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

Quality of service (QoS)-enabled component middleware can help reduce the complexity of deploying and configuring QoS 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. Moreover, few empirical studies have been conducted to examine the performance and flexibility trade-offs between standards-based and domain-specific DRE middleware solutions.

This paper makes three 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 performance trade-offs between 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 DRE system requirements. Third, it compares the performance of our static standards-based approach to an avionics domain-specific approach. Our results show that these optimizations (1) provide developers improved control over key trade-offs between flexibility and performance at different stages of the DRE system lifecycle, (2) enhance trustworthiness of component-based DRE systems by supporting greater customization of how they are configured to meet specific requirements of each application, and (3) offer greater flexibility at a reasonable performance cost, compared to a domain-specific approach.

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

1 Introduction

Challenging R&D problems are associated with producing software for *distributed, real-time, and embedded* (DRE) systems, where computers control physical, chemical, or biological processes or devices. Examples of such systems include airplanes and air traffic control systems, 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 software 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 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 component configurations 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 operation involves trade-offs 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 components in a manner that is correct and efficient, which is 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 DRE system hosts. This deployment process is also 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 characteristics of these systems can also be tedious and error-prone, therefore, particularly when components are deployed manually.

This section summarizes an ongoing evolution of *distribution middleware* platforms [1], which are system software that enable developers to achieve trustworthy DRE system performance while meeting the increasingly rapid system development, deployment, and upgrade cycles demanded by the economics of

* 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.

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

Conventional distributed object computing (DOC) middleware such as CORBA and Java RMI significantly reduces the complexity of writing client programs by separating application-level code from reusable system-level code, but does not address QoS requirements.

Conventional component middleware technologies, such as the CORBA Component Model (CCM), J2EE, and DCOM, 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 vs. 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 DRE systems that have more stringent QoS needs.

QoS-enabled DOC middleware technologies, such as Real-Time CORBA (RTCORBA) and the Real-Time Specification for Java, address key QoS aspects in DRE systems. These technologies support explicit configuration of systemic QoS aspects, such as the priorities of threads invoking object methods. They do not provide component deployment and configuration support, however, which can lead to unnecessary tangling of application logic with code for managing QoS aspects.

QoS-enabled component middleware technologies address the limitations with earlier middleware techniques for 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) [2], which combines Lightweight CCM [3] mechanisms (*e.g.*, for specifying, implementing, packaging, assembling, and deploying components) and Real-time CORBA mechanisms (*e.g.*, thread pools and priority preservation policies) to simplify and automate the trustworthy (re)deployment and (re)configuration of application components and QoS aspects in DRE systems. CIAO is built atop The ACE ORB (TAO) [4], which is a widely used RTCORBA ORB.

Our previous work on CIAO [5] 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 the deployment, configuration and performance of the real-time aspects, whereas this paper considers the performance of the deployment and configuration (D&C) 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 in avionics mission computing systems that can be rebooted or reconfigured while in service [6]. Moreover, few empirical case studies have been conducted to compare the flexibility *and* performance, or to compare standards-based vs. domain-specific component D&C approaches.

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 that we have integrated into CIAO, and compare its alternative dynamic and static deployment and configuration mechanisms. In addition to issues of static vs. dynamic linking/loading [7], this paper considers a wider range of issues relevant to component middleware, *e.g.*, configuration parsing and component assembly. We compare the performance of dynamic and static component deployment and configuration mechanisms, using an illustrative example application built with CIAO. The resulting performance profiles and analysis help DRE system developers choose which component deployment and configuration mechanisms to use for particular DRE systems. Finally, we present an empirical case study that compares deployment and configuration mechanisms in CIAO, which implements the OMG Lightweight CCM standard, vs. PRISM [8], which is an avionics domain-specific component model developed by Boeing. This case study helps developers of DRE systems understand trade-offs in performance vs. flexibility when applying standards-based vs. domain-customized component deployment and configuration solutions.

The remainder of this paper is organized as follows: Section 2 describes a representative example application and describes a framework that enables dynamic and static deployment 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, such as end-to-end latency of component method

invocations, availability of CPU cycles to meet computation deadlines, and rates of invocation of component methods. In open DRE systems [9], both these stringent QoS requirements and requirements for standards-based, modular, and interoperable design must be satisfied simultaneously. To ensure that these 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 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 meta-data, such as XML configuration information for CPU and communication resource allocations, that specify the interfaces of each component and the logical connections between components.

