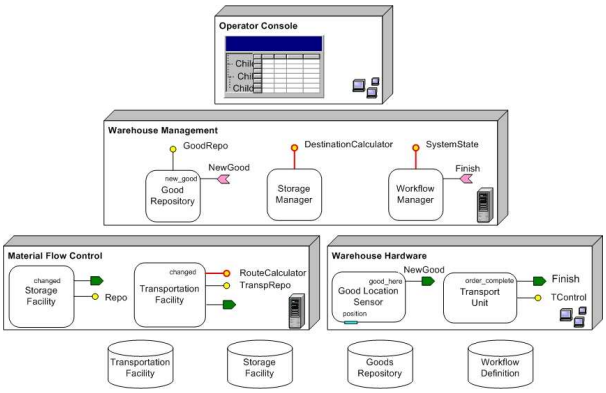

Fig. 1. Key Components in ITS

- **Warehouse management**, whose high-level functionality and decision-making components calculate the destination locations of goods and delegate the remaining details to other ITS subsystems. In particular, the warehouse management subsystem does not provide capabilities like route calculation for transportation or reservation of intermediate storage units.
- **Material flow control**, which handles all the details (such as route calculation, transportation facility reservation, and intermediate storage reservation) needed to transport goods to their destinations. The primary task of this subsystem is to execute the high-level decisions calculated by the warehouse management subsystem.
- **Warehouse hardware**, which deals with physical devices (such as sensors) and transportation units (such as conveyor belts, forklifts, and cranes).

2.2 Requirements for Dynamic Component Updates

Throughout the lifetime of an ITS, new physical devices may be added to support the activities in the warehouse. Likewise, new models of existing physical devices may be added to the warehouse, as shown in Figure 2. This figure shows the addition of a new conveyor belt that handles heavier goods in a warehouse. The ITS contains many software controllers, which collectively manage the entire system. For example, a software controller component manages each physical device controlled by the warehouse hardware subsystem. When a new device is introduced, a new component implementation must be loaded dynamically into the ITS. Likewise, when a new version of a physical device arrives, the component that controls this device should be updated so the software can manage the new version. ITS vendors are responsible for providing these new implementations.

As shown in Figure 2, a workflow manager component is connected to a conveyor belt component using a facet/receptacle pair and an event source/sink pair. To support this scenario, the ITS needs middleware that can satisfy the following three requirements:

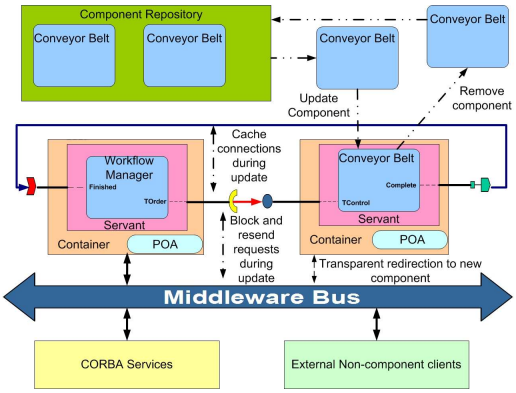

Fig. 2. Component Updating Scenario in ITS

1. Consistent and uninterrupted updates to clients. As part of the dynamic update process, a component's implementation is deactivated, removed, and updated. To ensure that the ITS remains consistent and uninterrupted during this process, the middleware must ensure that (1) ongoing invocations between a component and a client are completed and (2) new invocations from clients to a component are blocked until its implementation has been updated. Figure 2 shows that when a conveyor belt's component implementation is updated, pending requests from the workflow manager to the conveyor belt component to move a new good to a storage system should be available for processing after the implementation is updated. Section 3.1 explains how SwapCIAO supports this requirement.

2. Efficient client-transparent dynamic component updates. After a component is updated, the blocked invocations from clients should be redirected to the new component implementation. This reconfiguration should be transparent to clients, *i.e.*, they should not need to know when the change occurred, nor should they incur any programming effort or runtime overhead to communicate with the new component implementation. Figure 2 shows how a client accessing an ITS component should be redirected to the updated component transparently when dynamic reconfiguration occurs. Section 3.2 explains how SwapCIAO supports this requirement.

