Towards Heterogeneous Composition of Distributed Real-time and Embedded (DRE) Systems using the CORBA Component Model

James H. Hill
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

***Abstract*—This paper presents a method for using the CORBA Component Model (CCM) to integrate heterogeneous DRE systems. It has been realized in a extensible C++ template framework named CCM++. Experience from applying CCM++ on an example DRE shows can simplify heterogeneous DRE system composition while remaining spec-compliant. This therefore increases the portability of the solution to existing tools and frameworks that are CCM-compliant.**

***Keywords*-CORBA Component Model (CCM), CCM++, heterogeneous composition, distributed real-time and embedded (DRE) systems**

I. Introduction

Component-based software engineering [1] envisions rapid development of large-scale software systems by assembling and deploying units of application functionality that is modularized in reusable components. In addition, CBSE promotes separation-of-concerns such that a given concern of the software system can be addressed independent of other concerns. For example, the CBSE for enterprise distributed real-time and embedded (DRE) software systems separate deployment, configuration, and lifecycle management concerns from each other. This therefore provides a highly flexible and configurable solution that can easily support general-purpose to domain-specific tools [2]–[4].

The CORBA Component Model (CCM) [5]–[7], which is an Object Management Group (OMG) (www.omg.org), is an example CBSE paradigm. CCM arose as a need to provide a CBSE paradigm for implementing component-based DRE systems atop the CORBA [8] specification. Similar to other CBSE paradigms, application developers focus on the application-logic of the software systems. This is opposed to low-level implementation concerns that are handled by the underlying component-based middleware. Because of the increased level-of-abstract for application development, application developers—in theory—are able to design and implement component-based DRE systems more rapidly than traditional software systems [1].

Nowadays, there are new distributed middleware technologies for DRE systems, such as event-based middleware [9], that do not support a component model. Instead, DRE system developers who use such technologies must continue using the low-level abstractions until an adequate CBSE paradigm emerges—if this even happens. Moreover, this impedes DRE system developers ability to compose such systems from many different middleware technologies without using an *ad hoc* or proprietary approach that operates at a low-level of abstraction [10], [11].

Unfortunately, waiting for an adequate CBSE paradigm to emerge for new distributed middleware technologies can be timely. This is because it typically requires adaption of a standard specification, which must be then implemented. A more direct solution is to discover methods for integrating the distributed middleware technologies into component-based middleware without violating the current specification. This can allow DRE system developers to reap the benefits of CBSE, without the timely process of waiting for such a paradigm to emerge for the new distributed middleware technology.

This paper therefore discusses a method for integrating different distributed middleware technologies into CCM. The main contributions of this paper are as follows:

1) It discusses how the existing CCM specification can be leveraged to integrate different distributed middleware technologies—similar to how existing pluggable protocol frameworks operate;
2) It presents CCM++, which is a CCM compliant C++ template framework that simplifies integrating distributed middleware technologies into CCM;
3) It discusses different design alternatives for integrating distributed middleware technologies into CCM—including the advantages and disadvantages of each alternative; and
4) It presents lessons learned from the using CCM++ on different middleware technologies and discusses directions forward.

By leveraging the CCM specification to integrate different middleware technologies, DRE system developers have a portable solution that can easily integrate with existing tools that are CCM-compliant.

Paper organization. The remainder of this paper is organized as follows: Section II provides an overview of CCM and portions of the specification that is critical for

understanding CCM++; Section III discusses the design and functionality of CCM++; Section IV illustrates how CCM++ is used to integrate different distributed middleware technologies; Section V compares this work to other related work; and Section VI presents concluding remarks and lessons learned.

II. Overview of the CORBA Component Model

This section provides a brief overview of CCM and portions of the specification is that critical for the discussion later in this paper. In CCM, components have a well-defined structure. They can also send and receive remote-method invocations and events to/from other components. This paper, however, focuses on the event aspect of CCM.

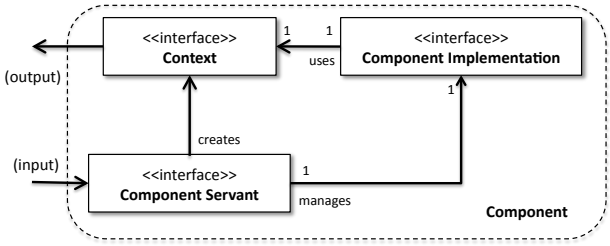

Figure 1. Overview of a OMG's component-based software engineering methodology in the CORBA Component Model.