To address the challenges of system performance and D&C flexibility, CIAO extends the component container definition and meta-data representation and manipulation capabilities found in conventional component middleware. For example, CIAO allows configuration of the RTCORBA priority model policies, RTCORBA threading policies, and invocation rates that are relevant to the example application we describe below, and to the experiments described in Sections 3 and 4.

Example application. To show how CIAO's configuration capabilities can be applied to real-world DRE systems, we now describe a representative example from the avionics domain [8]. Figure 1 illustrates a basic single-processor

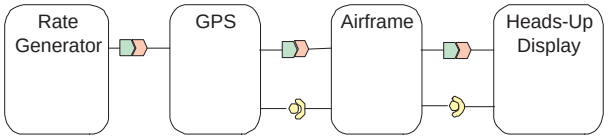

Fig. 1. **Example (Basic SP) Scenario**

(*Basic SP*) scenario involving four software components: (1) a **Rate Generator** component wraps a timer that triggers pushing of events at specific periodic rates to event consumers that register for those events; (2) a **GPS** component wraps one or more hardware devices for navigation; (3) an **Airframe** component wraps the avionics airframe hardware; and (4) a **Heads-up Display** component 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 pulls the new value from the GPS component. The Airframe component next pushes the triggering event to the Heads-up Display component, which pulls the new value from the Airframe component and updates its displays in the cockpit.

This example is representative of real-world avionics applications [10] in which

components executing on embedded micro-controller boards manage physical sensors throughout an aircraft to provide timely situational awareness to the pilot and other personnel. In practice, DRE systems based on QoS-enabled component middleware [8] often contain a large number (*i.e.*, several thousand) of components, with subsets of components linked via specialized networking devices, such as VME buses and Firechannel interconnects. Although applications and their real-time requirements and operating 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.

D&C capabilities for CIAO. The *Deployment And Configuration Engine* (DAnCE) is a middleware framework we developed for CIAO based on the OMG's Deployment and Configuration (D&C) specification [11], which is part of the Lightweight CCM specification [3]. This specification standardizes many deployment and configuration aspects of component-based systems, including component configuration, assembly, and packaging; package configuration and deployment; and 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 meta-data that describes component assemblies and their deployment and configuration attributes. The runtime model defines a set of managers that process the meta-data described in the data model to deploy, execute, and control application components.

We now describe the dynamic assembly of components, where component implementations are loaded from dynamically linked libraries (DLLs). 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 meet the stringent memory and performance constraints of DRE systems. Irrespective of whether configuration is dynamic or static, however, 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, (6) establish *connections* between components, and (7) activate the components so they can process and make requests. Section 3 uses the relative latency of each of these steps to compare the performance of dynamic and static D&C mechanisms in DAnCE. Section 4 then uses these steps to compare DAnCE's static configuration mechanisms to those in Boeing's domain-specific PRISM component model.

Dynamic D&C using DAnCE. As is shown in Figure 2, a DRE system deployer creates XML descriptors for application deployment and configuration meta-data using *model-driven engineering* tools [12]. This meta-data

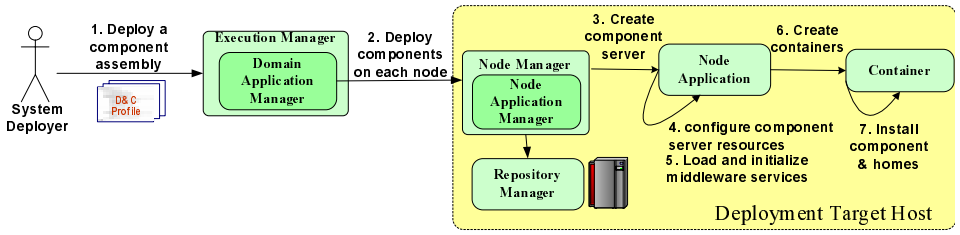

Fig. 2. **Dynamic D&C with the DAnCE Framework**

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. This meta-data is compliant with the data model in 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, DAnCE runs an `ExecutionManager` as a daemon to manage the deployment process for one or more *domains*, which are target environments consisting 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 sub-plans, 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 the particular application with which they are associated. The `NodeManager` creates the `NodeApplicationManager`, which in turn creates the `NodeApplication` component servers that 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 process 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 key configuration tasks are performed off-line, including parsing the XML files and finding the function entry points for creating homes and components. Moreover, all run-time and deployment-time configuration mechanisms use statically linked C++ objects rather than loading implementation libraries dynamically. These enhancements serve two purposes: (1) the components in an application can be

