The Design and Performance of Component Middleware for QoS-enabled Deployment and Configuration of DRE Systems [1]

Venkita Subramonian, [a] Gan Deng, [b] Christopher Gill, [*,a]
Jaiganesh Balasubramanian, [b] Liang-Jui Shen, [a] William Otte, [b]
Douglas C. Schmidt, [b] Aniruddha Gokhale, [b] Nanbor Wang [c]

[a] *CSE Department, Washington University, St. Louis, MO, USA* [2]

[b] *EECS Department, Vanderbilt University, Nashville, TN, USA* [3]

[c] *Tech-X Corp, Boulder, CO, USA*

Abstract

QoS-enabled component middleware can help reduce the complexity of deploying and configuring real-time aspects, such as priorities and rates of invocation. Few empirical studies have been conducted, however, to guide developers of distributed real-time and embedded (DRE) systems in choosing among alternative designs and performance optimizations. This paper makes two key contributions to research on QoS-enabled component middleware for DRE systems. First, it describes optimizations applied to an implementation of the OMG's Deployment and Configuration (D&C) of Components specification that enable static and dynamic configuration of important quality of service (QoS) aspects of DRE systems. Second, it compares the performance of several dynamic and static configuration mechanisms to help guide the selection of suitable configuration mechanisms based on specific requirements of each DRE system. Our results show that ...

Key words: QoS-enabled component middleware, DRE system configuration.

---

* Correspondence: Campus Box 1045, One Brookings Drive, St. Louis, MO, 63130, e-mail: cdgill@cse.wustl.edu, phone: (314) 935-7538, fax: (314) 935-7302.

[1] CIAO is available as open-source software and can be obtained from `deuce.doc.wustl.edu/Download.html`.

[2] Supported in part by DARPA contracts F33615-{01-C-3048, 03-C-4111} (PCES).

[3] Supported in part by DARPA, NSF, Lockheed Martin, Raytheon, and Siemens.

1 Introduction

Some of the most challenging R&D problems are those associated with producing middleware for *distributed, real-time, and embedded* (DRE) systems, where computer processors control physical, chemical, or biological processes or devices. Examples of such systems include airplanes and air traffic control systems, automobiles, power grids, oil refineries, and patient monitoring systems. Despite advances in standards-based commercial-off-the-shelf (COTS) middleware technologies, key challenges must be addressed before COTS middleware can be used to build mission-critical DRE systems effectively and productively. For example, as DRE systems have increased in scale and complexity over the past decade, a tension has arisen between stringent performance requirements and the ease with which those systems can be developed, deployed, and configured to meet those requirements.

DRE systems require design- and run-time configuration steps to customize the behavior of reusable components to meet QoS requirements in the context where they execute. Finding the right configurations of components that meet application QoS requirements is hard. For example, tuning the concurrency configuration of a multi-hypothesis tracker to support both real-time and fault-tolerant QoS involves tradeoffs that challenge even the most experienced engineers. Moreover, since application functionality is distributed over many components in a large-scale DRE system, developers must interconnect and integrate their components in a manner that is correct and efficient, which is very tedious and error-prone using conventional hand-crafted configuration processes.

In addition to being configured properly, the components assembled to form an application must be deployed on the appropriate nodes in an enterprise DRE system. This deployment process is hard since characteristics of hosts onto which components are deployed and the networks over which they communicate can vary statically (*e.g.*, due to different hardware/software platforms used in a product-line architecture) and dynamically (*e.g.*, due to damage, changes in mission modes of the system, or due to differences in the real vs. expected behavior of applications during actual operation). Evaluating the operational characteris,tics of these system deployments can therefore be tedious and error-prone, particularly when deployments are performed manually.

This section summarizes an 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 DRE system development. We also describe several remaining limitations of the state-of-the-art and explain how our work presented in this paper addresses those limitations.

Conventional distributed object computing (DOC) middleware such as the OMG's CORBA [1] and Sun's Java RMI [2] 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.

Conventional component middleware technologies, such as the CORBA Component Model (CCM) [3], J2EE [2] and COM+ [4], extend 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, and (3) dividing system development and configuration concerns into separate aspects, such as implementing application functionality, defining component metadata, and configuring resource management policies. These technologies alone do not adequately address the QoS limitations of DOC middleware, however, since they were designed largely to support enterprise applications, rather than the more stringent QoS needs of DRE systems.