As shown in Figure 1, a CCM component is composed of the following three distinct entities:

- **Component Servant** – The component servant is responsible for hosting a component implementation (see below). It is also the entity that receives events from other components on its declared events ports. When an event is received by a servant, the component servant forwards the event to the component implementation. The component servant also provides well-defined methods for establishing connections with other components. Such methods include a generic lookup method by port name, and port-specific methods. The component servant also exposes component lifecycle events, such as activate and passivate, that are forwarded to the component implementation. Finally, this entity is auto-generated by CCM compilers from user-defined Interface Definition Language (IDL) files.
- **Context** – The context allows a component to send events to other components. The context is created by the component servant when the component is first instantiated. The component servant then invokes `set_session_context()` method on the component implementation to assign it the new created context. The component implementation can also use the context to get objects available in its execution environment, such as domain-specific services and connected facets. Finally, this entity is auto-generated by CCM compilers from user-defined IDL files—similar to the component servant described above.
- **Component Implementation** – The component implementation is the core "business-logic" of the component. It defines the behavior and workload of the a component. Because of CCM's design, the component implementation is decoupled from the component servant and context. This implies that a spec-compliant component implementation can be used with any spec-compliant component servant—given the component servant can host the corresponding component implementation. Finally, the component implementation is the portion of the component that a distributed system developer must implement.

III. Detailed Design and Functionality of CCM++

This section discusses the detailed design and functionality of CCM++, which is a C++ template framework for integrated distributed middleware technologies into CCM. It also presents different design alternatives for realizing the integration.

A. Discussion of Design Alternatives

As discussed in Section II, a CCM component has three main entities: component implementation, context, and component servant. Because of a CCM component's structure, there are three different design alternatives for integrating distributed middleware technologies into CCM. The following is a description of the three different design alternatives, which are also highlighted in Figure 2:

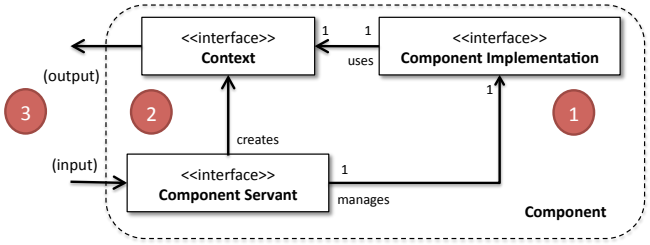

Figure 2. Different design alternatives for integrating middleware technologies into CCM.

1) **Direct integration.** Direct integration (location 1 in Figure 2) is when the different technologies are integrated at the implementation-level of the component. The advantage of direct integration is that it should have the best performance since the implementation interacts directly with the target technology. The disadvantage of this approach is that it tightly couples the component implementation with the distributed middleware technology. Moreover, this design alternative requires distributed system developers to understand each candidate middleware, which may be outside

of their knowledge domain. This method is therefore similar to using the existing low-level abstractions, and does not fully leverage the component paradigm. Finally, distributed system developers have to decide what technologies to use for sending events during early phases of the software lifecycle (*i.e.*, during the design phase) since application-logic must interact with it directly.

2) **Servant-based integration.** Servant-based integration (location 2 in Figure 2) is when the different distributed middleware technologies are integrated at the servant/context-level of the component (*i.e.*, outside of the component implementation). The advantage of this approach is that it shields the implementation (and distributed system developer) from the underlying complexities of different distributed middleware. Likewise, this approach is *semi-coupled*, *i.e.*, not tightly-coupled, but not completely loose-coupled. This is because the servant can host many different implementations without the implementation knowing what distributed middleware technology is used to carry out network communication. The disadvantage of this approach is that the servant and context must implement each candidate technology.
   This design alternative does not require selecting the candidate distributed middleware technology during early phases of the software lifecycle because the selection does not impact a component implementation's. It, however, requires upgrades, or extensions, to CCM compilers so they generate the appropriate component servant and context that use the corresponding distributed middleware technology.
3) **Gateway-based integration.** Gateway-based integration (location 3 in Figure 2) is when the distributed middleware technologies are integrated outside of the component using a mediator that is responsible for transforming the event to the target middleware technology [12]. The advantage of this design alternative is that it provides loose-coupling for integrating distributed middleware. Distributed system developers therefore delay selecting what distributed middleware to use for communication until late in the software lifecycle (*i.e.*, system integration time).
   The disadvantage of this approach is the transformation process occurs later in in the communication process, *i.e.*, after reaching the connector, which can negatively impact performance. Likewise, distributed system developers have only indirect access to the capabilities of the target technologies used in the connector. This, however, can be viewed as both an advantage and disadvantage.