3. Efficient (re)connections of components. Components being updated may have connections to other components through the ports they expose. The connected components and the component being updated share a requires/provides relationship by exchanging invocations through the ports. In Lightweight CCM, these connections are established at deployment time using data provided to the deployment framework in the form of XML descriptors. During dynamic reconfiguration, therefore, it is necessary to cache these connections so they can be restored immediately after reconfiguration. Figure 2 shows how, during the update of a conveyor belt component, its connections to the workflow manager

component must be restored immediately after the new updated conveyor belt component implementation is started. Section 3.3 explains how SwapCIAO supports this requirement.

3 The SwapCIAO Dynamic Component Updating Framework

This section describes the design of SwapCIAO, which is a C++ framework that extends CIAO to support dynamic component updates. Figure 3 shows the

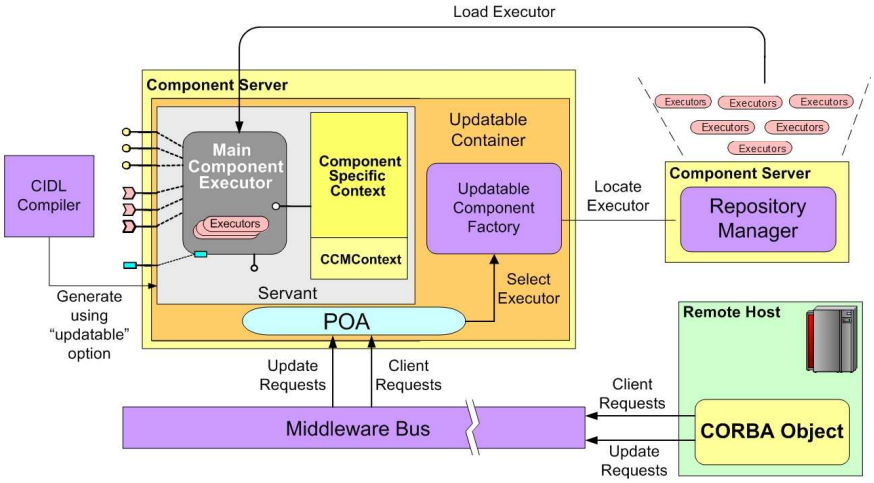

Fig. 3. Dynamic Interactions in the SwapCIAO framework

following key elements in the SwapCIAO framework:

- SwapCIAO's *component implementation language definition* (CIDL) compiler supports the *updatable* option, which triggers generation of "glue code" that (1) defines a factory interface to create new component implementations, (2) provides hooks for server application developers to choose which component implementation to deploy, (3) creates, installs, and activates components within a POA chosen by an application, and (4) manages the port connections of an updatable component.
- The *updatable container* provides an execution environment in which component implementations can be instantiated, removed, updated, and (re)executed. An updatable container enhances the standard Lightweight CCM *session container* [18] to support additional mechanisms through which component creation and activation can be controlled by server application developers.
- The *updatable component factory* creates components and implements a wrapper facade [10] that provides a portable interface used to implement the Component Configurator pattern [10], which SwapCIAO uses to open and load dynamic link libraries (DLLs) on heterogeneous run-time platforms.

– The *repository manager* stores component implementations. SwapCIAO's updatable component factory uses the repository manager to search DLLs and locate component implementations that require updating.

The remainder of this section describes how the SwapCIAO components in Figure 3 address the requirements presented in Section 2.2.

3.1 Providing Consistent and Uninterrupted Updates to Clients

Problem. Dynamic updates of component implementations can occur while interactions are ongoing between components and their clients. For example, during the component update process, clients can initiate new invocations on a component – there may also be ongoing interactions between components. If these scenarios are not handled properly by the middleware some computations can be lost, yielding state inconsistencies.

Solution → Reference counting operation invocations. In SwapCIAO, all operation invocations on a component are dispatched by the standard Lightweight CCM portable object adapter (POA), which maintains a *dispatching table* that tracks how many requests are being processed by each component in a thread. SwapCIAO uses standard POA reference counting and deactivation mechanisms [19] to keep track of the number of clients making invocations on a component. After a server thread finishes processing the invocation, it decrements the reference count in the dispatching table.

When a component is about to be removed during a dynamic update, the POA does not deactivate the component until its reference count becomes zero, *i.e.*, until the last invocation on the component is processed. To prevent new invocations from arriving at the component while it is being updated, SwapCIAO's updatable container blocks new invocations for this component in the server ORB using standard CORBA portable interceptors [20].

Applying the solution to ITS. In the ITS case study, when the conveyor belt component implementation is being updated, the warehouse hardware system could be issuing requests to the conveyor belt component to move goods. The updatable container (which runs in the same host as the conveyor belt component) instructs the SwapCIAO middleware to block those requests. After the requests are blocked by SwapCIAO, the updatable container's POA deactivates the conveyor belt component only when all requests it is processing are completed, *i.e*, when its reference count drops to zero.