QoS-enabled DOC middleware technologies, such as Real-Time CORBA (RTCORBA) [5] and the Real-Time Specification for Java [6], address key 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. They do not provide deployment and configuration support for component middleware, however, which again causes unnecessary tangling of code for managing QoS aspects with application logic.

QoS-enabled component middleware technologies address the limitations with earlier middleware techniques for use in DRE systems, by combining the capabilities of conventional component middleware and real-time DOC middleware. One such technology is the *Component Integrated ACE ORB* (CIAO) [7], which combines Lightweight CCM [8] mechanisms (such as standards for specifying, implementing, packaging, assembling, and deploying components) and Real-time CORBA mechanisms (such as thread pools and priority pres,ervation policies) to simplify and automate the (re)deployment and (re)configuration of application components in DRE systems. CIAO is built atop The ACE ORB (TAO) [9], which is a widely used ORB that implements the Real-Time CORBA [5] specification.

Our previous work on CIAO [10] focused on supporting declarative configuration of real-time aspects, conducting empirical studies 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. This prior work was concerned mainly with configuration and performance of the real-

time aspects, whereas this paper considers the performance of the deployment and configuration mechanisms themselves.

Our research on deployment and configuration of 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 and available features (*e.g.*, dynamic linking/loading), which the current generation of QoS-enabled component middleware does not address. For example, reinitialization 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 deployment and configuration mechanisms in terms of their flexibility *and* performance. Likewise, few empirical case studies have been conducted to compare the deployment and configuration of standards-based vs. domain-specific QoS-enabled component models.

To overcome limitations with prior work, this paper describes a framework for managing the deployment and configuration of QoS-aware components and middleware services. First, we describe the design and implementation of a new deployment and configuration framework we have integrated into CIAO and compare it to alternative dynamic mechanisms. In addition to issues of static vs. dynamic linking/loading [11], this paper next considers a wider range of issues relevant to component middleware, *e.g.*, configuration parsing, parameter modification, and component assembly. We present quantitative performance comparisons of dynamic and static component deployment and configuration mechanisms, using an illustrative example application built using CIAO. The resulting performance profiles and analysis of help guide DRE system developers in choosing which component deployment and configuration mechanisms to use for particular DRE systems. Finally, we present an empirical case study that compares CIAO with PRISM [12], which is an avionics domain-specific component model developed by Boeing. This case study aims at helping guide developers of DRE systems in making trade-offs in performance and flexibility when applying standards-based QoS-enabled component models vs. using domain-customized solutions.

The remainder of this paper is organized as follows: Section 2 describes a representative example application and describes the design and implementation of a framework that enables dynamic and static deploymend and configuration of CIAO components; Section 3 presents the results of empirical studies conducted to quantify the relative performance of alternative dynamic and static mechanisms; Section 4 evaluates the performance and flexibility of static mechanisms in CIAO and PRISM; Section 5 compares our work with related research on DRE system deployment and configuration tools and QoS-enabled component models; and Section 6 presents concluding remarks.

2 Deploying and Configuring Components in DRE Systems

Compared with conventional enterprise applications, DRE systems have more stringent QoS requirements (*e.g.*end-to-end latency of component method invocations, availability of CPU cycles to meet computation deadlines, and rates of invocation of component methods) that must be satisfied simultaneously at run-time. To ensure that DRE systems can meet their QoS requirements, various deployment and configuration (D&C) activities must be performed to allocate and manage system computing and communication resources end-to-end. To provision end-to-end QoS robustly throughout a DRE system and to improve component reusability, component D&C activities should be decoupled as much as possible from component implementations. For example, D&C directives 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.

To address the dual challenges of system performance and D&C flexibility, CIAO extends the component container definition and metadata representation and manipulation capabilities found in conventional component middleware. In particular, 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, and are exploited in the experiments described in Sections 3 and 4. RTCORBA priority model and threading policies manage resources in RTCORBA 1.0 ORBs [13]. 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.