Because of the advantages and disadvantages of the different design alternatives discussed above, this paper focuses on the servant-based integration method (*i.e.*, design alternative 2) for integrating distributed middleware into CCM components. This method was selected because (1) it decouples the component implementation from the underlying distributed middleware. This means different component implementations can be used with different component servants and middleware. Likewise, (2) this method of integration does not require modifications to the original CCM specification. It is therefore simpler to integrate this design alternative with tools that are CCM spec-compliant.

B. Servant-based Integration into CCM

Before discussing the details of achieving servant-based integration using CCM, it is first necessary to mention that this design alternative has been applied to only the event source and sink portions of CCM. This is because, to-date, only event-based distributed middleware, such as OpenSplice [13] and RTI-DDS [14], have been integrated into CCM at the servant-level. Although this may be the case, it is believed that the concepts discussed in the section can be applied to the facet/receptacle portions of CCM to support remote method invocation. This idea, however, is not validated in this paper.

With that being said, Figure 3 provides a detailed look at CCM's event model. As shown in this figure, the event consumer (*i.e.*, `EventConsumerBase`) is the key entity for event communication. Each CCM component that receives events is responsible for instantiating and activating an event consumer object for each of its event ports.

When another component wants to send an event to the receiving component's active event port, the client application must first request a reference to the target event consumer via the generic consumer accessor method, *i.e.*, `get_consumer(name)`. The sending component can also use a event-specific method to request to target event consumer reference. After getting a reference to the event consumer of interest, the client application then invokes the `connect_consumer()` or `subscribe()` method on the sending component for either single point or multiple point delivery, respectively. This process is representative of establishing an event-based connection between two CCM components.

```
1  module Components
2  {
3    typeprefix Components ``omg.org'';
4
5    interface EventConsumerBase
6    {
7      void push_event (in EventBase evt)
8        raises (BadEventType);
9    };
10 };
```

Listing 1. The Interface Definition Language specification for a EventConsumerBase object in CCM.

Once an event-based connection is established between two components, the sending component can send events via the generic `push_event(in EventBase ev)` method

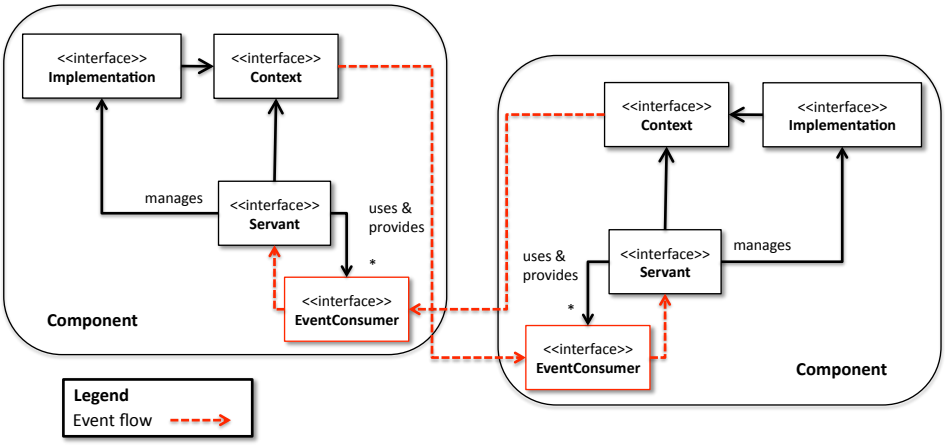

Figure 3. Detailed overview of the event-based communication architecture in CCM. This figure shows how events flow between components via the event consumer entity. The CCM specification assumes that event communication over the network uses CORBA.

implemented by all event consumer objects as shown by Listing 1. Unfortunately, this method assumes network communication uses CORBA events, which are object-by-value types. Instead, the desire is to have events that pass through their corresponding event consumer object be sent via some other distributed middleware technology.