3.2 Ensuring Efficient Client-transparent Dynamic Component Updates

Problem. As shown in the Figure 3, many clients can access a component whose implementation is undergoing updates during the dynamic reconfiguration process. In Lightweight CCM, a client holds an object reference to a component. After a component implementation is updated, old object references are no longer valid. The dynamic reconfiguration of components needs to be

transparent to clients, however, so that clients using old references to access updated component do not receive "invalid reference" exceptions. Such exceptions would complicate client application programming and increase latency by incurring additional round-trip messages, which could unduly perturb the QoS of component-based DRE systems.

Solution → Use servant activators to redirect clients to update components transparently. Figure 4 shows how SwapCIAO redirects clients transparently to an updated component implementation. During the component updating process,

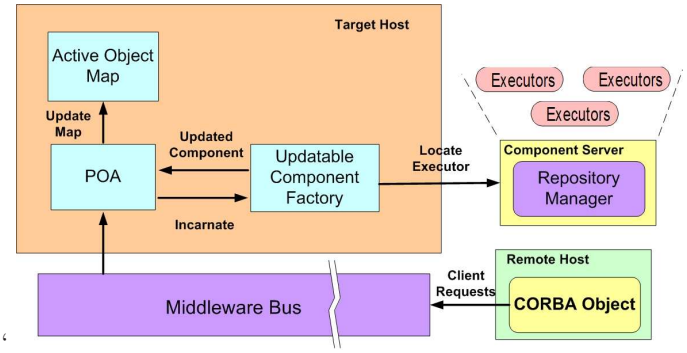

Fig. 4. Transparent Component Object Reference Update in SwapCIAO

the old component implementation is removed. When a client makes a request on the old object reference after a component has been removed, the POA associated with the updatable container intercepts the request via a *servant activator*. This activator is a special type of intercepter that can dynamically create a component implementation if it is not yet available to handle the request. Since the component has been removed, the POA's active object map will have no corresponding entry, so the servant activator will create a new component implementation dynamically.

SwapCIAO stores information in the POA's active object map to handle client requests efficiently. It also uses CORBA-compliant mechanisms to activate servants via unique user id's that circumvent informing clients of the updated implementation. This design prevents extra network round-trips to inform clients about an updated component's implementation.

Applying the solution to ITS. In the ITS case study, when the conveyor belt component implementation is being updated, the warehouse hardware system could be issuing requests to the conveyor belt component to move goods. After the current conveyor belt component is removed, the servant activator in the updatable container's POA intercepts requests from the warehouse hardware subsystem clients to the conveyor belt component. The servant activator then activates a new conveyor belt component implementation and transparently redirects the

requests from the warehouse hardware subsystem to this updated implementation. SwapCIAO uses these standard CORBA mechanisms to enable different component implementations to handle the requests from warehouse hardware subsystem clients transparently, without incurring extra round-trip overhead or programming effort by the clients.

3.3 Enabling (Re)connections of Components

Problem. As discussed in Sidebar 1, Lightweight CCM applications use the standard OMG Deployment and Configuration (D&C) [15] framework to parse XML assembly descriptors and deployment plans, extract connection information from them, and establish connections between component ports. This connection process typically occurs during DRE system initialization. When component implementations are updated, it is therefore necessary to record each component's connections to its peer components since their XML descriptors may not be available to establish the connections again. Even if the XML is available, reestablishing connections can incur extra round-trip message exchanges across the network.

Solution → Caching component connections Figure 5 shows how SwapCIAO handles component connections during the component update process. During

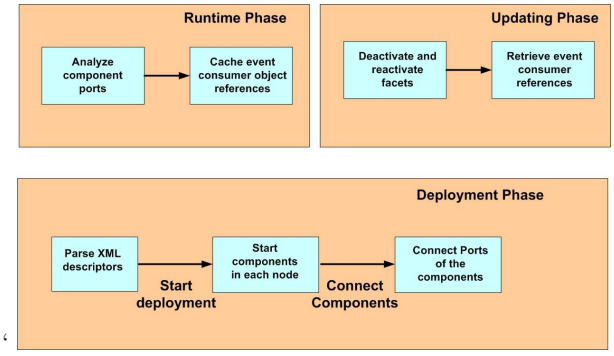

Fig. 5. Enabling (Re)connections of Components in SwapCIAO

the component updating process, SwapCIAO caches component connections to any of its peer component ports. SwapCIAO automatically handles the case where the updated component is a facet and the connected component is a receptacle. Since the receptacle could make requests on the facet while the component implementation is being updated, SwapCIAO uses the mechanisms described in Section 3.1 to deactivate the facets properly, so that no invocations are dispatched to the component. When the new component is activated, the facets are reactivated using the SwapCIAO's POA servant activator mechanism discussed in Section 3.2. For event source and event sinks, if the component