Example application. To show how CIAO's configuration capabilities can be applied to real-world DRE systems, we now describe a representative example drawn from the avionics domain [12]. Figure 1 illustrates a basic single-

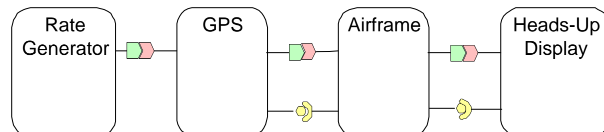

Fig. 1. **Example (Basic SP) Scenario**

processor (Basic SP) scenario 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 [12] 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, many DRE systems share the types 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 applies.

CIAO D&C capabilities. The CIAO *Deployment And Configuration Engine* (DAnCE) is a middleware framework we developed based on the OMG's Deployment and Configuration (D&C) specification [14], which is part of the Lightweight CCM specification [8]. This specification standardizes many aspects of configuration and deployment for component-based distributed systems, including component configuration, component assembly, component packaging, package configuration, package deployment, and target domain resource management. These aspects are handled via a *data model* and a *runtime model*. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describes component assemblies and their configuration and deployment characteristics. The runtime model defines a set of managers that process the metadata described in the data model during system deployment.

We now 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 D&C approach that is better suited to the stringent memory and performance constraints of DRE systems. Irrespective of whether configuration is dynamic or static, however, CIAO's DAnCE allows different functional and real-time policies and mechanisms to be configured in each of the following canonical steps of its overall D&C 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.

Section 3 uses the relative latency of each of these steps to compare the performance of CIAO's dynamic and static D&C mechanisms in DAnCE. Section 4 then uses these steps to compare CIAO's configuration mechanisms to Boeing's PRISM component model.

Dynamic configuration using DAnCE.

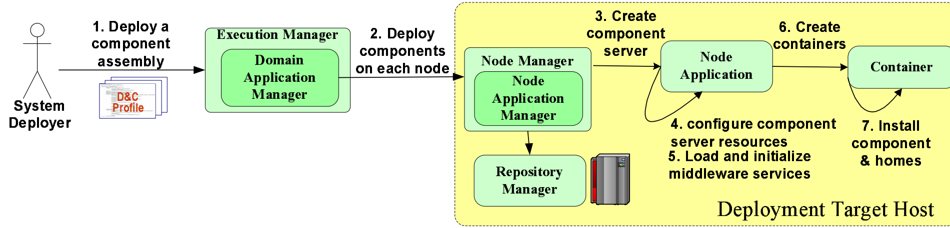

Fig. 2. **Overview of CIAO's DAnCE Framework**

As shown in Figure 2, a DRE system deployer creates XML descriptors for application deployment and configuration metadata using *model-driven development* (MDD) tools [15]. This metadata describes (1) the DRE system component instances to deploy, (2) how these components should be initialized, (3) what QoS policies these components must contain, (4) what middleware services the components use, and (5) how the components are connected to form component assemblies. The metadata is compliant with the data model defined by the OMG D&C specification.

To support additional D&C concerns not addressed by the OMG specification, we enhanced the specification-defined data model by describing additional deployment concerns (such as real-time QoS requirements and middleware service configuration and deployment) discussed in Section 2. By default, DAnEE runs an `ExecutionManager` as a daemon to manage the deployment process for one or more domains. A *domain* is a target environment composed of *nodes*, *interconnects*, *bridges*, and *resources*. An `ExecutionManager` manages a set of `DomainApplicationManagers`, which in turn manage the deployment of components within a single domain. A `DomainApplicationManager` splits a deployment plan into multiple subplans, one for each node in a domain. A `NodeManager` runs as a daemon on each node and manages the deployment of all components that reside on that node, irrespective of which application they are associated with. The `NodeManager` creates the `NodeApplicationManager`, which in turn creates the `NodeApplication` component servers, which host application-specific containers and components.

Static D&C using DAnCE. Although the dynamic approach DAnCE offers by default provides a highly flexible environment for system deployment and configuration, it also suffers from the following drawbacks for DRE systems with stringent performance constraints: (1) XML parsing may be too expensive to be performed during system (re)initialization, (2) multiple pro-

cess address spaces may be required to coordinate the creation and assembly of components, and (3) on-line loading of component implementations may not be possible on real-time OS (RTOS) platforms, such as VxWorks, where dynamically linking facilities are not available.