Although it may seem like a trivial process to send events via a different distributed middleware, this is not the case—especially when (1) attempting to do so without violating the CCM specification and (2) pushing complexities associated with the target distributed middleware into the component servant. Moreover, its even harder to establish a valid for the target distributed middleware when such details are not provided via CCM. To overcome this challenge, each middleware is responsible for extending the standard CCM event consumer definition with attributes and methods needed to (1) establish a valid connection and (2) send an event via its corresponding middleware (*i.e.*, not using CORBA).

```
1  module Components
2  {
3    module TCPIP
4    {
5      struct Endpoint
6      {
7        // IP address of the target
8        string address;
9        // Unique Id of the target object
10       string UUID;
11       // The event id
12       unsigned long event;
13     };
14
15     interface EventConsumerBase :
16       ::Components::EventConsumerBase
17     {
18       // Target endpoint for the connection.
19       readonly attribute Endpoint remote_endpoint;
20     };
21   };
22 };
```

Listing 2. The Interface Definition Language specification for a EventConsumerBase object in CCM.

For example, Listing 2 illustrates an extended event consumer definition for sending events over a propriety TCP/IP connection. This is analogous of integrating CCM into distributed middleware for legacy systems. As shown in this example, the TCP/IP event consumer object provides an remote endpoint attribute. The sending component uses this information when either `connect_consumer()` or `subscribe()` is invoked the establish the connection.

At that point, the sending component's servant narrows the event consumer reference to its concrete type and requests the connection information. The servant then establishes the corresponding connection so the component is able to send events via `send_event()` over the TCP/IP middleware instead of using CORBA. Finally, if the concrete event consumer is not of the expected type, then the connection process fails.

Translating events for communication. The goal of the servant-based integration design alternative is to decouple the component implementation from different distributed middleware. The current approach therefore uses the default definition of a the `send_event()` method to transmit events to other components. This implies that the creation and initialization of the events are done in CORBA. Unfortunately, this prohibits sending events "as-is" via another distributed middleware technology.

To overcome this challenge, each concrete event consumer and its proxy is a strongly-typed object that knows its CORBA type and its event type in the target distributed middleware. When the `send_event()` method is invoked, the strongly-typed event consumer reference transforms the object into the target type for its corresponding distributed middleware. Whenever the event is received on the other end of the connection, it is transformed back into a CORBA event of the corresponding type, and passed to the component implementation. Although this approach will experience a performance penalty because of the translation [15], its de-

couples the component implementation from the distributed middleware without violating the CCM specification.

C. The CCM++ Template Framework

The CCM++ Template Framework is designed to ease integrating different distributed middleware technologies written in C++ into CCM. Furthermore, it is a result of gained experience from integrating different distributed middleware technologies into CCM and realizing there is a common pattern behind the integration process. Figure 4 provides an high-level overview of the CCM++ Template Framework.

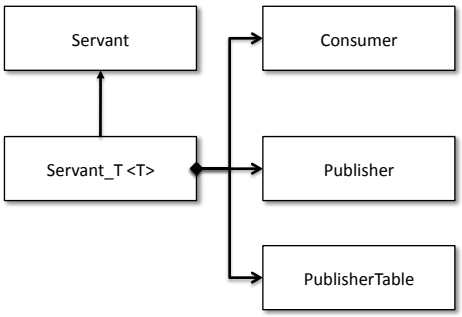

Figure 4. Overview of the CCM++ Template Framework for integrating distributed middleware into CCM at the servant-level.

As shown in this figure, the CCM++ Template Framework provides five key elements. The `Servant` is the base class for all servants. This object implements all the required methods of a CCM component servant that are not based on the component servant's type, or its event types. `Servant_T` is a strongly typed servant object. It is parameterized by the concrete servant type, the component implementation type, and the context type. When this object is instantiated, it will create the context and install it into the hosted component implementation. This object also provides a constructor for binding the component servant to an component implementation, and configuring the POA for its event consumers.

The `Servant_T` object has a collection of `Consumer` objects, which are implementations of concrete event consumers (see Listing 2 for an example) stored by their name. It also contains a collection of event consumer proxy objects, which are stored in either a `Publisher` or `PublisherTable` container. The goal of the `Servant_T` object is to implement all the generic methods that do not require strongly-typed information, such as `get_consumer()`.

When using CCM++ to integrate a different technology, the integrator has several responsibilities. First, the integrator must extend the `Servant_T`, `Consumer`, `Publisher`, and `PublisherTable` objects (as needed) with concrete elements specific to their distributed middleware. For example, the `Consumer`, `Publisher`, and `PublisherTable` objects should be extended such that they contain logic for establishing a connection, translating an event, and sending an event. Likewise, `Servant_T` is extended if more domain-specific configuration is required, such as reading a proprietary configuration document that defines quality-of-service parameters for the event ports.