being updated is the publisher, SwapCIAO caches the connections of all the connected consumers. When the updated component implementation is reactivated, its connections are restored from the cache. As a result, communication can be started immediately, without requiring extra network overhead.

Applying the solution to ITS. In the ITS, a conveyor belt component in the warehouse hardware subsystem is connected to many sensors that assist the conveyor belt in tracking goods until they reach a storage system. When a conveyor belt component is updated, its connections to sensor components are cached before deactivation. When the updated conveyor belt component implementation is reactivated, the cached connections are restored and communication with the sensors can start immediately and all requests blocked during the update process will then be handled.

4 Empirical Results

This section presents the design and results of experiments that empirically evaluate how well SwapCIAO's dynamic component updating framework described in Section 3 addresses the requirements discussed in Section 2.2. We focus on the performance and predictability of SwapCIAO's component updating mechanisms provided by version 0.4.6 of SwapCIAO. All experiments used a single 850 MHz CPU Intel Pentium III with 512 MB RAM, running the RedHat Linux 7.1 distribution, which supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. The benchmarks ran in the POSIX real-time thread scheduling class [21] to increase the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

Figure 6 shows key component interactions in the ITS case study shown in Figure 1 that motivated the design of these benchmarks using SwapCIAO.

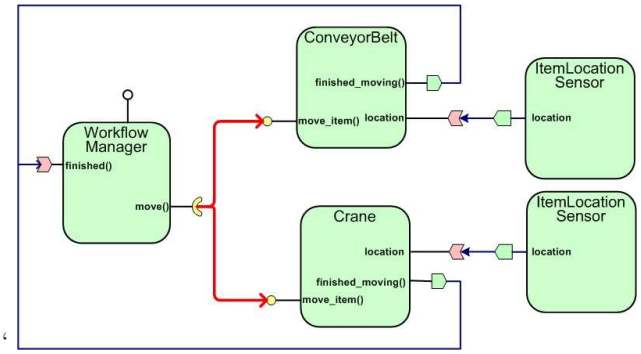

Fig. 6. Component Interaction in the ITS

As shown in this figure, the workflow manager component of the material flow control subsystem is connected to the conveyor belt and forklift transportation units of the warehouse hardware subsystem. We focus on the scenario where the

workflow manager contacts the conveyor belt component using the `move_item()` operation to instruct the conveyor belt component to move an item from a *source* (such as a loading dock) to a *destination* (such as a warehouse storage location). The `move_item()` operation takes source and destination locations as its input arguments. When the item is moved to its destination successfully, the conveyor belt component informs the workflow manager using the `finished_moving()` event operation. The conveyor belt component is also connected to various sensor components, which determine if items fall off the conveyor belt. It is essential that the conveyor belt component not lose connections to these sensor components when component implementation updates occur.

During the component updating process, workflow manager clients experience some delay. Our benchmarks reported below measure the delay and jitter (which is the variation of the delay) that workflow manager clients experience when invoking operations on conveyor belt component during the component update process. They also measure how much of the total delay is incurred by the various activities that SwapCIAO performs when updating a component implementation. In our experiments, all components were deployed on the same machine to alleviate the impact of network overhead in our experimental results.

The core CORBA benchmarking software is based on the single-threaded version of the "`TestSwapCIAO`" performance test distributed with CIAO.[4] This benchmark creates a session for a single client to communicate with a single component by invoking a configurable number of `move_item()` operations. The conveyor belt component is connected to the sensor components using event source/sink ports.

Section 3.3 describes how caching and reestablishing connections to peer components are important steps in the component updating process. We therefore measured the scalability of SwapCIAO when an updated component has upto 16 peer components using event source/sink ports. The tests can be configured to use either the standard Lightweight CCM session containers or SwapCIAO's updatable containers (described in Section 3). `TestSwapCIAO` uses the default configuration of TAO, which uses a reactive concurrency model to collect replies.

