The Design and Performance of Configurable Component Middleware for Distributed Real-Time and Embedded Systems *

Venkita Subramonian, Liang-Jui Shen, and Christopher Gill
{venkita,ls1,cdgill}@cse.wustl.edu
Dept. of Computer Science and Engineering
Washington University, St. Louis, MO

Nanbor Wang
nanbor@txcorp.com
Tech-X Corp
Boulder, CO

Abstract

QoS-enabled component middleware solutions can help reduce the programming complexity of configuring real-time aspects, such as priorities and rates of invocation. However, few empirical studies have been conducted to guide distributed real-time and embedded (DRE) system developers in choosing among alternative configuration mechanisms and performance optimization techniques in practice.

This paper makes three contributions to research on QoS-enabled component middleware for DRE systems in the context of the Component-Integrated ACE ORB (CIAO). First, it describes the design of CIAO's static component configuration mechanisms, which enhance configurability by avoiding features that are not supported by key real-time platforms, while reducing run-time overhead and footprint. Second, it compares the performance of dynamic and static configuration mechanisms in CIAO to help guide the selection of suitable configuration mechanisms based on specific requirements of each DRE system. Third, it presents an empirical comparison of CIAO's static configuration mechanisms to the static configuration mechanisms in Boeing's PRISM avionics component middleware solution.

Keywords: QoS-enabled component middleware, DRE systems, configuration mechanisms.

1 Introduction

As DRE systems have increased in scale and complexity over the past decade, a variety of middleware technologies have been developed to address the challenges those systems pose. A key challenge posed by large-scale complex DRE systems is the tension between stringent performance requirements and the ease with which those systems can be developed and configured to meet those requirements. This section first summarizes the ongoing evolution of middleware technologies to allow developers to achieve suitable DRE system performance while meeting the increasingly stringent system development, deployment, and upgrade cycles demanded by the economics of modern system development. It then describes several remaining limitations of state-of-the-art middleware technologies and explains how our work in this paper addresses those limitations.

1.1 Evolution of Middleware Configuration Capabilities for DRE Systems

The following four stages in the evolution of middleware provide the context for this paper:

1. Distributed object computing (DOC) middleware. Conventional commercial-off-the-shelf (COTS) DOC middleware technologies, such as The Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [1], Sun's Java RMI [2], and Microsoft's COM+ [3], have matured considerably over the past decade. These technologies significantly reduce the complexity of writing client programs by providing an object-oriented programming model for distributed systems that separates application-level code from reusable system-level code that handles sockets, threads, and other network programming mechanisms.

Unfortunately, conventional DOC middleware technologies have two major limitations for use in DRE systems:

1. **Separation of application-level and system-level concerns is incomplete** due to a lack of generic standards for installing objects within server environments, assembling objects into applications, and configuring and deploying applications [4]. In practice, this tight coupling results in object implementations that are unnecessarily inter-dependent and applications that are excessively hard to deploy, maintain, and extend.
2. **Only functional and not systemic QoS concerns are addressed.** While DOC middleware addresses *functional* aspects, such as how to define and integrate object interfaces and implementations, these technologies do not address key DRE *systemic* QoS aspects, such

* This work was supported in part by the DARPA PCES program under contracts F33615-01-C-3048 and F33615-03-C-4111.

as how to define and enforce operation deadlines, resource access priorities, or rates of method invocations. In practice, this means that additional code for managing these systemic QoS aspects must be tangled within the application programs, making DRE systems unnecessary brittle and hard to evolve.

2. Conventional component middleware. Conventional component middleware technologies such as the CORBA Component Model (CCM) [5], J2EE [6] and COM+ [3] overcome limits with overly coupled DOC middleware by (1) providing mechanisms that automate common middleware idioms, such as interface navigation and event handling, (2) defining containers to encapsulate common component functionality, such as security, transactions, and load balancing, and (3) dividing system development and configuration concerns into separate aspects, such as implementing application functionality, defining component metadata, and configuring resource management policies.

Conventional component middleware technologies do not, however, adequately address the systemic QoS limitations of DOC middleware, since they were designed largely to support enterprise applications and cannot handle the more stringent QoS needs of DRE systems. Further evolution of conventional component middleware technologies was therefore needed to address the broad range of DRE system requirements.

3. Real-time DOC middleware. Real-time DOC middleware technologies, such as Real-Time CORBA (RTCORBA) [7, 8] and the Real-Time Specification for Java [9], address key systemic QoS aspects in DRE systems. Interfaces and mechanisms provided by these technologies support explicit configuration of systemic QoS aspects, such as the priorities of threads invoking object methods. These technologies, however, still suffer from the first limitation of conventional DOC middleware, *i.e.*, they do not provide configuration support by component middleware, which again causes unnecessary tangling of code for managing systemic QoS aspects with application logic.