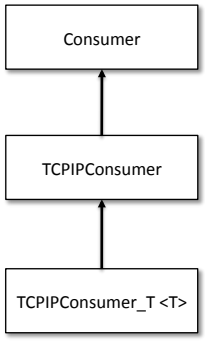

Figure 5. A method for extending the event-related objects in CCM++ to maximize the most code reuse across different event types.

The ideal way to extend the CCM++ event objects is shown in Figure 5. The first derived class contains code that is not bound to a specific type, such as establishing a connection. The most derived class is a template object that is parameterized by the event type. This class therefore implements type specific methods, such as `send_event()`.

Once CCM++ has been extended to support the new distributed middleware, the next step of the integration process is updating the CCM++ compiler to generate operators that transform CORBA events to events (or objects) in the target distributed middleware. At this point, there is no simpler method than manually updating the compiler with a new backend to perform the transformation for the given distributed middleware. Finally, the integrator must update the CCM++ compiler to generate the correct source code for the strongly-typed servant and context. Although this manual effort is required, it is a one-time only process—given the CCM++ extensions do not change.

IV. Experience Using CCM++ To Integrate Distributed Middleware Technologies

This section discusses our experience and effort using CCM++ to integrate different distributed middleware technologies into CCM.

A. Integrating Object-based TCP/IP

The object-based TCP/IP middleware, which is implemented atop of the Adaptive Communication Environment (ACE) [16], is a proprietary solution developed specifically to showcase integrating a distributed middleware technology that is not based on the CORBA specification. Although CORBA uses TCP/IP, this middleware implementation does not have many features found in CORBA, such as reliable

communication, an expensive IOR (in terms of size), and string-based (or name-based) dispatching.

The IDL specification for extending CCM++ to support the object-based TCP/IP middleware was shown in Listing 2. As shown in that listing, the object-based TCP/IP event consumer provides an attribute that details the connection information. This information includes the IP-address and target port of the listening process, the unique id of the component to receive the event (similar to an IOR), and the event id for dispatching to the correct method on the receiving component implementation.

The integration effort resulted in 1,055 source lines of code (SLOC)[1] when extending the CCM++ Template Framework with constructs specific for this distributed middleware. Likewise, it required 3,906 SLOC to extend the CCM++ compiler to generate the appropriate servants and contexts.

B. Integrating OpenSplice DDS and RTI-DDS

OpenSplice DDS and RTI-DDS are implementations of the OMG Data Distribution Services (DDS) [9] specification. The DDS specification uses a publisher-subscriber paradigm to communicate events between senders and receivers. In addition, client's (both publishers and subscribers) can configure different QoS parameters on the publishers and subscribers, such as acceptable publication and subscription rates.

```
1  module Components
2  {
3    module OpenSplice
4    {
5      struct TopicDescription
6      {
7        /// The name of the topic.
8        string name;
9
10       /// The type name of the topic.
11       string type_name;
12     };
13
14     /// Extension interface for connecting subscriber
15     /// using CCM component connections as virtual
16     /// subscriptions.
17     interface EventConsumer :
18       ::Components::EventConsumerBase
19     {
20       /// Get the topic description for this
21       /// event consumer.
22       readonly attribute
23         TopicDescription topic;
24     };
25   };
26 };
```

Listing 3. IDL specification for the OpenSplice DDS event consumer.

Listing 3 illustrates the event consumer for OpenSplice DDS. The event consumer for RTI-DDS is similar, and is therefore not illustrated. As shown in this listing the OpenSplice DDS event consumer has a single attribute that the publisher's topic. The information contained in this topic is the actual topic's name and the string version of the topic type. The contents of this topic structure, *i.e.*, `TopicDescription`, are required for a publisher to create a topic for writing, and a subscriber to create to create a topic for reading.

Finally, DDS requires each type to be bound to an concrete event type, not the to be confused with `type_name` above. This requirement is handled directly by the CCM specification itself. In that, each event source/sink has a concrete event type. This is used to represent the concrete event type in DDS. The type name is determined at compile-time when the CCM++ backend for either DDS backend execute. The topic name, however, is determined at deployment time—either statically or dynamically. For example, one approach is to make define the topic name as a concatenation of the component servants unique id and the event source name, such as `StockDistributor.stock_update`.

