# Using Component-based Middleware to Design and Implement Data Distribution Service (DDS) Systems

Dennis Feiock and James H. Hill
Dept. of Computer and Information Science
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
Email: dfeiock@iupui.edu, hillj@cs.iupui.edu

***Abstract*—This short paper presents a framework named *integrated CCM (iCCM)* for integrating DDS into the CORBA Component Model (CCM). The goal of iCCM is (1) to promote reuse as opposed to reinvention without compromising performance; (2) reduce deployment and configuration complexities associated with DDS; and (3) to allow DRE system developers to focus more on the business-logic of the application instead of low-level implementation details.**

## I. Introduction

Enterprise distributed real-time and embedded (DRE) systems are increasing in both size and complexity [5]. Because of such advancements, traditional abstractions for implementing such systems are no longer sufficient. For example, client-server middleware, such as the Common Object Request Broker Architecture (CORBA) [10], was the *de facto* standard for implementing enterprise DRE systems. Because of emerging application domains and their problem space, it is now *hard* to use a one-fit-all-solution approach [7].

The Data Distribution Services (DDS) [9] is one such example of an emerging standard for implementing enterprise DRE systems. In particular, DDS is used to implement publisher-subscriber enterprise DRE systems. This is because traditional approaches, such as the client-server model realized by traditional CORBA, did not provide "easy to use" abstractions for this application domain. Instead, DRE system developers were using "homegrown" solutions that were plagued by many shortcomings experienced before client-server middleware emerged. Moreover, it resulted in re-invention of core-intellect within this domain.

Although DDS is improving development concerns of publisher-subscriber enterprise DRE systems, its current development model requires system developers to interact with low-level abstractions. For example, DRE system developers must *manually* manage subscriptions and must *manually* configure the publishers and subscribers. There have be proposed solutions for improving DDS's programming model [11] and using models to configure DDS application [4], but there is still disconnect between the programming model and the configuration model, and the solutions are *ad hoc* or require modification to an existing standard. In essence, DDS is experiencing the same "growing pains" that client-server middleware (*i.e.*, CORBA) experienced when it first appeared on the scene [3].

Although traditional middleware experienced the same difficulties that DDS is currently experiencing, the client-server programming model was able to overcome its limitations by increasing programming level-of-abstraction. For example, the component-based middleware [6] provided a solution that addressed many shortcomings of the traditional client-server programming model. Instead of wrestling with low-level implementation details, the component model allowed DRE system developers to focus mainly on the business-logic of the application. Concerns that were traditionally handled by the DRE system developer, such as deployment, configuration, lifecycle management, were now handled by the component middleware—and easily configurable.

This short paper therefore presents a framework for increasing the level-of-abstraction of DDS's programming model. The main contributions of this paper are as follows:

- It presents *integrated CCM (iCCM)*—a framework for integrating DDS into the CORBA Component Model (CCM) [10], which is a standard-based programming model for implementing component-based DRE systems;
- It is the first paper, to the best of the authors knowledge, on a systematic approach for integrating DDS into component-based middleware without requiring any modifications to either the CCM or DDS specifications;
- It discusses how the *Data Quality Modeling Language (DQML)* [4], which is a domain-specific modeling language for configuring DDS applications, was extended to support a component-based design methodology; and

**Paper organization.** The remainder of this paper is organized as follows: Section II discusses iCCM and its integration with DDS; Section III compares iCCM to other related works; and Section IV provides concluding remarks and lessons learned.

This work was sponsored in part by the Australian Defense Science and Technology Organization (DSTO).

## II. The Design and Functionality of iCCM

### A. Integrating DDS into CCM

To get a better understanding of how DDS can be integrated into CCM, it is first necessary to understand the structure of CCM, and how components send events to each other.

```
module Components {
  interface EventConsumerBase {
    void push_event (in EventBase evt)
      raises (BadEventType);
  };
};
```

Listing 1. The interface definition for a CCM event consumer.

Listing 1 shows the IDL for the `EventConsumerBase`, which is the object used to send events to other components. As shown in this listing, the interface contains a single method `push_event`. This method is invoked by one component whenever it needs to send an event to another component.

