Event Channel Configuration in Cadena *

Gurdip Singh, Prashant S. Kumar & Qiang Zeng
234 Nichols Hall, Computing and Information Sciences
Kansas State University
Manhattan, Kansas 66506
{singh,prashant,qzeng}@cis.ksu.edu

Abstract

Distributed systems typically involve a large number of interacting components, and developing such systems is a challenging task. We are building Cadena – an integrated environment for designing and building Corba Component Model (CCM) based systems. Cadena provides facilities for specifying CCM components and their interconnections and for analyzing and deploying assembled applications. The deployment typically requires the use of distributed middleware services to implement application requirements such as data distribution and event communication. To satisfy performance goals, however, a designer is often faced with choosing between a number of possible middleware services or specialized version of the same service. Writing code to deploy the system (glue code to interconnect components via middleware services) and to configure middleware services can be a tedious and an error-prone task. To address these issues, we propose an event communication framework for implementing event communication between components in Cadena. The framework includes a Corba-based event channel which has been integrated into the OpenCCM infrastructure used by Cadena. The framework also provides a set of alternative light-weight mechanisms to implement the event service functionality. We have evaluated the performance of these mechanisms, and have identified the context in which each one is suitable. The framework enables us to implement algorithms in Cadena to analyze an application to identify the mechanisms best suited for implementing the different event connections, and to automatically synthesize code and metadata to configure the containers to used the appropriate mechanism for each connection.

*This work was supported by DARPA PCES contract F33615-00-C-3044 and NSF awards EIA9980321 and CCR0098179.

1 Introduction

Distributed systems often involve a large number of components, distributed across several processing nodes. The components may interact with one another in complex ways via synchronous as well as asynchronous communication mechanisms. Writing code to interconnect the components via appropriate communication mechanisms can be a tedious exercise. This task however has been simplified by the advent of a number of frameworks such as CORBA Component model (CCM), Enterprise JavaBeans (EJB), and Distributed Component Object Model (DCOM). In addition to automated configuration code generation tools, these frameworks provide middleware services for aspects such as event notification, distributed deployment, transactions and data sharing.

To satisfy performance goals, however, a designer may be faced with the choice between a number of different middleware services or different specialized variations of the same service. For example, we have been studying the Boeing Bold Stroke System – a platform to develop avionics applications [Sha99]. Components in a Bold Stroke system interact by events as well as synchronous method calls. Using a middleware service such as an event service greatly simplify the implementation of Bold Stroke applications. However, Bold Stroke developers have identified several special cases where specialized configuration of the event service is preferred for performance optimization. With scenarios involving a large number of components, identifying such optimization opportunities and subsequently writing the code to connect different components via different specialized versions can be a tedious and an error prone task. Therefore, tools need to be developed to identify optimization opportunities and perform automatic configuration and code generation.

We are developing Cadena - a modeling and deploying tool to build CCM based systems. Cadena provides facilities for defining component types using CCM IDL, and assembling systems from CCM components by specifying the connections between the data and event ports of the components. Cadena also provides code generation and deployment facilities to implement application scenarios. We use OpenCCM CCM IDL to Java compiler for the component stub and skeleton code generation [GOA02]. Configuration code to create component instances, interconnect the components, and configure the containers and middleware services to implement communication is also generated.

An important part of the deployment is the implementation of the event and data connections via the underlying middleware services. CCM defines a container based framework wherein each container interacts with the underlying middleware services and provides simplified service access to its component [Gro02, WR02]. In this paper, we concentrate on event communication middleware. Event based communication has been used extensively in component based systems and several implementations of the event service compliant with the OMG standard are available [HLS97, Sch94, MSS97, HBBM96, CNF98, CRW99, KR95]. To study issues of middleware optimization and configuration, we propose an event communication framework (ECF) which supports several mechanisms for implementing event connections in Cadena. In particular, the framework includes a Corba-based Adaptive Event Service (AES). We describe