To address these limitations of dynamic component assembly, we have extended DAnCE to support an alternative static approach where some of the configuration lifecycle tasks is performed off-line, *e.g.*, parsing the XML files and finding the function entry points for creating homes and components. Moreover, all of the runtime and deployment time configuration mechanisms use statically linked C++ objects rather than spawn new processes or load implementation from shared objects or DLLs. These enhancements to DAnCE serve two purposes: (1) for testing and verification purposes the set of components in an application can be identified and analyzed before run-time 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 (*e.g.*, dynamically linked libraries) are available or usable for systems configuration. By refactoring the configuration mechanisms to use only statically linked components, we ensure that our approach can be realized on highly constrained RTOS platforms, such as VxWorks.

CIAO's DAnCE static D&C approach is illustrated in Figure 3. As shown in

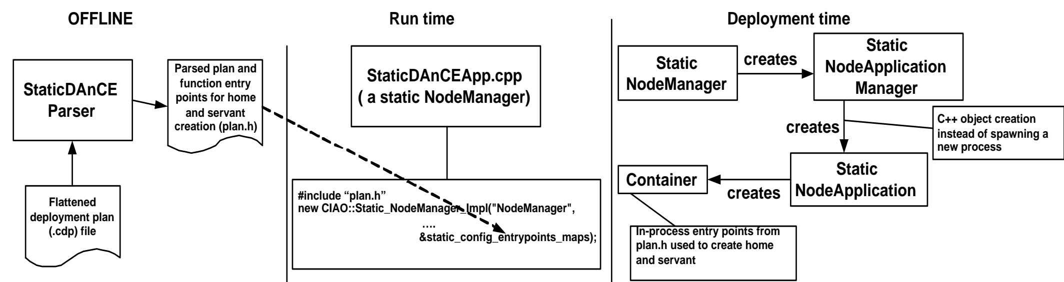

Fig. 3. **Static Component Assembly in DAnCE**

this figure, an offline parser is used to parse the XML deployment plan and generate a C++ header file containing the entry points to functions for creating homes and components. These entry points reference implementations that are statically linked in with a daemon process hosting a `StaticNodeManager` object that takes in the generated entry points as a parameter. The `Static-NodeManager` has the same interface as a `NodeManager` and hence supports all the operations on a `NodeManager`. At deployment time, the `StaticNode-Manager` creates C++ object versions of `NodeApplicationManager` (`Static-NodeApplicationManager`) rather than spawn new processes. Similarly, `Static-NodeApplicationManager` creates statically linked `StaticNodeApplication` C++ objects rather than spawn new processes. Finally, the statically linked component implementations are also instantiated as C++ objects rather than loading the implementations from DLLs. As a result of the static approach, each endsystem can now be booted and initialized within a single address

space, so there is no need for inter-process communication to create and assemble components.

3 Empirical Comparison of Dynamic and Static Deployment and Configuration

To evaluate CIAO's DAnCE dynamic and static configuration mechanisms described in Section 2, we used the Basic SP scenario described in Section 2 as the basis for experiments we conducted to quantify the performance of static and dynamic D&C mechanisms. To ..., these experiments were conducted both with and without real-time extensions, which we term RTCIAO and CIAO, respectively. Our experiments were performed using 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, and leveraging the Pentium time stamp counter to obtain nanosecond resolution in our timing measurements.

Time for assembly. We examined the time taken to assemble all the components in the Basic SP application, including the time to create the server, homes, containers and components and to establish necessary registrations of, and connections between, the components. Application assembly with the static D&C approach takes almost two orders of magnitude less time than with the dynamic approach. This improvement occurs since the dynamic configuration approach parses XML files at run-time and also loads component implementation libraries dynamically, both of which are performed off-line in the static approach. The assembly times involving D&C 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 D&C process to determine which segments contribute the most to the longer assembly times seen with the dynamic approach.

Component server creation. The results of this comparison, shown in Figure 4, reveal 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 D&C mechanisms in Section 2. 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 is seen in the performance of the dynamic approach.