4. QoS-enabled component middleware. To address the limitations with earlier middleware techniques for use in DRE systems, a new generation of *QoS-enabled component middleware* has emerged that combines the capabilities of conventional component middleware and real-time DOC middleware. One such technology is the Component Integrated ACE ORB (CIAO) [10], which is a QoS-enabled middleware framework based on the CCM specification that allows declarative specification of RTCORBA [11] features. CIAO is built atop The ACE ORB (TAO) [12], which is a widely used ORB that implements the Real-Time CORBA 1.0 [7] and 1.2 (formerly named 2.0 [8]) specifications. Both CIAO and TAO are available as open-source software [13] and can be obtained from `deuce.doc.wustl.edu/Download.html`.

1.2 Motivation and Contributions

Our previous work on CIAO [14] focused on supporting declarative configuration of real-time aspects [15], conducting empirical studies [16] to compare the performance of those aspects in CIAO to their performance in TAO, and examining how configuring aspects at different stages of the system lifecycle can improve performance in comparison to real-time middleware approaches [17]. This work has been concerned mainly with configuration and performance of the real-time aspects, rather than the performance of the configuration mechanisms themselves. The remainder of this section describes key remaining limitations with state-of-the-art in QoS-enabled component middleware and outlines this paper's contributions to address those limitations.

Limitations with the state-of-the-art. Our research on configuration mechanisms for QoS-enabled component middleware is motivated by the following limitations with the current state-of-the-art in middleware technologies:

- Although our previous work has made CIAO suitable for many DRE systems, some DRE systems have additional constraints on system initialization times, available features (*e.g.*, dynamic linking and loading), and memory usage, which the current generation of QoS-enabled component middleware does not address. For example, (re)initialization time can be significant if a system must be rebooted or reconfigured while it is in service.
- Few quantitative comparisons have been made between alternative configuration mechanisms in terms of their flexibility *and* performance.
- Few empirical case studies have been conducted to compare the effectiveness of standards-based vs. domain-specific QoS-enabled component models.

Contributions of this research. This paper extends our previous work on CIAO to address the limitations with the state-of-the-art in QoS-enabled component middleware, as follows:

- We describe the design and implementation of CIAO's static configuration mechanisms and compare them to its original dynamic configuration mechanisms. CIAO now supports both dynamic and static configuration mechanisms to better meet the requirements of a wide range of DRE systems. In addition to issues of static vs. dynamic linking/loading [18], this paper considers a wider range of issues relevant to component middleware, *e.g.*, configuration parsing, parameter modification, and component assembly.
- It quantitatively compares CIAO's dynamic and static component configuration mechanisms with respect to the requirements of different classes of DRE systems.

The resulting performance profiles and analysis of configuration flexibility help guide DRE system developers in choosing which configuration mechanisms to use for particular DRE systems.

- It presents an empirical case study that compares CIAO's configuration mechanisms to those in PRISM [19], which is an avionics domain-specific component model developed by Boeing. This case study also helps guide DRE system developers on the trade-offs in performance and flexibility between applying standards-based QoS-enabled component models vs. using domain-customized solutions.

Paper organization. The remainder of this paper is organized as follows: Section 2 summarizes how RTCORBA policies and mechanisms can be configured in CIAO and describes a representative example drawn from the avionics domain that is used throughout this paper; Section 3 describes the design and implementation of dynamic and static configuration mechanisms in CIAO; Section 4 presents the results of empirical studies conducted to quantify the relative performance of CIAO's dynamic and static configuration mechanisms; Section 5 presents an empirical case study comparing the design, performance, and flexibility of static configuration mechanisms in CIAO with the avionics domain-specific PRISM component model developed by Boeing; Section 6 describes related work in the areas of DRE system configuration tools and QoS-enabled component models; and Section 7 presents concluding remarks.

2 Configuring DRE Aspects in CIAO

Compared to conventional desktop and enterprise applications, DRE systems have more stringent QoS requirements that must be satisfied simultaneously at run-time. Examples of such QoS requirements include end-to-end latency of component method invocations, availability of CPU cycles to meet computation deadlines, and rates of invocation of component methods. To ensure that DRE systems can meet their QoS requirements, various *QoS configuration* activities must be performed to allocate and manage system computing and communication resources end-to-end. These QoS configuration activities can be performed in the following ways [20]:

- *Dynamically*, where the resources required are determined and adjusted based on the runtime system status. Examples of dynamic QoS configuration include bandwidth reallocation or task re-prioritization to handle bursty system load.
- *Statically*, where ensuring adequate resources required to support a particular degree of QoS is pre-configured into an application. Examples of static QoS configuration include task prioritization and bandwidth reservations configured off-line, and initialized through low-latency driver code at system initialization.

To configure end-to-end QoS robustly throughout a DRE system and to improve component reusability, QoS configuration specifications should be decoupled as much as possible from component implementations. Instead, configurations should be specified using component composition metadata, which consists of configuration information (*e.g.*, in XML) for CPU and communication resource allocations apart from descriptors that specify the interfaces of each component and the connections between components.

2.1 CIAO Configuration Overview

To address the dual challenges of system performance and configuration flexibility, CIAO extends the component container definition and metadata representation and manipulation capabilities found in conventional component middleware [14]. For example, CIAO allows configuration of the RTCORBA priority model policies, RTCORBA threading policies, and invocation rates that are relevant to the example application described in Section 2.2, and are exploited in the experiments described in Sections 4 and 5.2. RTCORBA priority model and threading policies manage resources in RTCORBA 1.0 ORBs [21]. Likewise, method invocation rates on component facets, along with rates of event pushes to component event sinks, determine the rates at which operations are executed within the specified components, and implicitly within other connected components.