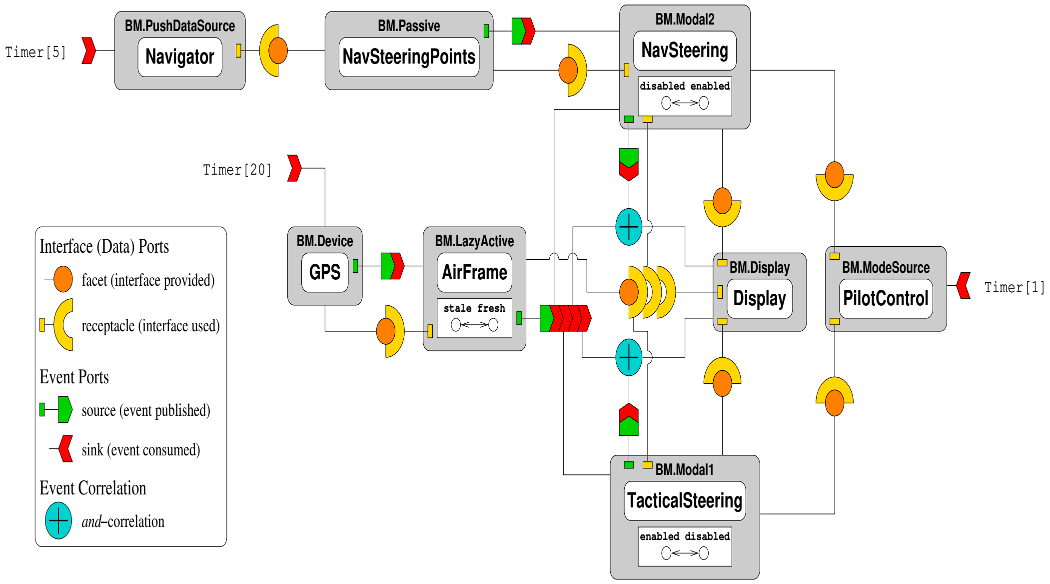

Figure 1: Simple avionics system

the integration of AES into OpenCCM containers used in Cadena and the generation the necessary configuration (glue) code and XML metadata to configure the containers and the AES.

Although AES provides support for implementing event connections, alternative (lightweight) implementations of event channel features may be preferable in different application contexts. These features include maintaining subscriber lists, correlation, filtering and distributed notification. For example, it may be more efficient to do event notifications via a direct method call from a producer to a consumer if both are residing on the same processor rather than via AES. We have incorporated a number of such alternative implementations for event channel features in ECF, and have evaluated their performance to determine the context in which each performs better than its alternatives. For example, with respect to correlation, we have evaluated alternatives that implement correlation in a correlator component, correlation in the event channel and correlation in the containers. Our results show that different lightweight implementations can be exploited to optimize performance.

The availability of alternative implementations in ECF allows us to implement algorithms in Cadena that can analyze the application scenario to determine the best possible configuration for each event connection. However, to use different implementations for different event connections in the same application, we need to configure the containers to use the appropriate implementation. Algorithms have been implemented in Cadena to identify optimized implementations for event connections and to generate configuration code and metadata for the OpenCCM containers to use the appropriate implementations.

2 Cadena overview

Cadena is an integrated modeling environment for modeling and building CCM systems. It provides facilities for defining component types using CCM IDL, assembling systems from CCM components, producing CORBA stubs and skeletons implemented in Java, and generating deployment and configuration code. To describe the features of Cadena, we will use as an example of a simple avionics system that shows steering cues on a pilot's navigational display (detailed examples are available at http://www.cis.ksu.edu/cadena). The pilot can choose between two different display modes — each mode yields a different set of steering cues. A *tactical* display mode displays cues related to a tactical *mission* objective. A *navigation* display mode displays cues related to a navigational objective. Cues for the navigation display are derived in part from navigation steering points data that can be entered by the navigator.

Figure 1 presents the CCM architecture for the example system. The system is realized as a collection of components coupled together via data and event connections. Input position data is gathered periodically at a rate of 20 Hz in the `GPS` component and then passed to an intermediate `AirFrame` component (which in a more realistic system would take position data from a variety of other sensors). Both the `NavSteering` and `TacticalSteering` component produce cue data for `Display` based on air frame position data. The `Navigator` component polls for inputs from the plane navigator at a rate of 5 Hz that are used to form `NavSteeringPoints` data. This data is then used to form navigational steering cues in `NavSteering`. `PilotControl` polls for a pilot steering mode at a rate of 1 Hz and enables or disables `NavSteering` and `TacticalSteering` accordingly.