In order to integrate DDS into CCM, it is necessary to first extend this definition of an `EventConsumerBase` with constructs for establishing a DDS connection (*i.e.*, the publisher and subscriber communicate on the correct topic). The data type can easily be determined by the concrete event type. The topic, however, must be determined by either the publisher or subscriber.

```
module Components {
  module DDS {
    /// Extension interface for DDS.
    interface EventConsumer :
      ::Components::EventConsumerBase {
      void add_topic (in string topic_name);
      void remove_topic (in string topic_name);
    };
  };
};
```

Listing 2. The extended version of a CCM event consumer for DDS.

Listing 2 shows the extended version of the event consumer for DDS with methods for adding and removing topics by name. In this design, the event consumer is equivalent of a DDS subscriber. We call this design *publisher-oriented* because the publisher determines the connection details.

**Transforming DDS events to CCM events.** In CCM, components can use event types to communicate with each other. This is illustrated in Listing 1 where the `push_event()` method takes an `EventBase` as its only parameter. In DDS, however, events are not defined as event types, but as structures. Moreover, events in DDS have *keys*, which DDS implementations use to help with data management.

One of the major challenges when providing a standard method for integrating DDS into CCM is transforming events between the two technologies. If done incorrectly, systems implemented using iCCM will pay a performance penalty, *e.g.*, the tower-of-babel software anti-pattern [13]. Although this is a concern, it is possible to leverage the fact that both DDS and CCM use the same underlying *platform-specific model* [8]. It is therefore possible to define a CCM event in terms of an DDS event.

```
// StockInfo.idl
module Example {
  struct StockInfo {
    string symbol;
    unsigned long open, high, close;
  };

  // define the DDS key for this message
  #pragma keylist StockInfo symbol
};
```

Listing 3. An OpenSplice DDS event for publishing stock information.

For example, Listing 3 shows a stock event in OpenSplice (www.opensplice.org), which is an open-source implementation of DDS. Likewise, Listing 4 shows the equivalent of a CCM event in iCCM for the DDS event illustrated in Listing 3. As shown in Listing 4, an iCCM event replicates the data members defined for the target DDS event. By encapsulating the DDS events in the CCM event, DRE system developers are able to set the DDS event values directly. This alleviates the need for iCCM to physically translate the CCM event to a DDS event (or message) when the event is being published.

```
// StockInfoEvent.idl
module Example {
  // This event type is a wrapper for a DDS event.
  eventtype StockInfoEvent{
    public string symbol;
    public unsigned long open, high, close;
  };
};
```

Listing 4. CCM version of the DDS event from Listing 3.

Events received by a DDS data reader must also be transformed into a CCM event before the CCM event can be pushed to the component's implementation. This also presents an opportunity to transform the event, and pay a performance penalty. To overcome this design challenge, iCCM extends the CCM event class to read a DDS event. Listing 5 illustrates the DDS event wrapper class.

```
namespace Example
{
class StockInfoEventUpcall :
  public virtual StockInfoEvent,
  public virtual ::CORBA::DefaultValueRefCountBase
{
public:
  StockInfoEventUpcall (StockInfo & dds_event)
  : dds_event (dds_event) { }

  virtual void symbol (char * val) {
    this->dds_event_.symbol = val;
  }

  virtual void symbol (const char * val) {
    this->dds_event_.symbol = CORBA::string_dup (val);
  }

  virtual void symbol (const ::CORBA::String_var & val) {
    ::CORBA::String_var dup = val;
    this->dds_event_.symbol = dup._retn ();
  }

  virtual const char * symbol (void) const {
    return this->dds_event_.symbol.in ();
  }

  // ...

private:
  // The contained DDS event object
  StockInfo & dds_event_;
};
}
```

Listing 5. iCCM's wrapper that transforms a CCM event to a DDS event.

A wrapper class is generated for each eventtype specified in the IDL. The wrapper class has a single member variable that is a reference to the DDS event recently read from the DDS data reader. This means that the event for this wrapper must be set at construction time. Finally, the wrapper class implements the CCM getters setters for all data members, which provides the component with access to the encapsulated events. This means that it is possible for a component to receive a DDS event, and resend the same event (or make changes to the existing data) without making a deep copy of the data.

Lastly, the downcall event wrapper class is similar to the upcall event wrapper class illustrated in Listing 5. The main different, however, is instead of storing a reference to an existing DDS event, the downcall wrapper class statically declares a DDS event internally. Similar to the DDS upcall event wrapper class, the implementation of the event's methods are delegated to the internal DDS event.