To enhance configuration flexibility, CIAO's dynamic configuration process allows developers to configure real-time policies at multiple stages of the component-oriented application development lifecycle, *e.g.*, component implementation, component packaging, application assembly, and application deployment. Although these stages are condensed in CIAO's static configuration process (Section 3.2), similar actions are performed in both approaches.

Compared to using conventional real-time DOC middleware, more steps are seemingly required to develop the same application using CIAO. Much of this complexity, however, is *inherent* to the process of developing and deploying DRE systems. CIAO alleviates other *accidental complexities* that can arise when developing DRE systems using either (1) conventional real-time middleware directly or (2) component model implementations that do not explicitly support configuring real-time systemic aspects.

2.2 Example Application

To examine how CIAO's configuration capabilities can be applied to real-world DRE systems, we now describe a representative example drawn from the avionics domain [22]. Figure 1 illustrates a basic single-processor (Basic SP) sce-

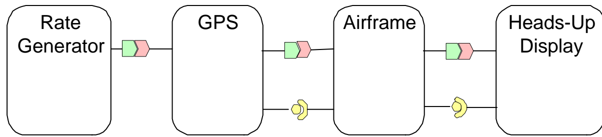

Figure 1: **Example (Basic SP) Scenario**

nario involving four software components: (1) a **Rate Generator** component that wraps a timer that triggers pushing of events at specific periodic rates to event consumers that register for those events, (2) a **GPS** component that wraps one or more hardware devices for navigation, (3) an **Airframe** component that wraps the avionics airframe hardware, and (4) a **Head-up Display** component that wraps the hardware for a display device in the cockpit. When the GPS component receives a triggering event from the Rate Generator component it refreshes its location from the navigation hardware device and caches this value. The GPS component then pushes a triggering event to the Airframe component, which in turn pulls the new value from the GPS component. The Airframe component next pushes the triggering event to the Heads-up Display component, which in turn pulls the new value from the Airframe component and updates its displays in the cockpit.

In practice, production DRE systems based on QoS-enabled component middleware [19] often contain a large number (*i.e.*, hundreds to thousands) of components, with subsets of components connected via specialized networking devices, such as VME buses. Although applications and their real-time requirements and deployment environments may differ, however, many DRE systems share the kinds of rate-activated computation and display/output QoS constraints illustrated by the Basic SP example described above. This example therefore represents a broader class of systems to which our work on CIAO applies.

The Basic SP example also helps to illustrate the benefits of developing DRE systems with CIAO, instead of with RTCORBA directly or with component model implementations that do not support explicit fine-grain configuration of key QoS aspects. With CIAO's extensions [14] to the CCM development paradigm, extending the example application shown in Figure 1 simply involves implementing new component, packaging them with XML descriptors of their interfaces and QoS constraints, and then using this package metadata to compose new DRE systems via an XML assembly file. In contrast, a direct RTCORBA implementation of the same application would require more effort, *e.g.*, (1) modifying code in each component to configure ORBs and their object adapters to accommodate the QoS requirements of their servants, (2) activating the servants, (3) setting up QoS attributes of the connections between the servants, and (4) modifying how the servants interact.

3 CIAO Configuration Mechanisms

This section describes the process of assembling an application using CIAO. We first describe the dynamic assembly of components, where component implementations are loaded from dynamically linked libraries. We then describe the limitations with this approach in the context of DRE systems and explain how we overcome these drawbacks by using a static configuration approach that is better suited to the stringent memory and performance constraints of DRE systems. Irrespective of whether configuration is dynamic or static, CIAO allows different functional and real-time policies and mechanisms to be configured in each of the following canonical steps of its overall configuration process:

1. Create the *component server* environment within which homes and containers reside,
2. Create *home* factories for the component containers,
3. Create *containers* for the components,
4. Create the *components* themselves,
5. *Register* components, and
6. Establish *connections* between components.

We use the relative latency of each of these steps to compare the performance of CIAO's dynamic and static configuration mechanisms in Section 4. These steps are also used in Section 5 to compare CIAO's configuration mechanisms to Boeing's PRISM component model.

3.1 Dynamic Assembly of Components

CIAO's process for dynamically assembling application components is shown in Figure 2. The first stage of the

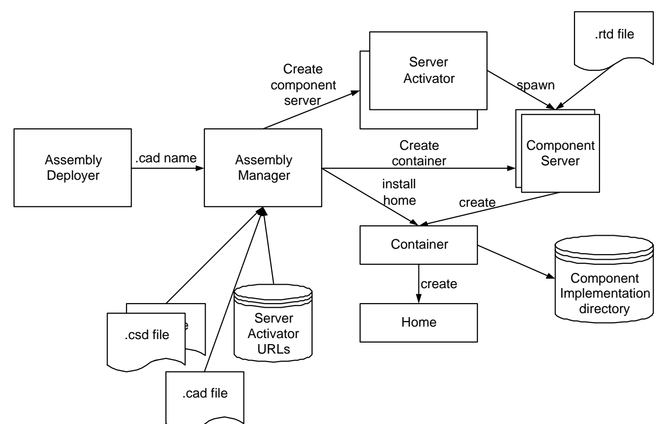

Figure 2: **Dynamic Component Assembly in CIAO**

CIAO system lifecycle occurs off-line, when component package (.csd) and assembly (.cad) files are generated by the application assembler, often via modeling tools such as CoSMIC [23]. These files contain an abstract specification

of the configuration that CIAO must create for each deployment environment. The real-time descriptor (.rtd) file contains specifications for configuring real-time policies and mechanisms (*e.g.*, in RTCORBA).