Figure 2(a) gives the CCM IDL that defines the component types `BMLazyActive` and `BMModal1` for the `AirFrame` and `TacticalSteering` component instances in Figure 1. CCM components *provide* interfaces to clients on ports referred to as *facets*, and *use* interfaces provided by other clients on ports referred to as *receptacles*. Components *publish* events on ports referred to as *event sources*, and *consume* events on ports referred to as *event sinks*. In the `BMLazyActive` component type of Figure 2, `dataOut` is the name of a facet with interface type `DataAvailable`, and `dataIn` is the name of a receptacle with interface type `DataAvailable`. Similarly, `inDataAvailable` is the name of an event sink of type `DataAvailable`, and `outDataAvailable` is the name of a event source of type `DataAvailable`.

Figure 2(b) displays a fragment of the textual Cadena Assembly Description (CAD) for the example system in Cadena. Cadena also provides graphical and form-based CAD specification facilities with the abstract syntax tree of the textual form providing the canonical representation. The assembly description for a particular scenario (application) in CAD involves specifying the instances of the components to be created and the connections between the ports of the component. For each component instance, a developer specifies an abstract location upon which the instance is to be allocated. Abstract locations are mapped to CORBA specific notions such as component servers at deployment time. For data and event sink ports, a `connect` clause declares a connection between a port of the current instance and a port of the component that provides the interface/event (see Figure 2(b)).

```
\#pragma prefix "cadena"
module modalsp {
 interface ReadData {
  readonly attribute any data;
 };

 eventtype TimeOut {};
 eventtype DataAvailable {};

 enum LazyActiveMode {stale, fresh};
 component BMLazyActive {
  provides  ReadData dataOut;
  uses      ReadData dataIn;
  publishes DataAvailable outDataAvailable;
  consumes  DataAvailable inDataAvailable;
  attribute LazyActiveMode dataStatus;
 };

 enum OnOffMode {enabled, disabled};
 interface ChangeMode {
  attribute OnOffMode modeVar;
 };

 component BMModal1 {
  provides  ChangeMode modeChange;
  provides  ReadData dataOut;
  uses      ReadData dataIn;
  publishes DataAvailable outDataAvailable;
  consumes  DataAvailable inDataAvailable;
 };
};
```

(a)

```
system ModalSPScenario {
 import cadena.common, cadena.modalsp;

 Rates 1, 5, 20;         // Hz rate groups
 Locations l1, l2, l3;  // abstract deployment locs
 ...
 Instance AirFrame implements BMLazyActive on #LAloc {
  connect this.inDataAvailable
       to GPS.outDataAvailable runRate #LArate;
  connect this.dataIn to GPS.dataOut;
  }
 Instance TacticalSteering implements BMModal1 on l2 {
  connect this.inDataAvailable
       to AirFrame.outDataAvailable runRate 5;
  connect this.dataIn to AirFrame.dataOut;
 }
 ...
}
```

(b)

Figure 2: (a) CCM/Cadena artifacts, (b) Cadena Assembly Description for ModalSP (excerpts)

3 Code Generation in Cadena

In the previous section, we discussed the specification/design stage in Cadena where the designer specifies and analyzes the application scenario. Cadena provides a number of analysis tools to analyze and validate the specifications [HDD+03]. The next step is the generation the code for deployment and configuration. The overall code generation process in Cadena is shown in Figure 3 (the code architecture is shown in Figure 4). We use the OpenCCM CCM IDL to Java compiler for component stub and skeleton generation (the solid arrows in Figure 3 denote the use of OpenCCM tools). ¿From the IDL3 description, the code architecture generated for components and containers is shown in Figure 5 (only some files relevant to event communication are shown; the code architecture corresponds to OpenCCM 0.4). `GPSImpl` and `AirFrameImpl` are the implementation files into which the designer is expected to fill in the business logic (in addition to specifying the scenario, this is the only code that the designer has to write). The files `Context` and `Wrapper` are the ones that interact with the component to provide access to the underlying services to the component.