**Private vs. non-private topics.** As discussed above, iCCM uses publisher-oriented connections. Because topics are determined by the publisher, iCCM suppors to kinds of topics: *private* and *non-private*. Private topics are topics that are bound to a specific component. Non-private topics are those that are accessible by any component's event ports. In iCCM, the name of the input event port maps to the topic's name. If the topic name is prefixed with the component instance name, which is unique across the entire system, then the topic is considered a private topic. If no such prefix exists, then the topic is considered a non-private topic. By default, all topics in iCCM are non-private topics unless overridden at deployment and configuration time (see Section II-B).

**Implementing a DDS component using iCCM.** The discussion above pertained to how iCCM integrates DDS into CCM, which takes place at the component servant level. The component servant in CCM is typically auto-generated from IDL. Keeping inline with these expectations, iCCM auto-generates a CCM servant that can send DDS events. The auto-generated servant uses the iCCM abstractions previously discussed.

```
1  module Example {
2    component StockDistributor {
3      publishes StockInfo stock_notify;
4    };
5
6    component StockBroker {
7      consumes StockInfo stock_info;
8    };
9  };
```

Listing 6. CCM IDL specification for two components.

Because the component servant is auto-generated, developers are responsible for implementing only the component implementation (in addition to the component's specification in IDL). For example, Listing 6 presents the IDL specification for two components that either send (publish) or receive (subscribe to) an event. iCCM uses the specification in Listing 6 to auto-generate the stubs, skeleton, and component servant.

```
1
2  void StockDistributor_Impl::auto_notify_thread (void) {
3    // create a new event.
4    ::Example::StockNotifyEvent_var ev (
5      this->ctx_->new_stock_notify_event ());
6
7    // Set the events values.
8    ev->symbol ("GOOG"); ev->open (578);
9    ev->high (580); ev->low (573);
10
11   // Send the event. The underlying servant will send
12   // this event as a DDS message!
13   this->ctx_->push_stock_notify (ev);
14 }
```

Listing 7. Code snippet of the `StockDistributor` CCM component that publishes the DDS `StockInfo` message.

Listing 7 shows a portion of the `StockDistributor` component implementation that DRE system developers must implement based on the IDL presented in Listing 6. As shown in this example, a component's implementation is the same as a standard CCM component's implementation.

### *B. Deploying and Configuring iCCM Systems*

To assist with the deployment and configuration phase, iCCM uses the *Deployment And Configuration Engine (DAnCE)* [2], which is an open-source tools that implements the OMG D&C specification [12], to deploy its DDS-based components. The iCCM components "as is" can be deployed by DAnCE without modifying DAnCE. To configue DDS QoS parameters, however, iCCM extends DAnCE's deployment and configuration handlers.

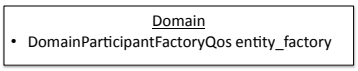


Fig. 1. Metamodel for iCCM's domain specification.

To configure the DDS QoS parameters, iCCM uses two different XML specifications based on the DDS QoS data model. The composition of the XML specification is also inline with iCCM's component-based design. The first specification is the *domain specification*, which is illustrated in Figure 1. The domain model is simple in that it determines if domain entities, such as DDS participants, are auto-enabled at creation time. The domain specification is applied by adding an iCCM-specific attribute to the locality manager with the name of `DDSQoS`, and setting its value to the location of the domain specification file.

The second specification is called the *participant specification*, which is shown in Figure 2. The participant specification configures different QoS parameters for DDS entities of a DDS participant, such as data reader and writer partitions, participant QoS, and data reader and writer QoS. Because the design is publisher-oriented, the data writer determines the topic's QoS if it is a non-private topic. If the topic is a non-private topic, then its QoS is determined at the participant-level (*i.e.*, the entity that creates the topic). Finally, the participant specification is applied by setting a component instance's attribute named `DDSQoS` to the location of the participant specification file. iCCM's configuration handlers then use the specified file to configure the entities accordingly.