CIAO's dynamic configuration mechanisms process these .csd, .cad, and .rtd files to create and configure the appropriate components, their run-time server environments, and QoS properties within the supporting ORB infrastructure. CIAO's dynamic configuration framework currently runs several daemon processes for each deployment environment, including

- One or more Component Installation/Server Activation (CISA) daemons on each machine where applications can be deployed. Each CISA daemon represents the deployment target and manages the additional information about available component implementations on that endsystem. Each component implementation is identified by a universally unique identifier (UUID). A CISA daemon manages its component implementations by mapping each UUID it knows to the actual component implementation specific to the platform. Furthermore, a CISA daemon is responsible for spawning component server processes and managing server configuration information.
- An Assembly Manager daemon that manages the target deployment topology, as a collection of available named CISA daemon references which the Assembly Manager can contact to deploy applications over the deployment targets.
- An Assembly Deployer process that tells the Assembly Manager which component assemblies (each assembly is defined in a separate .cad file) should be deployed on which target machines.

The Assembly Manager parses the XML structures in the .cad file, and generates its own internal data structure as an intermediate representation of that assembly. The Assembly Manager then traverses this intermediate representation, instructing each CISA daemon to install/configure specific component servers and containers, to create specific homes, and to instantiate specific component instances.

Although CIAO offers a highly flexible environment for system configuration, the dynamic assembly approach described above suffers from the following drawbacks for DRE systems with stringent performance constraints:

- XML parsing may be too expensive to be performed during system (re)initialization.
- Multiple process address spaces may be required to coordinate the creation and assembly of components.
- Online loading of component implementations may not be possible on real-time OS platforms, such as VxWorks, where facilities such as dynamically linked libraries are not available.

3.2 Static Assembly of Components

To address the limitations of dynamic component assembly described in Section 3.1, we have implemented an alternative static approach where as much of the configuration lifecycle as possible is performed off-line. The fundamental intuition in understanding our static configuration approach is that stages of the overall DRE system configuration lifecycle similar to those in the dynamic approach must still be supported. In our static configuration approach, however, several stages of the lifecycle are *compressed* into the compile-time and system-initialization phases, so that (1) for testing and verification purposes the set of components in an application can be identified and analyzed before runtime and (2) overheads for run-time operation following initialization are reduced and made more predictable.

Due to the nuances of the platforms traditionally used for deploying DRE systems, not all features of conventional platforms are available or usable. In particular, dynamically linked libraries and on-line XML parsing are often either unavailable or too costly in terms of performance. We therefore take the existing configuration phases from the dynamic configuration approach described in Section 3.1 and push several of them earlier in the configuration lifecycle. We also ensure that our approach can be realized on highly constrained RTOS platforms, such as VxWorks, by re-factoring the configuration mechanisms to use only mechanisms that are available and efficient.

In CIAO's static configuration approach, illustrated in Figure 3, the same automated build processes that manage compilation of the application and middleware code first insert a code generation step just before compilation. In this

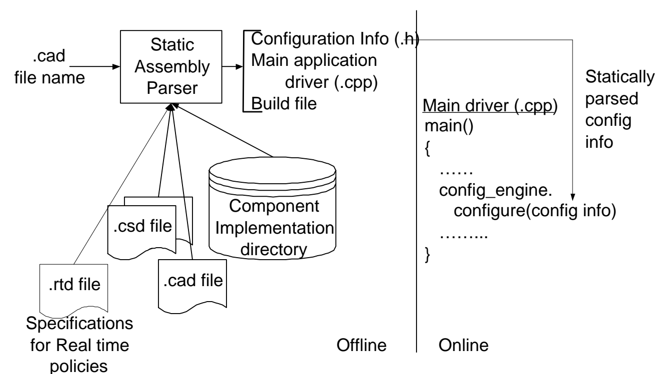

Figure 3: **Static Component Assembly in CIAO**

step, CIAO's XML configuration files (.csd, .cad, .ssd) are translated into C++ header files that are then compiled into efficient run-time configuration drivers, which in turn are linked statically with the main application. These run-time drivers are invoked during application (re)initialization at run-time, and rapidly complete the remaining on-line system configuration actions at that point.