The code generation phase results in a set of Java files. The actual instances of the components and containers are created by a *configuration file*. In Cadena, the CAD file specifying the component instances and the interconnections is parsed and an abstract syntax tree (AST) is generated (denoted by dotted lines in Figure 3). From this AST , as shown in Figure 3,

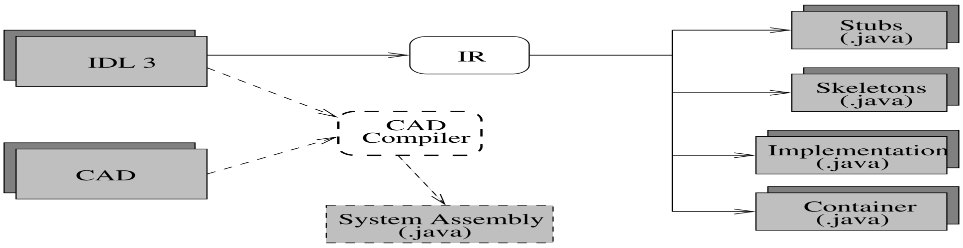

Figure 3: Code Generation in Cadena

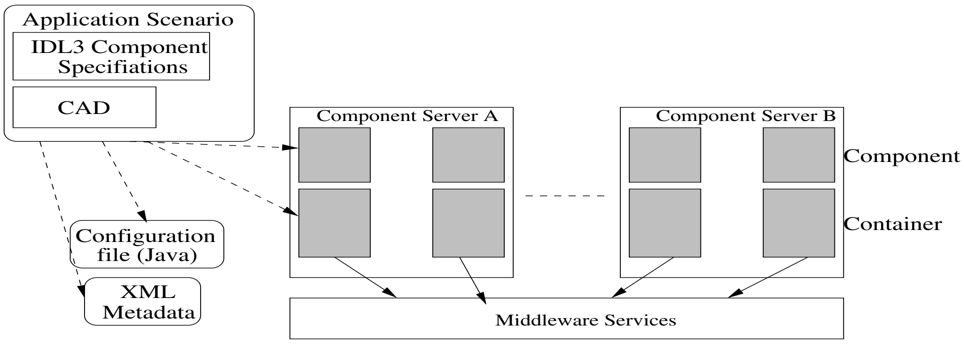

Figure 4: Deployment architecture

the CAD compiler generates a configuration (Java) file that contains code for assembling the system (by performing activities such as create component servers, creating homes for the components, creating components, connecting event source ports to sink ports and connecting facets to receptacles). Figure 6 gives an excerpt of the configuration file. As shown in the figure, the configuration file first creates the component servers, and then the components on each component server.

4 Event Service Overview

As shown in Figure 4, the containers interact with the underlying middleware services (or themselves provide these services) to implement the event and data connections. Since we want to allow the containers to use different middleware services depending on the application scenario, the container code needs to be configured using application-specific metadata. In this following, we study the integration of event channel middleware with Cadena, and address the issue of configuring the containers and the middleware.

Event-based communication has been used extensively in component-based systems where events may be used by components to announce data availability or state changes [HLS97, CNF98, KR95, CRW99, HBBM96]. Event service based communication is very well suited for such systems as it provides anonymous, one-way communication. In an event service,