4.1 Measuring SwapCIAO's Updatable Container Overhead for Normal Operations

Rationale. Section 3 described how SwapCIAO extends Lightweight CCM and CIAO to support dynamic component updates. DRE systems do not always require dynamic component updating, however. It is therefore useful to compare the overhead of SwapCIAO's updatable container versus the standard Lightweight CCM session container under *normal operations* (*i.e.*, without any updates) to evaluate the tradeoffs associated with this feature.

Methodology. This experiment was run with two variants: one using the SwapCIAO updatable container and the other using the standard CIAO session container. In both experiemnts, we used high-resolution timer probes to measure

[4] The source code for `TestSwapCIAO` is available at `www.dre.vanderbilt.edu/~jai/TAO/CIAO/performance-tests/SwapCIAO`.

the latency of `move_item()` operation from the workflow manager component to the conveyor belt component. Since SwapCIAO caches and restores a component's connections to its peer components, we varied the number of sensor components connected to the conveyor belt and then collected latency data with 2, 4, 8, and 16 ports to determine whether SwapCIAO incurred any overhead with additional ports during normal operating mode. The `TestSwapCIAO` client made 200,000 invocations of `move_item()` operation to collect the data shown in Figure 7.

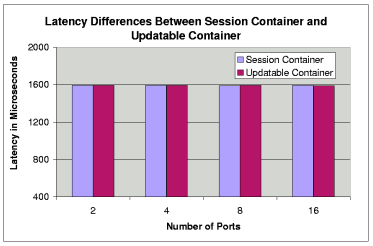

Fig. 7. Overhead of SwapCIAO's Updatable Container

Analysis of results. Figure 7 shows the comparitive latencies experienced by the workflow manager client when making invocations on conveyor belt component created with the session container versus the updatable container. These results indicate that no appreciable overhead is incurred by SwapCIAO's updatable container for normal operations that do not involve dynamic swapping.

The remainder of this section uses the results in Figure 7 as the *baseline processing delay* to evaluate the delay experienced by workflow manager clients when dynamic updating of a conveyor belt component occurs.

4.2 Measuring SwapCIAO's Updatable Container Overhead for Updating Operations

Rationale. Evaluating the efficiency, scalability, and predictability of SwapCIAO's component updating mechanisms described in Section 3.2 and Section 3.3 is essential to understand the tradeoffs associated with updatable containers. SwapCIAO's *component update time* includes (1) the *removal time*, which is the time SwapCIAO needs to remove the existing component from service, (2) the *creation time*, which is the time SwapCIAO needs to create and install a new component, and (3) the *reconnect time*, which is the time SwapCIAO needs to restore a component's port connections to its peer components.

Methodology. Since the number of port connections a component has affects how quickly it can be removed and installed, we evaluated SwapCIAO's component update time by varying the number of ports and measuring the component's:

- *Removal time*, which was measured by adding timer probes to SwapCIAO's `CCM_Object::remove()` operation, which deactivates the component servant, disassociates the executor from the servant, and calls `ccm_passivate()` on the component.

– *Creation time*, which was measured by adding timer probes to SwapCIAO's `PortableServer::ServantActivator::incarnate()` operation, which creates and installs a new component, as described in Section 3.2.
– *Reconnect time*, which was measured by adding timer probes to `CCM_Object::ccm_activate()`, which establishes connections to ports.

We measured the times outlined above whenever a component update occurs during a `move_item()` call for 200,000 iterations and then calculated the results presented below.