With CIAO's static configuration approach, all XML

parsing is moved before run-time to the off-line build process, and all remaining information and executable code needed to complete configuration is linked statically into the application itself. Each endsystem can then be booted and initialized within a single address space, and there is no need for inter-process communication to create and assemble components. Our static configuration approach in CIAO thus maintains a reasonable degree of configuration flexibility, while reducing the run-time cost of configuration, as shown by the empirical results presented in Section 4.

4 Empirical Comparison of Dynamic and Static Configuration

To evaluate CIAO's dynamic and static configuration mechanisms described in Section 3, we used the Basic SP scenario described in Section 2.2. This section describes experiments conducted to quantify the performance improvement that CIAO's static configuration mechanisms offer with respect to initialization time and footprint, compared to its dynamic configuration mechanisms. These experiments are conducted both with and without real-time extensions (.rtd file), which we term RTCIAO and CIAO, respectively. Table 1 summarizes our results.

Experiment testbed. Experiments to compare the dynamic and static configuration mechanisms described in Sections 3.1 and 3.2 must be conducted on a platform that (1) supports necessary dynamic configuration mechanisms, such as dynamically linked libraries, and (2) offers suitable real-time performance. We therefore chose KURT-Linux [24] as our experimental platform since it provides both of these capabilities. Our experiments comparing dynamic and static configuration were performed with the Basic SP application running atop CIAO 0.4.1 on a Pentium-IV 2.5 GHz machine with 500 MB RAM, 512 KB cache, running KURT-Linux 2.4.18. Our timing measurements used a timer implementation based on the the Pentium time stamp counter (incremented at the processor's rate) to obtain nanosecond resolution.

Time for assembly. Assembly time includes the time to create the server, homes, containers and components and to establish registrations of, and connections between, the components. Application assembly with the static configuration approach takes almost two orders of magnitude less time than with the dynamic approach. This improvement is not surprising since at run-time the dynamic configuration approaches parses XML files and also loads shared objects containing component implementations, both of which are achieved off-line in the static approach.

The assembly times involving configuration of real-time aspects (RTCIAO) are higher than those without real-time aspects (CIAO). The reason for this is that with real-time descriptors, CIAO must also create RTCORBA thread pools, lanes, and threads at run-time in both approaches. We now compare the individual segments of the configuration process to determine which segments contribute the most to the longer assembly times seen with the dynamic approach.

Component server creation. Our results in Table 1 show that this stage contributes the most to the delay observed in the dynamic approach, which is consistent with our expectations based on the discussion of the dynamic and static configuration mechanisms in Section 3. Specifically, in this stage a separate component server process is spawned in the dynamic approach, whereas in the static approach a component server object is created in-process at the beginning. Spawning a separate process incurs significant overhead, as seen in the performance of the dynamic approach.

Home creation. The creation time for component homes is significantly higher in the dynamic approach, which again is not surprising because loading dynamically linked libraries is relatively expensive.

Container creation. We attribute differences in container creation times between RTCIAO and CIAO to the different real-time and non-real-time container implementations. Creation time is slightly higher in the dynamic approach.

Component creation. The dynamic approach takes slightly more time than the static approach in this case. We believe this occurs because in the dynamic approach the component implementations are packaged in shared object libraries and hence a greater overhead is incurred to load these libraries into memory.

Component registration. In component registration, the object reference to a component is published to, *e.g.*, a naming service or a disk file, as specified in the XML descriptor files. In our experiments, the object reference to the Rate Generator component was stored in a disk file.

As Table 1 shows, the time the static CIAO mechanisms took to write a component object reference to a file was slightly lower *with* configuration of real-time features than without. We divided the code sequence for component registration into different segments and measured the duration through each of these segments, again using high resolution (nsec) timers. The results of these measurements are shown in Table 2. These micro-benchmarks revealed that the creation of the stringified interoperable object reference (IOR) contributed the most to the difference ($\sim$0.03 msec) between static CIAO and static RTCIAO. We attribute this difference to the fact that static CIAO used a POA, while static RTCIAO used an RTPOA, to create the IOR.

Connection establishment. We note that the time differences between the dynamic and static versions seen for container creation, component creation, component registration and connection creation are relatively small compared to

	Static CIAO	Static RTCIAO	Dynamic CIAO	Dynamic RTCIAO
Assembly	2,320 / 2,449 / 2,362	7,684 / 7,882 / 7,789	279,936 / 283,252 / 281,004	356,641 / 361,122 / 358,231
Component creation	121 / 197 / 161	154 / 759 / 514	1,585 / 1,896 / 1,624	1,871 / 2,544 / 2,036
Component Server creation	102 / 125 / 105	12,837 / 13,086 / 12,931	123,843 / 126,647 / 125,407	180,170 / 182,589 / 180,988
Connection creation	59 / 91 / 68	323 / 786 / 513	1,117 / 1,615 / 1,374	1,389 / 2,205 / 1,696
Container creation	243 / 282 / 250	219 / 395 / 303	2,981 / 3,081 / 3,008	1,069 / 2,935 / 1,999
Home creation	134 / 188 / 152	160 / 597 / 325	21,028 / 28,212 / 24,165	25,065 / 36,225 / 29,629
Component Registration	154 / 177 / 157	129 / 134 / 131	136 / 170 / 141	138 / 175 / 144

Table 1: **Min / Max / Mean Time in usec**

those seen for component server and home creation. We attribute the differences in container creation and connection creation to the XML parsing overhead incurred by the dynamic approach. In particular, the dynamic approach uses the visitor pattern to traverse the parsed XML data structure, which incurs slightly more overhead at each step when compared to the simple loop constructs used by the static approach to traverse information stored in C++ arrays.