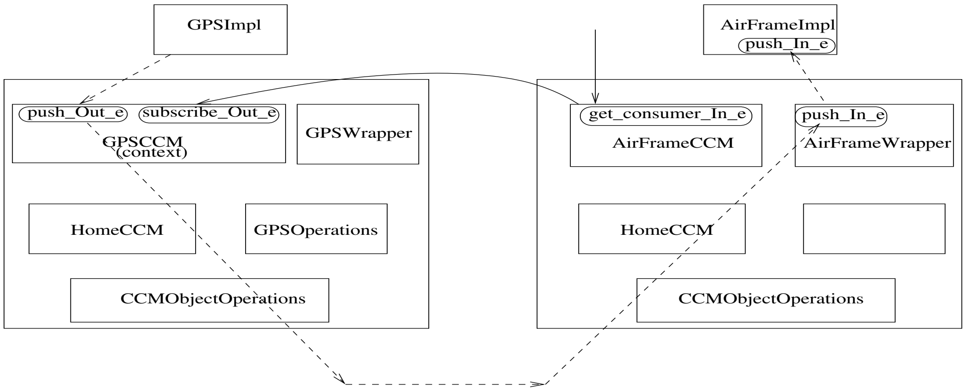

Figure 5: Code architecture for a component

an entity can register with the event channel as a producer of an event or as a subscriber of an event. Whenever a producer produces an event, all current subscribers for that event are notified of the event occurrence. Tao's Real-time event service has been widely used in designing distributed real-time embedded systems [HLS97]. We have developed a Java-based event service, Adaptive Event Service (AES), that has an architecture similar to that of Tao's event service. Figure 7 shows the structure of AES. Each producer has a proxy object in the channel that handles events published by the producer. The correlation and synchronization module allows subscription to composite events (conjunction or disjunction of events) and condition events (these are discussed later in more detail). The dispatching module is responsible for dispatching events to the consumers. The event channel can be configured to support various strategies for event dispatching as in [HLS97]. For producers and consumers not residing on the same machine, we use the notion of gateways for efficient event notification across processor boundaries (this aspect is discussed in more detail in Section 5.2). AES is a mode-aware event service that allows it to adapt to the application modes to quench the flow of notifications not required in specific modes.

4.1 AES Integration in Cadena

We now discuss the integration of AES into OpenCCM containers in Cadena. It is possible to simply use AES as a stand alone CORBA service for event communication. In this case, each container would interact with AES via the ORB. This, however, is inefficient as every notification will be a remote call via the ORB. Therefore, we adopted an architecture wherein an event cannel is created in each component server as shown in Figure 8. The components on the same component server communicate via the local event channel. Event communication between components on different component servers is via gateways, which is described later in Section 5.2.

We follow a pluggable-module approach to integrate middleware services with the containers. Each middleware service must provide a plug-in interface module with which the container

```
package cisksu.basicsp;
public class BasicSP
{
 public static void main(String[] args)
   {
     Init the OpenCCM Components Runtime.
     create the component servers.
     Obtain the container homes and archive installators.
     Install archives.
 //Instantiating homes...
   :
   h = server2_cont.install_home("basicsp", "cisksu.basicsp.BMClosedEDHomeImpl.create_home",
                           new org.omg.Components.ConfigValue[0]);
    BMClosedEDHome bmclosed = BMClosedEDHomeHelper.narrow(h);
    :
 // Create components.
   BMTimeOut event = sh.create();
   BMClosedED airframe = bmclosed.create();
   BMDevice gps = bmdevice.create();
   BMDisplay Display = bmdisplay.create();

 // Connect the components.
   event.subscribe_timeOut20(gps.get_consumer_timeOut20());
   gps.subscribe_outDataAvailable(airframe.get_consumer_inDataAvailable());
   airframe.subscribe_outDataAvailable(Display.get_consumer_inDataAvailable());
   ReadData dataOut = gps.provide_dataOut();
   airframe.connect_dataIn(dataOut);
   dataOut = airframe.provide_dataOut();
   Display.connect_dataIn(dataOut);
   :
}
}
```

Figure 6: Configuration file(excerpts)

can interact. The plug-in module must contain the code to connect and interact with the corresponding middleware service. Figure 8 shows the event channel plug-in module, ECP, used to integrate AES. To allow the flexibility to use different middleware services, the container must be configured to use the appropriate plug-ins for the various connections. As shown in Figure 8, Cadena tools generate an XML file with meta-data necessary to configure the containers. Figure 9 gives an excerpt of this file that contains information for each port. For example, the event channel processes event based on their ids whereas at the CCM level, events are identified by their types and the port producing the event. Therefore, we need to specify the mapping of CCM events to event ids which is provided in the XML file. Other important attributes include priority of the events (rate) and the synchronization id (this determines whether the event is involved in correlation, synchronization or mode-control; these features are discussed in later sections). Some of the attributes (such as rate and event_name) are obtained from the CAD specification, whereas others are inferred by analysis algorithms (for example, an analysis algorithm will compute the “service” tag by determining the best possible event channel implementation for that connection).