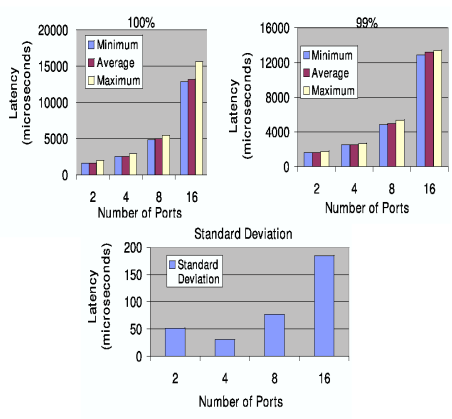

Fig. 8. Latency Measurements for Component Creation

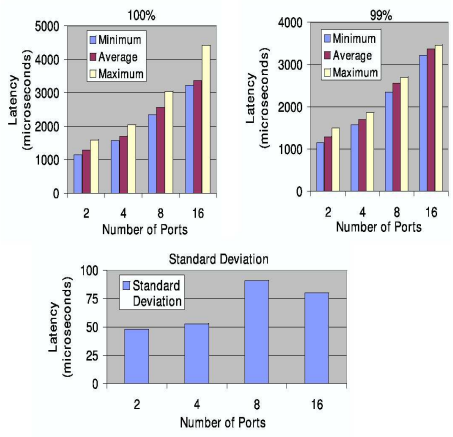

Fig. 9. Latency Measurements for Reconnecting Component Connections

Analysis of creation time. Figure 8 shows the minimum, average, and maximum latencies, as well as the 99% latency percentile, incurred by SwapCIAO's servant activator to create a new component, as the number of ports vary from 2, 4, 8, and 16. This figure shows that latency grows linearly as the number of ports initialized by `PortableServer::ServantActivator::incarnate()` increases. It also shows that SwapCIAO's servant activator spends a uniform amount of time creating a component and does not incur significant overhead when this process is repeated 200,000 times. SwapCIAO's creation mechanisms described in Section 3.2 are therefore efficient, predictable, and scalable in *ensuring efficient client-transparent dynamic component updates.*

Analysis of reconnect time. Figure 9 shows the minimum, average, and maximum latencies, as well as 99% latency percentile, incurred by SwapCIAO's reconnect mechanisms to restore a new component's connections, as the number of ports vary from 2, 4, 8, and 16. As shown in the figure, the reconnect time increases linearly with the number of ports per component. These results indicate that SwapCIAO's reconnect mechanisms described in Section 3.3 provide *efficient (re)connection of components* and do not incur any additional roundtrip delays by propagating exceptions or sending GIOP LOCATE_FORWARD messages to restore connections to components.

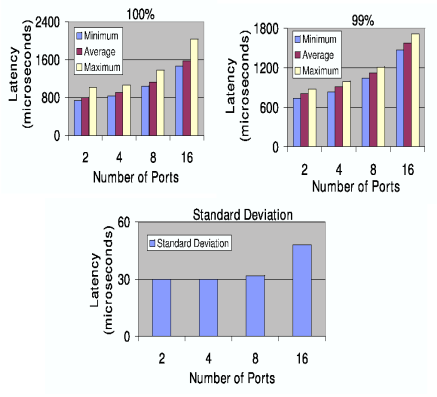

Fig. 10. Latency Measurements for Component Removal

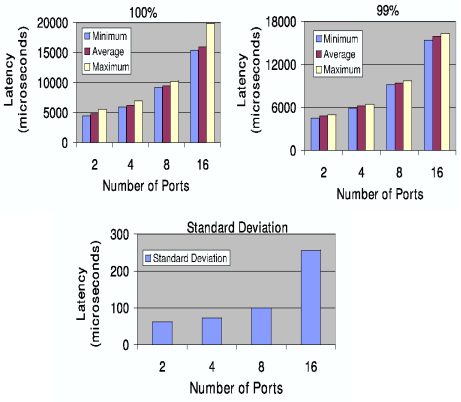

Fig. 11. Client Experienced Incarnation Delays during Transparent Component Updates

Analysis of removal time. Figure 10 shows the time used by SwapCIAO's removal mechanisms to cache a component's connections and remove the component from service, as a function of the number of its connected ports. This removal time increases linearly with the number of ports, which indicates that SwapCIAO performs a constant amount of work to manage the connection information for each port. SwapCIAO's removal mechanisms described in Section 3.1 are therefore able to *provide consistent and uninterrupted updates to clients.*

4.3 Measuring the Update Latency Experienced by Clients

Rationale. Section 3.2 describes how SwapCIAO's component creation mechanisms are transparent to clients, efficient, and predictable in performing client-transparent dynamic component updates. Section 4.2 showed that SwapCIAO's standard POA mechanisms and the servant activator create new component implementations efficiently and predictably. We now determine whether SwapCIAO incurs any overhead – other than the work performed by the SwapCIAO's component creation mechanisms – that significantly affects client latency.

Methodology. The *incarnation delay* is defined as the period of time experienced by a client when (1) its operation request arrives at a server ORB after SwapCIAO has removed the component and (2) it receives the reply after SwapCIAO creates the component, restores the component's connections to peer components, and allows the updated component to process the client's request. The incarnation delay therefore includes the *creation time*, *reconnect time*, and *processing delay* (which is the time a new component needs to process the operation request and send a reply to the client). To measure incarnation delay, we (1) removed a component and (2) started a high-resolution timer when the client invokes a request on the component. We repeated the above experiment for 200,000 invocations and measured the latency experienced by the client for each invocation. We also varied the number of ports between 2, 4, 8, and 16 as described in Section 4.2 to measure the extent to which SwapCIAO's component creation process is affected by the number of ports connected to a component.