Segment	CIAO (nsec)	RTCIAO (nsec)
object pointer instantiation	30	30
object pointer assignment	225	152
switch-case selection	34	34
stringified IOR creation	47,373	12,365
storing IOR string to file	109,621	108,590

Table 2: **Component Registration Segments**

Footprint measurements. One final comparison we made between the static and dynamic configuration mechanisms in CIAO was the fixed disk and memory storage needed for each approach. In future work, we plan a more complete study of the footprint and real-time performance trade-offs in CIAO (similar to one we have done for the nORB small footprint real-time ORB [25]), which will require further refactoring and finer-grained experiments. Our preliminary results, however, measured a notable improvement in footprint with the static approach, compared to the dynamic approach. The Basic SP example required 19,468 KB of disk space and 14,705 KB of memory with dynamic configuration, versus 7,612 KB and 4,947 KB respectively with static configuration. These disk and memory usage values were collected using the *size* command. The static configuration approach showed a nearly 11.5 MB (60%) reduction in the disk storage, and a 9.7 MB (66%) reduction in the memory required.

5 Case Study: Comparison of Static Configuration in CIAO and PRISM

This section compares the design, implementation, and performance of CIAO's static configuration mechanisms described in Section 3.2, to similar mechanisms in PRISM [19], which is an avionics domain-specific component model developed by Boeing. We note that both CIAO and PRISM share the same TAO middleware infrastructure. We first compare and contrast CIAO and PRISM configuration steps and then present an empirical performance comparison of CIAO and PRISM using the Basic SP scenario described in Section 2.2.

5.1 Configuration in CIAO and PRISM

Figure 4 illustrates the different steps involved in component assembly using CIAO and PRISM. In the experiments described in Section 5.2, we compare similar individual stages of the two models. In addition to the configuration activities shown in Figure 4 and examined in Section 5.2, CIAO also creates a server object and container objects. The PRISM component model also includes a number of other configuration activities beyond those examined in Section 5.2, including but not limited to initialization of services like persistence, distribution and concurrency. We focus only on a common segment of the CIAO and PRISM configuration processes containing the configuration activities in the component assembly stage that are comparable between CIAO and PRISM, and hence consider initialization of other PRISM services and creation of CIAO server and container objects to be out of scope for the purposes of this discussion.

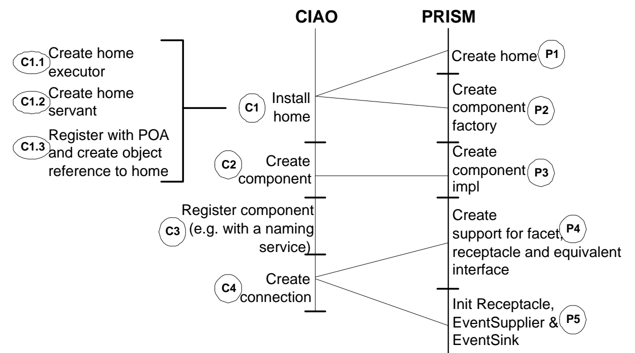

Figure 4: **Correspondence between CIAO and PRISM**

Assembly steps in CIAO. The following steps are performed in the assembly of components in CIAO. First, a home object is created (C1) and is installed on a container

object created earlier. This involves three substeps - A home executor object is created (C1.1) and a home servant object is created (C1.2). The home servant object is then registered with the POA (C1.3) and an object reference is created that can then be used to create components using the home. We note that, in contrast with the PRISM component model, many more CORBA objects are created in CIAO.

The next step (C2) creates components using the home object reference created in the previous step. A component's object reference is then advertised, *e.g.*, through registration with a naming service (C3). Step is C3 optional and is done only if it is specified in the assembly descriptor in CIAO. Since PRISM does not perform this action we omit step C3 from further consideration.

Finally, connections are established (C4) between matching publisher and consumer ports and facets and receptacles respectively, according to the connection specifications in the descriptor files. The CIAO implementation currently does not use the TAO Real-Time Event Channel (RTEC) to establish connections between publisher and consumer ports. The connection is instead achieved through a plain two-way call mechanism.

Assembly steps in PRISM. The following steps are performed in the assembly of components in PRISM. A home object is first created (P1) for each component. The home is then responsible for creating a component factory (P2) for that component. Each component's factory then creates the component implementation (P3).

Within the component implementation, the facets, receptacles and equivalent interfaces are created (P4) so that connections can be made from/to other components. Finally the connections between the facets and receptacles are established (P5). In step P5, connections are also made between event suppliers and sinks. In PRISM, a connection between an event supplier and an event sink is established by means of the TAO Real-Time Event Channel (RTEC). These correspond to the "publishes" and "consumes" ports in CCM, though as noted above the CIAO implementation currently does not use the RTEC to connect a publisher and consumer. For our comparisons, hence, we do not take into account the connections established by means of the RTEC. We also note that most of the objects created in these steps are C++ objects rather than CORBA objects as in CIAO.