identified and analyzed before runtime, which enhances testing and verification and (2) the latency and jitter of run-time operations following initialization is reduced. Due to the nuances of the platforms traditionally used for deploying DRE systems, not all features of conventional platforms (*e.g.*, DLLs) are available or usable for deployment and configuration. By refactoring the D&C mechanisms to use only statically linked components, we ensure that our approach can be realized on highly constrained RTOS platforms, such as VxWorks.

The static D&C approach in DAnCE is illustrated in Figure 3. As is shown in

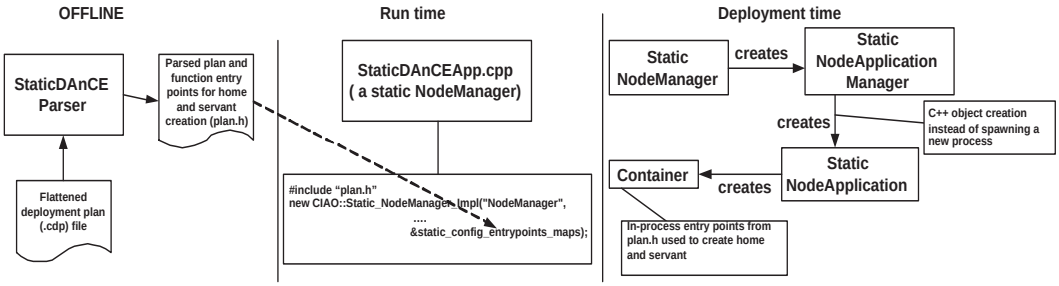

Fig. 3. **Static Component Assembly in DAnCE**

this figure, an offline parser is used (1) to parse the XML deployment plan and (2) 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 with a daemon process hosting a `StaticNodeManager` object that takes the generated entry points as a parameter. The `StaticNodeManager` has the same interface as a `NodeManager` and hence supports all the operations of a `NodeManager`.

At deployment time, the `StaticNodeManager` (3) creates a `StaticNodeApplicationManager` C++ object rather than spawning a new process. Similarly, `StaticNodeApplicationManager` (4) creates statically linked `StaticNodeApplication` C++ objects rather than spawning new processes. Finally, the statically linked component implementations are also (5) instantiated directly rather than by loading them from DLLs. Using this static approach, each endsystem can be booted and initialized within a single address space, so that if all of the components in an assembly are deployed on the same node, there is no need for inter-process communication to create and assemble components.

3 Empirical Comparison of Dynamic and Static D&C

To evaluate the dynamic and static mechanisms in CIAO's DAnCE described in Section 2, we used the Basic SP application (also described in Section 2) as the basis for experiments we conducted to quantify the performance of static and dynamic D&C mechanisms. To measure the cost of configuring real-time QoS aspects as well as the baseline cost of configuring components, these experiments were conducted both with and without real-time extensions, which we term RTCIAO and CIAO, respectively. Our experiments used 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. For each experiment presented in this section and the next, we report all data collected during a series of repeated configuration runs (one sample per run).

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 is largely because the dynamic configuration approach parses XML files at run-time and 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) since 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.

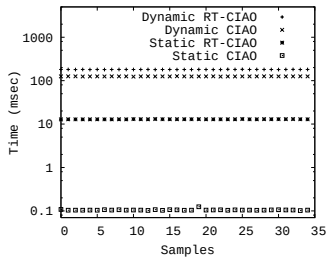

Fig. 4. **Server**

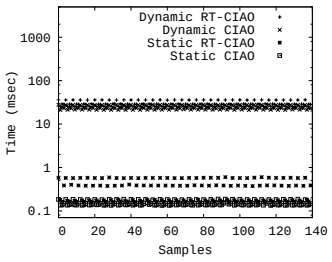

Fig. 5. **Home**

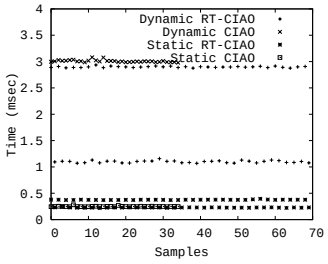

Fig. 6. **Container**

Home creation. Figure 5 shows that the creation time for component homes is much higher in the dynamic approach due to the cost of loading DLLs. More interesting is the bi-modal distribution of latencies seen in these results, for all four test programs (static/dynamic, with/without real-time features). 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 con-