Home creation. Figure 5 shows that the creation time for component homes is significantly higher in the dynamic approach, which occurs since loading libraries dynamically is relatively expensive. More interesting is the reasonably regular bi-modal distribution of latencies seen in these results, for all

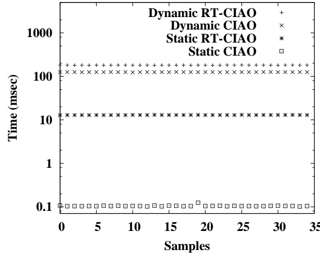

Fig. 4. **Server**

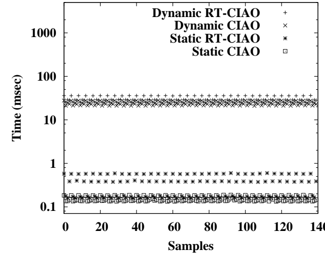

Fig. 5. **Home**

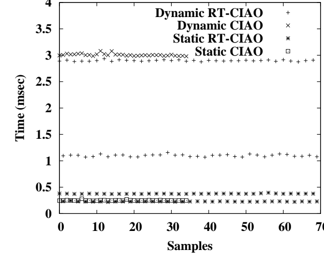

Fig. 6. **Container**

four (static/dynamic, with/without real-time) test programs. We attribute this effect to the differences between configuration parameters for different component homes.

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. The times taken to create containers are shown in Figure 6. The times for RTCIAO are again bi-modal, as may be expected since two different containers are created in our test program, each with a different policy configuration. Also, the number of samples collected for RTCIAO is twice that for CIAO, since in CIAO only one container is created without RT policies.

Component creation. The dynamic approach takes slightly more time than the static approach in this case. 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. We attribute the bi-modal distribution seen in Figure 7 to differences between the number of facets, receptacles, and other interface ports each component must support.

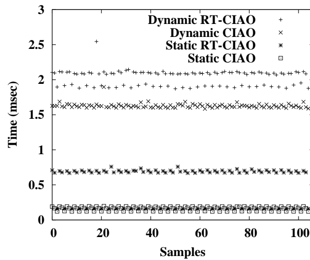

Fig. 7. **Components**

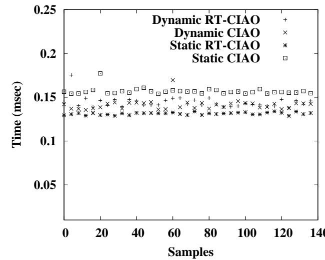

Fig. 8. **Registration**

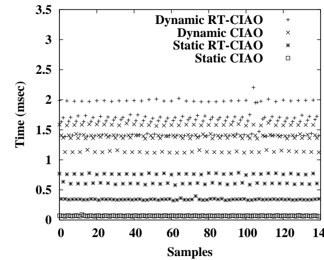

Fig. 9. **Connections**

Component registration. Figure 8 shows the time taken to register components, in which a reference to a component is published to, *e.g.*, a naming service or a disk file. In our experiments, the reference to the Rate Generator component was stored in a disk file. 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. These micro-benchmarks revealed that the creation of the stringified interop-

erable object reference (IOR) contributed the most to the difference (~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 RT-POA, to create the IOR.

Connection establishment. The results of this comparison are shown in Figure 9. We attribute the bi-modal distribution seen there to differences between the number and kinds of interface ports involved in each connection.

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 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.

4 Case Study: Static Deployment and Configuration in CIAO and PRISM

This section compares the design, implementation, and performance of CIAO's DAnCE static D&C mechanisms described in Section 2 against similar mechanisms in Boeing's PRISM [12], which is an avionics domain-specific component model developed by Boeing. DAnCE and PRISM both share the same TAO infrastructure. We first compare and contrast DAnCE and PRISM D&C steps and then present an empirical performance comparison of DAnCE and PRISM using the Basic SP scenario described in Section 2.

In the experiments described in this section, we compare similar individual stages of the two models. In addition to the D&C steps CIAO's DAnCE has in common with PRISM, DAnCE also creates a server object and container objects. The PRISM component model also includes a number of other configuration activities beyond those examined here, including but not limited to initialization of services like persistence, distribution and concurrency. We focus only on the D&C 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.