5.2 Evaluation of CIAO and PRISM

Comparison of open standards-based component models to domain-specific component models must be made on the platform for which the domain-specific component model implementation was designed. We therefore used a PowerPC board running VxWorks 5.4.2 to evaluate the performance of static configuration mechanisms in CIAO and PRISM. We instrumented the relevant parts of the PRISM component model that was provided in Boeing PCES OEP release 3.0. For these experiments we again used the Basic SP application shown in Figure 1 and described in Section 2.2.

	Static CIAO	PRISM
Home creation	2,507 / 6,477 / 3,507	107 / 491 / 283
Component creation	2,708 / 5,615 / 3.691	128 / 357 / 151
Connection establishment	1,736 / 2,017 / 1,812	44 / 337 / 101

Table 3: **Min / Max / Mean Time in usec**

A key difference between CIAO and PRISM is that CIAO provides distinct server and container objects, whereas PRISM does not. In terms of performance, the server and container creation overheads seen with CIAO's static configuration mechanisms in Section 4 are avoided in PRISM. However, this incremental improvement in performance comes at a cost of some flexibility in that component implementations are as a result more tightly coupled to the details of their server environment and supporting middleware services. The results of our experiments are summarized in Table 3. For all cases, PRISM performs better than CIAO.

Experiment testbed. The experiments comparing CIAO and PRISM static configuration were run on a Motorola 5110-2263 VME board with a MPC7410 500 MHz processor on a 100 MHz bus with 512 MB RAM, running VxWorks 5.4.2. High resolution timestamps were taken at the beginning and end of an interval. These experiments were conducted using a post-1.4 (pre-release) version of ACE/TAO/CIAO.

Home creation. In PRISM, the home object is a plain C++ object and its creation time is the time for one dynamic memory allocation and some subsequent initialization of the home object. In CIAO, home creation is a more elaborate process involving creation of a home executor and home servant. The home servant is registered in the POA and an object reference is stored for later use to create components. These are both CORBA objects and creating and activating them is more expensive than creating a plain C++ object. This accounts for at least part of the difference in home creation between CIAO and PRISM. Furthermore, additional overhead in CIAO can occur due such CORBA-related operations as building CORBA policy lists.

Component creation. We infer that the creation of CORBA objects versus plain C++ objects is again the dominant difference between the CIAO and PRISM models in terms of time to create components. This leads to an important observation for the design of component models in DRE systems: flexibility can be traded off for performance through greater coupling of component implementations by selectively replacing CORBA objects with C++ objects wherever the remote or cross-language invocation capabilities CORBA provides are not needed.

Connection establishment. These results further support our earlier inference that the use of CORBA objects instead of C++ objects is the dominant difference between CIAO and PRISM.

6 Related Work

The *Quality Objects* (QuO) framework [26, 27] is an example of configurable middleware, developed by BBN Technologies, which allows DRE system developers to use aspect-oriented [28] techniques to separate the concerns of QoS programming from application logic. In comparison to CIAO, QuO emphasizes dynamic QoS provisioning whereas CIAO emphasizes static QoS provisioning and integration of various mechanisms and behaviors during different stages of the development lifecycle. We are collaborating with BBN to integrate QuO and CIAO [15].

The *dynamicTAO* project [29] applies reflective middleware techniques that extend TAO to reconfigure the ORB at runtime by dynamically linking selected modules, according to the features required by the applications. Their work is similar to QuO in that both provide the mechanisms for realizing *dynamic* QoS provisioning at the middleware level. QuO offers a more comprehensive QoS provisioning abstraction, however, whereas Kon and Campbell's work concentrates on configuring *middleware* capabilities.

The Universal Network Architecture Service (UNAS) [30] is a commercial product that automatically generates software architectures and supports distributed and heterogeneous software systems. One of the main strengths of UNAS is its ability to build, execute and experimentally compare a set of reasonable architectural design alternatives. While UNAS is a proprietary CASE tool that addresses concerns of large-scale software development, the configuration techniques presented in this paper are integrated within an open-standards middleware environment, with applicability to a wider range of middleware implementations and tools.

7 Concluding Remarks

QoS-enabled component middleware is the latest stage of an ongoing evolution of middleware technologies for configuring complex DRE systems. The experimental results presented in Section 4 show that static component configuration mechanisms in the CIAO QoS-enabled component middleware can offer significant improvements in performance and footprint over dynamic mechanisms, while still offering flexibility for configuring component-based DRE systems. While that result is not surprising in light of the mechanisms involved, our detailed experiments also revealed areas where the cost of dynamic mechanisms was small relative to other factors, suggesting it may be useful to reintroduce some dynamic configuration features that were removed in the static approach.

The tension between flexibility of configuration in CIAO and mechanism-level performance in PRISM suggests a hybrid approach where we can be selective about using a component-oriented approach. Domain-specific components can be implemented as regular C++ objects rather than components, if there is not much to be gained in terms of reusability, possibly resulting in improved performance.