Table I
MEASURING THE EFFORT IN SLOC FOR INTEGRATING OPENSPLICE DDS AND RTI-DDS INTO CCM++.

Category	OpenSplice DDS	RTI-DDS
Template Framework	1,385	1,338
Compiler	1,049	1,258

Table I shows the integration effort for extending the CCM++ Template Framework with constructs to support both OpenSplice DDS and RTI-DDS. As shown in this table, both OpenSplice DDS and RTI-DDS required close to the same amount of effort to integrate into CCM at the servant-level. This is both are implementations of the same specification. It is also worth noting that although they both require similar effort, they did not require the same amount of time to implement. This is because experience gained from integrating one made it easier to implement the other. Finally, since OpenSplice and RTI-DDS uses their own naming conventions when not required by the DDS specification, it is *hard* to create a single implementation unless the *Wrapper Facade* [17] design pattern is used to abstract away this differences behind a common interface.

C. On Heterogenous Composition

The previous two examples integrated a single distributed middleware technology into CCM. This means that the DRE system is homogenous. Heterogeneous composition in CCM++ is similar to homogenous composition. This is because the CCM++ framework's primary purpose to the enforce the CCM specification. Heterogeneous composition is therefore the responsibility of the (heterogeneous) distributed middleware technology being integrated via CCM++.

To illustrate this concept, we developed a distributed middleware technology named *CUTS for Heterogenous Architectures, Objects, and Systems (CHAOS)*. CHAOS is an architecture that supports heterogeneous composition of

[1]All SLOC metrics were calculated using SourceMonitor 2.5. It is freely available for download from the following location: www.campwoodsw.com/sourcemonitor.html.

DRE systems using known distributed middleware technologies. For example, CHAOS currently supports heterogeneous composition of DRE system using the aformentioned technologies: OpenSplice, RTI-DDS, TCP/IP, and CORBA (automatically built-in).

The integration effort for CHAOS is similar to the previous two examples. The main challenge in CHAOS, however, is enforcing the connection semantics between two event ports. For example, an output event port that publishes an TCI/IP event should not be able to communicate with an input port that received DDS events. This is because the semantics of either port is different.

To enforce these semantics in CHAOS, CHAOS uses a domain-specific modeling language (DSML). DRE system developers use CHAOS by modeling the structure of the DRE system, and specifying the semantics of each port on the component. When DRE system developers model connections between different ports, the DSML ensures that the port semantics are of the same distributed middleware technology. Finally, DRE system developers use the model-based version of the CCM++ compiler to generate the appropriate code for CHAOS.

V. Related Work

Krishnakumar [12], [18] et al. presented an approach for integrating different distributed middleware technologies into CCM. In particular, they investigated an approach for integrating Microsoft .NET web services into CCM. The solution that Krishnakumar et al. presented used the gateway approach in that CCM events were sent to a mediator (or gateway) before sending the events to the web service, which communicated using the Simple Object Access Protocol (SOAP) [19]. CCM++ differs from the investigation conducted by Krishnakumar et al. in that it focuses on servant-based integration, instead of gateway-based integration. Moreover, in-depth study of the their investigation showed that there were slight modifications to the CCM specification to support the integration. The CCM++ framework presented in this paper does not require any modifications. Instead, it extends existing abstractions so it can remain spec-compliant.

DDS4CCM [20] is another another approach, and specification, for integrating different distributed middleware into CCM. Similar to Krishnakumar's work, the DDS4CCM approach also uses a gateway-based approach. In the DDS4CCM approach, the gateway abstractions are called *connectors*. The CCM++ approach differs from the DDS4CCM approach in that it (1) uses servant-based integration and (2) does not require modifications to the existing CCM specification. For example, the DDS4CCM approach is only possible because of extensions made to the existing CORBA specification, such as adding new keywords to IDL. As previously stated, the CCM++ approach extends existing abstractions in IDL and CCM.

Pluggable protocols [10] is an approach that can be used to integrating different distributed middleware technologies into any distributed middleware. The pluggable protocol approach operates at the protocol level for the middleware in that it transform data to different specifications before going over the network, or after coming from the network. Both CORBA and the Windows Communication Framework [21] support the notion of pluggable protocols. The CCM++ framework differs from the pluggable protocol approach in that the integration is done at the middleware-level. This therefore allows DRE system developers to configure the distributed system middleware being integrated into CCM. This is not possible using pluggable protocols since their integration level is below the middleware level. Moreover, the CCM++ approach extends the CCM specification and is more portable than the pluggable protocol approach, which is typically based on a proprietary framework.