Assembly steps in CIAO's DAnCE. DAnCE performs the following steps when assembling CIAO components. First, home executor and servant objects are created, the home servant object is registered with the POA, and an object reference is created for the home. The second step creates components using

the home object reference created in the previous step. A component's object reference is then advertised, *e.g.*, in a file or through a naming service. This last step is optional and is done only if it is specified in the assembly descriptor in CIAO (since PRISM does not perform this action we omit this step from further consideration). Finally, connections are established between matching publisher and consumer ports, according to the connection specifications in the descriptor files. The connections between publisher and consumer ports were achieved through a two-way call mechanism.

Assembly steps in Boeing's PRISM. The following steps are performed in the assembly of PRISM components. A home object is first created for each component. The home then creates a factory for that component. Each component's factory then creates the component implementation including facets, receptacles and equivalent interfaces so that connections can be made from/to other components. Finally the connections between facets and receptacles, and between event sources and sinks, are established. 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 the CIAO version used in our experiments did not use the RTEC to connect a publisher and consumer. For our comparisons, therefore, 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 CCM components as in CIAO.

Evaluation of CIAO's DAnCE and Boeing's PRISM. The experiments comparing DAnCE 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, using a post-0.4 (pre-release) version of CIAO and DAnCE and the Basic SP application (shown in Figure 1 and described in Section 2) as part of Boeing PCES Open Experimentation Platform (release 3.0). 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 3 are avoided in PRISM. This improvement comes at a cost in flexibility, however, in that component implementations are more tightly coupled to details of their server environment.

Home creation. Figure 10 shows the time taken by CIAO and PRISM to create a home object. In PRISM, the home object is a plain C++ object and its creation time includes one dynamic memory allocation and initialization of the home object. In CIAO, home creation involves 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 plain C++ objects. Moreover, additional overhead in CIAO can occur due to

standards-compliant operations, such as building CORBA policy lists.

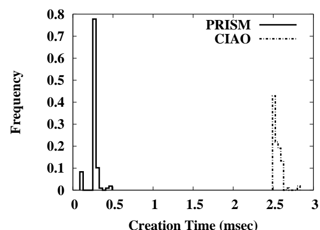

Fig. 10. **Home**

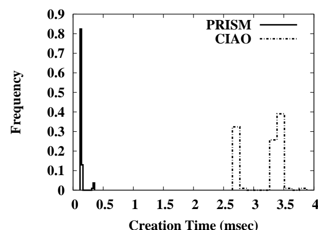

Fig. 11. **Components**

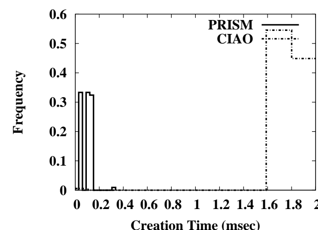

Fig. 12. **Connections**

Component creation. Figure 11 compares component creation times in CIAO and PRISM. In CIAO, the distribution of component creation times is bi-modal, with the different peaks in the graph corresponding to different components. The left peak corresponds to the Heads-Up Display component and the right peak corresponds to the GPS and Airframe components. We attribute this variation between components to differences in the component initialization code for the Heads-Up Display component versus the other components. As is shown in Figure 1 in Section 2, the Heads-Up Display component has only one receptacle and one consumer port. The GPS component is triggered by another component that sends periodic timer events. Hence the GPS and Airframe components each have a facet, a receptacle and a publisher port. The PRISM model does not show as pronounced a variation because the objects are plain C++ objects, as opposed to CORBA objects in CIAO. 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. Figure 12 shows a comparison of the connection establishment times in CIAO and PRISM. 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 configuration times.

5 Related Work

The OpenCCM (`corbaweb.lifl.fr/OpenCCM/`) Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CORBA Component Model (CCM) applications. It implements the Packaging and Deployment (P&D) model defined in the original CCM specification, which omits key aspects of the component configuration and deployment cycle. We are currently working with the OpenCCM team to enhance their DCI so that it is compliant with the OMG D&C specification.

[16] proposes using an architecture descriptive language (ADL) that allows assembly-level activation of components and describes assembly hierarchically. Although CIAO's DAnCE is similar, it uses the XML descriptors synthesized by MDD tools to characterize the metadata regarding components to deploy. Likewise, DAnCE descriptors can specify QoS requirements and/or server resource configurations, so its deployment mechanisms are better suited to deploy applications with desired real-time QoS properties.