Analysis of results. Figure 11 shows the delay experienced by a client as SwapCIAO creates a component with a varying number of connections to process client requests. By adding the delays in Figure 8, Figure 9, and Figure 7 and comparing them with the delays in Figure 11, we show how the incarnation delay is roughly equal to the sum of the creation time, reconnect time, and processing delay, regardless of whether the client invokes an operation on a updating component with ports ranging from 2, 4, 8, to 16.

These results validate our claim in Section 3.2 that SwapCIAO provides component updates that are transparent to clients. In particular, if SwapCIAO's servant activator did not transparently create the component and process the request, the client's delay incurred obtaining a new object reference would be larger than the sum of the creation time, reconnect time, and the processing delay. We therefore conclude that SwapCIAO provides efficient and predictable client transparent updates.

5 Related Work

This section compares our R&D efforts on SwapCIAO with related work ranging from offline updates to hot standby and application-specific techniques.

Offline techniques. Component updating has often been done via offline techniques, where applications are stopped to perform the update and restarted with the new implementation. For example, in [22] when a node is reconfigured, other nodes that require service from the target node are blocked completely, unnecessarily delaying services that are not being reconfigured. To minimize system interruption, [23] uses a centralized configuration manager, that oversees the interactions among components. The centralized configuration manager becomes the single point of failure and also a bottleneck for communication among components. Such techniques can be overly rigid and inefficient for certain types of DRE applications, such as online trading services and inventory tracking systems, where downtime is costly. To address these limitations, SwapCIAO updates component implementations dynamically by (1) queuing requests from other components during the component update and (2) transparently redirecting those requests to the updated implementation, thereby enabling uninterrupted online component updates.

Hot standby techniques. Another component updating technique uses online backup implementations, called "hot standbys." In this approach, when a component needs updating, requests to it will be transferred to the backup, during which the main implementation is updated [24]. Although this solution is common, it can be complex and resource-intensive. In particular, when adding backup implementations to resource-constrained DRE systems, such as satellite and avionics mission computing systems, it can be unduly expensive and complicated to keep the backup implementation updated and to reroute requests to this standby when the main implementation is being updated. To address these limitations, SwapCIAO does not run a backup implementation and instead updates

implementations dynamically. Although requests to the target component are queued during the update, no round-trip overhead is incurred to redirect client requests from one node to another. Moreover, queued requests in SwapCIAO are redirected transparently to the updated implementation once it is activated.

Application-specific techniques. Another technique employs application-specific modifications to handle component updates. For example, [25] introduces a component configurator that performs reconfiguration at the application level. As a result, application developers must implement a configurator for each component. Moreover, handling connections among components is hard since there is no central entity managing information about the overall DRE system structure. To address these issues, SwapCIAO leverages patterns (such as Reference Counting and Dispatching [19], Wrapper Facade [10] and Component Configurator [10]) and frameworks (such as Portable Interceptors [20] and ACE Service Configurator [26]) to implement the dynamic component updating capability in the middleware. Application developers are therefore able to focus on their component implementations, rather than wrestling with complex mechanisms needed to add dynamic component updating capabilities into their applications.

6 Concluding Remarks

This paper describes the design and implementation of SwapCIAO, which is a QoS-enabled component middleware framework based on Lightweight CCM that supports dynamic component updating. SwapCIAO is designed to handle dynamic operating conditions by updating component implementations that are optimized for particular run-time characteristics. The lessons learned while developing SwapCIAO and applying it to the ITS case study include:

- Standard Lightweight CCM interfaces can be extended slightly to develop a scalable and flexible middleware infrastructure that supports dynamic component updating. In particular, SwapCIAO's extensions require minimal changes to the standard Lightweight CCM server programming model. Moreover, its client programming model and client/server interoperability were unaffected by the server extensions. Developers of client applications in our ITS case study were therefore shielded entirely from SwapCIAO's component updating extensions.
- By exporting component implementations as DLLs, SwapCIAO simplifies the task of updating components by enabling their implementations to be linked into the address space of a server late in its lifecycle, *i.e.*, during the deployment and reconfiguration phases. These capabilities enabled developers in the ITS case study to create multiple component implementations rapidly and update dynamically in response to changing modes and operational contexts.
- SwapCIAO adds insignificant overhead to each dynamic component updating request. It can therefore be used even for normal operations in ITS applications that do not require dynamic component updating. Moreover, due to

the predictability and transparency provided by SwapCIAO, it can be used efficiently when operating conditions trigger mode changes.