VI. Concluding Remarks

The CORBA Component Model (CCM) is a specification that can be easily extended to provide more domain-specific functionality. As shown in this paper, CCM was extended to provide a component model for different distributed middleware technologies. Moreover, its fundamental abstractions were used to support heterogeneous composition of DRE systems. Based on experience gained from using the CCM++ framework to realize this capability, the following is a list of lessons learned and future research directions:

- **Integrating different platform independent models may violate the CCM specification.** The current distributed middleware technologies, besides TCP/IP, integrated into CCM all use the same platform independent model. Although the TCP/IP technology does not use the same platform independent model, the sending/receipt of an event does not result in transforming the event to between different platform independent models, which will result in performance degradation. Future work therefore includes investing the impact of integrating distributed middleware technologies that have platform independent models different from CCM, and understanding how the CCM specification may have to change to support such needs.
- **Fine-grain comparison of different distributed middleware technologies.** This is because the component servant can host any spec-compliant component implementation. For example, the results in the PingPong example were calculated by implementing the component implementation once, and auto-generating different component servants. Because of this design feature, distributed system developers can seamless evaluate the performance of different distributed middleware technologies and configurations under different component behaviors. This will help reduce cost and effort

associated with selecting the candidate technology that yields the best (or desired) performance results.

- **Supporting composition for remote method invocation connections.** The current design and functionality of CCM++ only supports event-based communication. Although this is the case, it is believe that the same approach can be applied to remote method invocation connections. Future work therefore is to investigate how CCM++ can be extended to support such connections.

CCM++ has been integrated into the CUTS system execution modeling tool, which is freely available for download from the following location: www.cs.iupui.edu/CUTS.

REFERENCES

[1] G. T. Heineman and W. T. Councill, Eds., *Component-based Software Engineering: Putting the Pieces Together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] S. Lacour, C. Perez, and T. Priol, "Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit," *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 35–49, 2004.

[3] T. Kichkaylo and V. Karamcheti, "Optimal Resource-Aware Deployment Planning for Component-Based Distributed Applications," in *High Performance Distributed Computing: Proceedings of the 13 th IEEE International Symposium on High Performance Distributed Computing*, vol. 4, no. 06, 2004, pp. 150–159.

[4] T. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," in *EWSA*, ser. Lecture Notes in Computer Science, vol. 3527/2005. Springer Berlin/Heidelberg, 2005, pp. 1–17.

[5] *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03 ed., Object Management Group, Nov. 2001.

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

[7] *CORBA Components v4.0*, OMG Document formal/2006-04-01 ed., Object Management Group, Apr. 2006.

[8] *The Common Object Request Broker: Architecture and Specification, Version 3.0*, Object Management Group, Jul. 2001.

[9] *Data Distribution Service for Real-time Systems Specification*, 1.2 ed., Object Management Group, Jan. 2007.

[10] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Proceedings of the IFIP* 6^{th} *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*. Salem, MA: IFIP, Aug. 1999.

[11] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*, 1st ed. Addison-Wesley Professional, 2009.

[12] K. Balasubramanian, D. C. Schmidt, Z. Molnar, and A. Ledeczi, "Component-based system integration via (meta)model composition," in *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 93–102.

[13] Prism Technologies, "OpenSplice Data Distribution Service," www.prismtechnologies.com/, 2006.

[14] Real-time Innovations, "NDDS: The Real-time Publish-Subscribe Middleware," www.rti.com/products/ndds/ndwp0899.pdf, 1999.

[15] C. Smith and L. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA, USA: Addison-Wesley Professional, September 2001.

[16] S. D. Huston, J. C. E. Johnson, and U. Syyid, *The ACE Programmer's Guide*. Boston: Addison-Wesley, 2002.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[18] K. Balasubramanian, "Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Sep. 2007.

[19] J. Snell and K. MacLeod, *Programming Web Applications with SOAP*. O'Reilly, 2001.

[20] *DDS for Lightweight CCM (DDS4CCM)*, ptc/2009-10-25 ed., Object Management Group, February 2009.

[21] Microsoft, "Windows communication foundation," msdn.microsoft.com/winfx/technologies/communication/default.aspx, 2006.