[17] proposes the use of the Globus Toolkit to deploy CCM components on a computational grid. Unlike CIAO's DAnCE, this approach does not provide model-driven development (MDD) tools that enable developers to capture various concerns, such as deployment planning and server configuration, visually. Moreover, DAnCE is targeted at DRE systems with stringent real-time QoS requirements, rather than grid applications, which do not provide real-time support.

Proactive [18] is a distributed programming model for deploying object-oriented grid applications and is similar to CIAO's DAnCE in that it also separately describes the target environment using XML descriptors. But DAnCE goes further to specify component interdependencies and ensure system consistency at deployment time.

The *Quality Objects* (QuO) framework [19] separates QoS programming from application logic. While QuO emphasizes dynamic QoS provisioning, CIAO emphasizes static QoS provisioning and integration of various mechanisms and behaviors during different stages of the development lifecycle. The *dynamicTAO* project [20] applies reflective middleware techniques to realize *dynamic* QoS provisioning in the TAO ORB at runtime by dynamically linking selected modules, according to the features required by the applications.

6 Concluding Remarks

QoS-enabled component middleware is the latest stage of an ongoing evolution of technologies for the deployment and configuration (D&C) complex DRE systems. The experimental results presented in Section 3 show that static component D&C mechanisms can offer significant improvements in performance and footprint over dynamic mechanisms, while still offering flexibility for component-based DRE systems. Our detailed experiments revealed areas where the cost of dynamic mechanisms was small relative to other factors, suggesting it may be useful to reintroduce some dynamic D&C features that were removed in the static approach.

The results presented in Section 3 also revealed one area, component registration, in which performance of the static approach was comparable to (RT-

CIAO) 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.

Based on the results in Section 4, we now offer the following observations and recommendations for developers of complex DRE systems: Our results show that dynamic D&C 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. Moreover, shared objects are not available on all platforms. DRE system developers should use the static D&C approach for applications with more stringent system initialization time constraints, or that must operate on a wider range of platforms. 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. 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. Our comparison of PRISM and CIAO showed that flexibility can be traded for performance by using C++ objects instead of CORBA objects. Some CORBA objects could be replaced with C++ objects to optimize system initialization times and this process should be guided by a thorough empirical evaluation as well as application requirements.

Acknowledgments. We are grateful to Dennis Noll and James McDonnell for their assistance with the open experimentation platform and development environment at The Boeing Company in St. Louis.

References

[1] OMG, *The Common Object Request Broker: Architecture and Specification*, 2002.

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

[3] OMG, *CORBA Components*, formal/2002-06-65 ed., June 2002.

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

[5] OMG, *Real-Time CORBA Specification*, 1.1 ed., Aug. 2002.

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

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

[8] Object Management Group, *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., May 2003.

[9] 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.

[10] N. Wang, *Composing Systemic Aspects into Component-Oriented DOC Middleware.* PhD thesis, Washington University, May 2004. Tech Report WUCSE-2004-23 at http://www.cse.seas.wustl.edu/research-techreports.asp.

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

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

[13] 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 RTAS*, 2000.

[14] OMG, *Deployment and Configuration Adopted Submission*, Document ptc/03-07-08 ed., July 2003.

[15] K. Balasubramanian, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems," *International Journal of Embedded Systems special issue on Design and Verification of Real-Time Embedded Software*, Apr. 2005.

[16] V. Quema, R. Balter, L. Bellissard, D. Feliot, A. Freyssinet, and S. Lacourte, "Asynchronous, Hierarchical and Scalable Deployment of Component-Based Applications," in *Proc. of the 2nd International Working Conference on Component Deployment (CD 2004)*, (Edinburgh, UK), May 2004.

[17] S. Lacour, C. Perez, and T. Priol, "Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit," in *Proc. of the 2nd International Working Conference on Component Deployment (CD 2004)*, (Edinburgh, UK), May 2004.

[18] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, "Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications," in *Proc. of the 11th International Symposium on High Performance Distributed Computing (HPDC'02)*, (Edinburgh, UK), July 2002.

[19] 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.

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