Our future work will focus on developing selection algorithms [27] that can automatically choose the most suitable component implementation to update in a particular operating condition. We will implement these selection algorithms and validate them in the context of DRE systems, such as our ITS case study. To enhance the autonomic properties of DRE systems, we are developing a monitoring framework within SwapCIAO that (1) observes the performance of different components, (2) identifies when performance is not within the desired QoS bounds, and (3) automatically updates component implementations using our selection algorithms.

References

1. Schmidt, D.C., Schantz, R., Masters, M., Cross, J., Sharp, D., DiPalma, L.: Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. CrossTalk (2001)
2. Nechypurenko, A., Schmidt, D.C., Lu, T., Deng, G., Gokhale, A., Turkay, E.: Concern-based Composition and Reuse of Distributed Systems. In: Proceedings of the 8th International Conference on Software Reuse, Madrid, Spain, ACM/IEEE (2004)
3. Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003. (2003)
4. Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., Duzan, G.: Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems. In: Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04), Agia Napa, Cyprus (2004)
5. Wang, N., Gill, C.: Improving Real-Time System Configuration via a QoS-aware CORBA Component Model. In: Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)
6. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus (2004)
7. Ritter, T., Born, M., Unterschütz, T., Weis, T.: A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In: Proceedings of the 36^{th} Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)
8. Roll, W.: Towards Model-Based and CCM-Based Applications for Real-Time Systems. In: Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Hakodate, Hokkaido, Japan, IEEE/IFIP (2003)
9. Balasubramanian, K., Schmidt, D.C., Wang, N., Gill, C.D.: Towards Composable Distributed Real-time and Embedded Software. In: Proc. of the 8^{th} Workshop on Object-oriented Real-time Dependable Systems, Guadalajara, Mexico, IEEE (2003)

10. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. Wiley & Sons, New York (2000)
11. Wang, N., Schmidt, D.C., Kircher, M., Parameswaran, K.: Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. IEEE Distributed Systems Online **2** (2001)
12. Wang, N., Schmidt, D.C., Parameswaran, K., Kircher, M.: Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. In: 24th Computer Software and Applications Conference, Taipei, Taiwan, IEEE (2000)
13. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In: Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Sym., San Francisco, CA (2005)
14. Object Management Group: Light Weight CORBA Component Model Revised Submission. OMG Document realtime/03-05-05 edn. (2003)
15. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
16. Object Management Group: Real-time CORBA Specification. OMG Document formal/02-08-02 edn. (2002)
17. Nechypurenko, A., Schmidt, D.C., Lu, T., Deng, G., Gokhale, A.: Applying MDA and Component Middleware to Large-scale Distributed Systems: a Case Study. In: Proceedings of the 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application, Enschede, Netherlands, IST (2004)
18. Schmidt, D.C., Vinoski, S.: The CORBA Component Model Part 3: The CCM Container Architecture and Component Implementation Framework. The C/C++ Users Journal (2004)
19. Pyarali, I., O'Ryan, C., Schmidt, D.C.: A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware. In: Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), Newport Beach, CA, IEEE/IFIP (2000)
20. Wang, N., Schmidt, D.C., Othman, O., Parameswaran, K.: Evaluating Meta-Programming Mechanisms for ORB Middleware. IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies **39** (2001) 102–113
21. Khanna, S., *et al.*: Realtime Scheduling in SunOS 5.0. In: Proceedings of the USENIX Winter Conference, USENIX Association (1992) 375–390
22. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering **SE-16** (1990)
23. Bidan, C., Issarny, V., Saridakis, T., Zarras, A.: A Dynamic Reconfiguration Service for CORBA. In: International Conference on Configurable Distributed Systems (ICCDS '98). (1998)
24. Tewksbury, L., Moser, L., Melliar-Smith, P.: Live upgrades of CORBA applications using object replication. In: International Conf. on Software Maintenance, Florence, Italy (2001) 488–497
25. Kon, K., Campbell, R.: Dependence Management in Component-based Distributed Systems. IEEE Concurrency **8** (2000)
26. Schmidt, D.C., Huston, S.D.: C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks. Addison-Wesley, Reading, Massachusetts (2002)

27. Yellin, D.M.: Competitive algorithms for the dynamic selection of component implementations. IBM Systems Journal **42** (2003)