The results presented in Section 4 also revealed one area, component registration, in which performance of the static approach was comparable to (RTCIAO) or even lagged behind (CIAO) that of the dynamic approach. These results serve to emphasize the importance of conducting detailed segment measurements rather than relying on aggregate latency to assess performance of individual mechanisms, and may help to identify areas where the static approach could be optimized further.

The empirical case study of Section 5 showed that using C++ objects instead of CORBA objects where possible further improves performance of static configuration mechanisms, though this is achieved at a cost of some increase in coupling of component implementations to their server environments and supporting middleware. Based on these results, we now offer the following observations and recommendations for developers of complex DRE systems:

Observation and recommendation 1. Our results show that dynamic configuration features such as dynamic loading of shared object libraries and spawning new processes may be too costly for some DRE systems resulting in high initialization/reboot times, and shared objects are not available on all platforms. DRE system developers should use the static configuration approach for applications with more stringent system initialization time constraints, or that must operate on a wider range of platforms.

Observation and recommendation 2. Although parsing XML files at runtime incurs a measurable performance cost, that cost is much smaller than the costs of loading dynamically linked libraries or spawning new processes, at least within the context of the example studied in our experiments. Depending on the size of the XML files, the parsing overhead could grow substantially, thereby affecting the run-time performance of the system.

For systems with extreme constraints on initialization times, we recommend that XML descriptor files be parsed into C++ arrays of structures and then compiled into driver programs. However, our results also indicate that a median design point between the static and dynamic approaches may be useful where online parsing of small XML descriptors is performed within an otherwise static configuration approach, *e.g.*, to support limited forms of on-line upgrade and reconfiguration using configuration information that is

not known prior to compile time.

Observation and recommendation 3. Our comparison of PRISM and CIAO showed that flexibility can be traded for performance by using C++ objects instead of CORBA objects. CIAO is more flexible than PRISM and can be used in a variety of domains including the avionics domain for which PRISM was designed.

In systems with stringent performance constraints, *i.e.*, systems with many components or with tight system initialization time constraints, some CORBA objects should be replaced with C++ objects to optimize system initialization times. However, the analysis of which CORBA objects to replace should be driven both by (1) empirical evaluation of which objects contribute the most to delays in system initialization, and (2) which objects must support direct method invocation from objects on remote endsystems.

Acknowledgments

We are grateful to Douglas C. Schmidt for shepherding the final draft of this paper, and to Dennis Noll and James McDonnell for assistance with the experimentation environment at The Boeing Company in St. Louis.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Dec. 2002.

[2] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[3] J. P. Morgenthal, "Microsoft COM+ Will Challenge Application Server Market." www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[4] N. Wang, D. C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

[5] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

[6] Sun Microsystems, "Java^{TM} 2 Platform Enterprise Edition." java.sun.com/j2ee/index.html, 2001.

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

[8] Object Management Group, *Dynamic Scheduling Real-Time CORBA 2.0 Joint Final Submission*, OMG Document orbos/2001-06-09 ed., June 2001.

[9] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[10] Institute for Software Integrated Systems, "Component-Integrated ACE ORB (CIAO)." www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[11] Object Management Group, *Real-Time CORBA Specification*, 1.1 ed., Aug. 2002.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[13] D. C. Schmidt, "Copyright and Licensing Information for ACE^{TM} and TAO^{TM}." www.cs.wustl.edu/∼schmidt/ACE-copying.html.

[14] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, (Agia Napa, Cyprus), Oct. 2004.

[15] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, pp. 45–54, mar 2003.

[16] N. Wang, *Composing Systemic Aspects into Component-Oriented DOC Middleware*. PhD thesis, Washington University, St. Louis, MO 63130, May 2004. Available as Technical Report WUCSE-2004-23 at http://www.cse.seas.wustl.edu/research-techreports.asp.

[17] N. Wang and C. Gill, "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, Jan. 2003.

[18] M. Franz, "Dynamic Linking of Software Components," *IEEE Computer*, pp. 74–81, Mar. 1997.

[19] W. Roll, "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, May 2003.

[20] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, "QoS-enabled Middleware," in *Middleware for Communications* (Q. Mahmoud, ed.), New York: Wiley and Sons, 2003.

[21] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine, "Evaluating Policies and Mechanisms for Supporting Embedded, Real-Time Applications with CORBA 3.0," in *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium*, (Washington DC), IEEE, May 2000.

[22] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[23] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "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*, 2004.

[24] Douglas Niehaus, *et al.*, "Kansas University Real-Time (KURT) Linux." `www.ittc.ukans.edu/kurt/`, 2004.

[25] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[26] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging Quality of Service Control Behaviors for Reuse," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Crystal City, VA), pp. 375–385, IEEE/IFIP, April/May 2002.

[27] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[29] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.

[30] W. Royce, B. Boehm, and C. Druffel, "Employing UNAS technology for software architecture education at the University of Southern California," in *Proceedings of the eleventh annual Washington Ada symposium & summer ACM SIGAda meeting on Ada*, pp. 113–121, ACM Press, 1994.