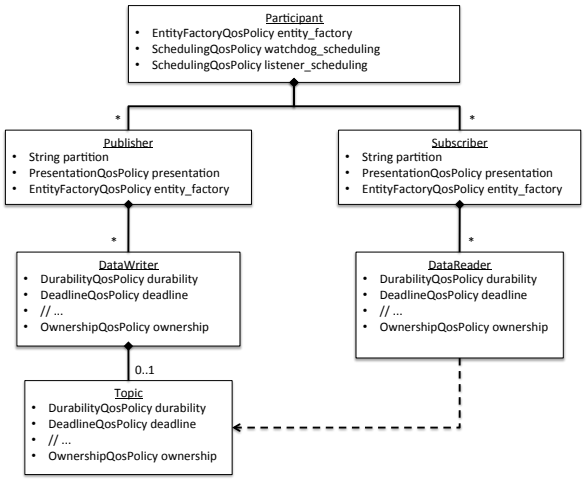


Fig. 2. Metamodel for iCCM's participant specification.

**Enforcing D&C semantics.** To assist with valid construction of the domain and participant specification and enforcing semantics, we first leverage the *Platform Independent Component Modeling Language (PICML)* [1]. PICML is used to model the systems' composition. The resultant model is then transformed to an extended the version of the *Data QoS Modeling Language (DQML)* [4]. The extension of DSML was necessary because its current version uses a flat data model that is consistent with the traditional approach for implementing DDS applications. iCCM, however, uses a component-based approach. The metamodel for the extended version of DQML was originally shown in Figure 1 and Figure 2.

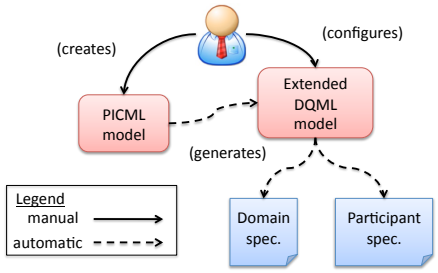


Fig. 3. iCCM's workflow for modeling the deployment and configuration of its applications.

Using the workflow shown in Figure 3, DRE system developers for model their component-based system in PICML. Then using the PICML model, model intepreters automatically transformed into a DQML model as where each component instance in PICML is mapped to a DDS domain participant. Each event port is mapped to either a data reader or data writer depending on the event port semantics. The developer then sets the each entities QoS in the auto-generated DQML model. Finally, the DQML model is transformed to corresponding domain and participant specification file.

## III. Related Works

DDS4CCM [11] uses a gateway approach to integrating DDS into CCM This means that events are sent to a mediator and transformed and then transformed into DDS messages. iCCM's approach differs from the DDS4CCM approach in that it (1) performs event transformation in the component's servant, as opposed to in another object; 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.

## IV. Concluding Remarks

This short paper presented *integrated CCM (iCCM)*, which is a method for integrating DDS into CCM. By integrating DDS into CCM, DRE system developers are able to focus on the system's business-logic instead of wrestling with low level implementation details. Although iCCM is able to abstract away the low-level complexities of DDS, there is still much work to be done to realize a optimized and lightweight solution that allows DRE system developers to leverage all aspects of DDS via iCCM. iCCM is currently integrated into the CUTS system execution modeling tool. It is freely available for download from the following location: cuts.cs.iupui.edu.

## References

[1] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society.

[2] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, Nov. 2005.

[3] M. Henning. The Rise and Fall of CORBA. *Queue*, 4(5):28–34, 2006.

[4] J. Hoffert, D. Schmidt, and A. Gokhale. A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms. In *Proceedings of International Conference on Distributed Event-Based Systems (DEBS)*, pages 140–145, Toronto, Canada, June 2007.

[5] S. E. Institute. Ultra-Large-Scale Systems: Software Challenge of the Future. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, June 2006.

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

[7] A. M. Memon, A. Porter, and D. Scmidt. Feedback-driven design of distributed real-time and embedded component middleware via model-integrated computing and distributed continuous quality assurance. In *Proceedings of The National Science Foundation Invitational Workshop – Science of Design: Software-Intensive Systems*, nov 2003.

[8] Object Management Group. *Model Driven Architecture (MDA) Guide V1.0.1*, OMG Document omg/03-06-01 edition, June 2001.

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

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 edition, Jan. 2008.

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

[12] OMG. *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 edition, Apr. 2006.

[13] C. Smith and L. Williams. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *CMG*, volume 2, pages 667–674, Dallas, TX, 2003.