tainer 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 since two different containers are created in our test program, each with a different policy configuration. The number of samples collected for RTCIAO is also twice that for CIAO, since CIAO only creates one container without RT policies.

Component creation. The dynamic approach takes slightly more time than the static approach to create components. In the dynamic approach, the component implementations are packaged in DLLs 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.

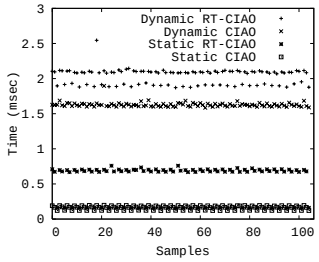

Fig. 7. **Components**

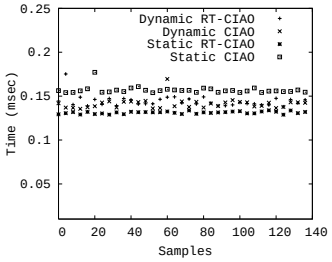

Fig. 8. **Registration**

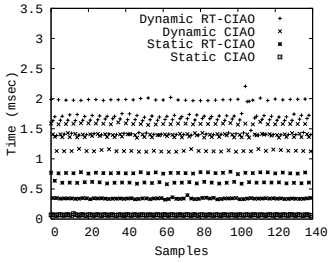

Fig. 9. **Connections**

Component registration. Figure 8 shows the time taken to register components, in which a reference to a component is published, *e.g.*, to 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 of each of these segments. 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 different portable object adapters (CORBA mechanisms for hosting objects and dispatching requests to them) used by static CIAO vs. static RTCIAO 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 compared to the simple iterator constructs used by the static approach to traverse information stored in C++ arrays.

4 Case Study: Static D&C in CIAO and PRISM

This section compares the design, implementation, and performance of DAnCE's static D&C mechanisms described in Section 2 with similar mechanisms in Boeing's PRISM [8], 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 the static D&C steps and then present an empirical performance comparison of the static D&C mechanisms in DAnCE and PRISM using the Basic SP scenario described in Section 2. To our knowledge, this is the first empirical comparison of an implementation of the OMG's D&C specification with a domain-specific approach.

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 first 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 via 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 next 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, we therefore do not take into account the connections established by means of the RTEC. We also note that most of the PRISM objects created in these steps are plain C++ objects, rather than CCM components used in CIAO.

Evaluation of DAnCE and PRISM static configuration. These experiments 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 components 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 so its creation time consists of one dynamic memory allocation and initialization of the home object. In CIAO, home creation involves creation of a home executor and a 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.

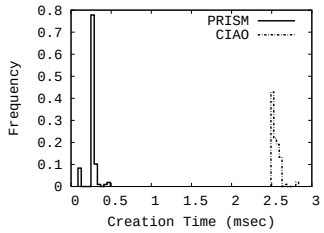

Fig. 10. **Home**

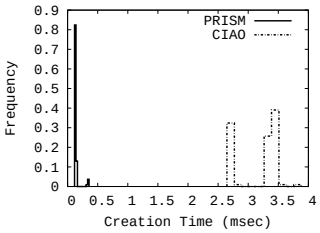

Fig. 11. **Components**

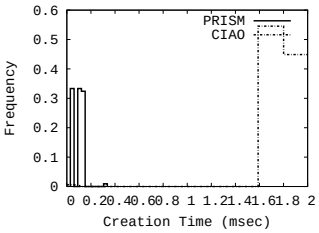

Fig. 12. **Connections**

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 left peak corresponding to the Heads-Up Display component and the right peak corresponding 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 result leads to an important observation for the design of component D&C mechanisms for 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. Our future work focuses on automating these optimizations.

Connection establishment. Figure 12 shows a comparison of the connection establishment times in CIAO and PRISM. These results further support our earlier observation 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 distributed services to form a deployment domain for CORBA Component Model (CCM) applications. We are working with the OpenCCM team to enhance their DCI so that it is compliant with the OMG D&C specification and interoperable with DAnCE.

[13] proposes using an architecture description language that allows assembly-level activation of components and describes assembly hierarchically. [14] proposes to use the Globus Toolkit to deploy CCM components in a computational grid. DAnCE descriptors can specify QoS requirements and/or server resource configurations, so it is customized to meet the D&C needs of applications with real-time QoS requirements.

[14] proposes to use the Globus Toolkit to deploy CCM components in a computational grid. Unlike DAnCE, this approach does not provide model-driven engineering tools [12] that enable developers to capture 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 systems, which do not provide real-time support.

Proactive [15] is designed for deploying object-oriented grid applications and is similar to DAnCE in that it also separately describes the target environment using XML descriptors. DAnCE goes further to specify component interdependencies and to ensure system consistency at deployment time.

The *Quality Objects* (QuO) framework [16] separates QoS programming from application logic. Where QuO emphasizes dynamic QoS provisioning, DAnCE emphasizes static QoS provisioning and integration of needed D&C mechanisms at different stages of the development lifecycle. The *dynamicTAO*

project [17] 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. Unlike DAnCE, however, *dynamicTAO* uses a conventional DOC middleware paradigm for deployment and configuration, rather than a component middleware paradigm.

Gorton and Liu [18] have studied the performance of an example stock trading application under different EJB-based component architectures. These studies focused on scalability and other concerns appropriate to enterprise computing applications, Whereas our studies have focused on real-time concerns appropriate to the DRE application domains for which CIAO and DAnCE were designed. Weis, et al., have developed a model-driven approach to configuring QoS aspects of distributed systems [19] that is complementary to our approach with CIAO and DAnCE.

6 Concluding Remarks

QoS-enabled component middleware is the latest stage of an ongoing evolution of technologies for the development, deployment, and configuration (D&C) of complex DRE systems. Our experimental results in this paper 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 (RTCIAO) or even lagged behind (CIAO) that of the dynamic approach. These results 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.

This paper has presented several new empirical benchmarks and practical techniques for use in configuring component-based DRE systems. For example, our results show that dynamic D&C features such as DLLs and spawning new processes may be too costly for some DRE systems, resulting in high initialization/reboot times. Moreover, since DLLs are not available on all platforms, our static D&C mechanisms expand the range of platforms on which component based DRE systems can be built. DRE system developers should use the static D&C approach for applications with more stringent system initialization time constraints, or that must operate on highly constrained platforms.

This paper has also quantified trade-offs between standards-based and domain-specific D&C mechanisms, so that system developers can make informed and precise engineering decisions to meet the performance and flexibility needs

of each particular DRE application. For example, our studies of PRISM and CIAO showed that flexibility can be traded for performance by using C++ objects instead of CORBA objects, and quantified those performance trade-offs on realistic DRE system platforms. 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 by application requirements.

Many of the techniques presented in this paper can be applied to other component based environments, such as EJB. For example, the performance cost of XML parsing at runtime can be reduced by pre-parsing XML descriptor files into Java arrays of structures off-line and then compiling them into efficient on-line driver programs, much as we did in C++.

References

[1] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2002.

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

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

[4] 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.

[5] 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.

[6] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

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

[8] 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.

[9] Open Systems Joint Task Force, "What is an Open System?." http://www.acq.osd.mil/osjtf/whatisos.html.

[10] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," in *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04), Embedded Applications Track*, (Toronto, CA), IEEE, May 2004.

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

[12] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, 2006.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] I. Gorton and A. Liu, "Performance Evaluation of EJB-Based Component Architectures," *IEEE Internet Computing*, vol. 7, pp. 18–23, May 2003.

[19] T. Weis, A. Ulbrich, K. Geihs, and C. Becker, "Quality of Service in Middleware and Applications: A Model-Driven Approach," in *8th IEEE International Enterprise Distributed Object Computing Conference (EDOC '04)*, pp. 160–171, Sept. 2004.