The container reads the XML file and determines the plug-in to which it must delegate the calls for different connections. The XML meta-data information for the event ports is also used by the ECP to make the appropriate connections to the event channel. Based

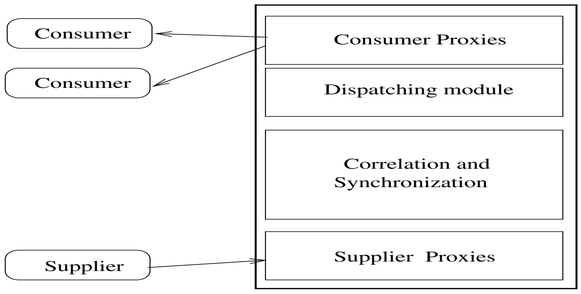

Figure 7: Architecture of the Event Service

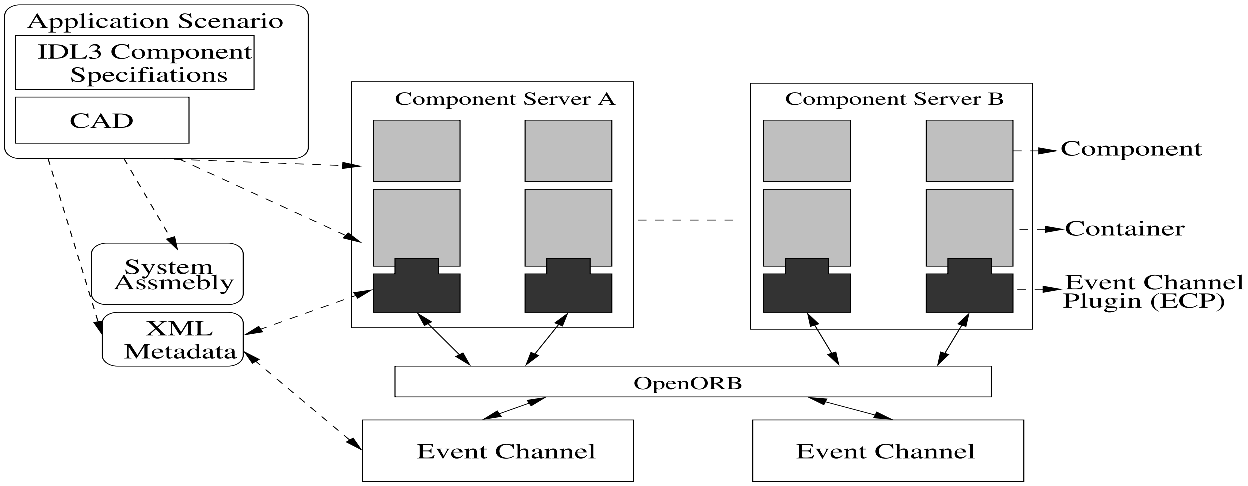

Figure 8: Event Channel Integration

on the role of the component, if producer, ECP creates a producer proxy for the component. This is done by invoking eventChannel.for_suppliers().obtain_push_consumer(), where eventChannel is the reference retrieved from the Naming service. This proxy reference would be used for subsequent 'push' to the event channel. A consumer side proxy is created for each CCM event type and registered with the event channel using the method eventChannel.for_consumers().obtain_push_supplier(). The component is connected as a push consumer by passing the reference to the event channel (which is obtained from the container as the container maintains a list of these ports).

When a publisher is ready to push an event, the component pushes the CCM event to its container. This call is further delegated to the ECP. Since AES operates on the basis of integer type event id, the ECP looks up the map generated during configuration time and retrieve the associated event id for the CCM event. Since the consumer component is designed to receive only a CCM event, the CCM event is included a payload in an AES event. The event is structured in a CORBA Any and pushed to the producer side proxy. The event channel pushes the event to the consumer where the CCM event payload is unpacked and the appropriate push method of the component is invoked.

We have completed the integration of AES in Cadena (and is available in the latest Cadena

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
:
:
<connection_spec>
<component_name>GPS</component_name>
<component_id>2</component_id>
<service>AES</service>
<role>1</role>
<event_name>cadena.common.DataAvailable</event_name>
<port_name>outDataAvailable</port_name>
<event_id>2</event_id>
<sync_id>0</sync_id>
<rate>20</rate>
</connection_spec>
<connection_spec>
<component_name>AirFrame</component_name>
<component_id>3</component_id>
<service>AES</service>
<role>2</role>
<event_name>cadena.common.DataAvailable</event_name>
<port_name>inDataAvailable</port_name>
<event_id>2</event_id>
<sync_id>0</sync_id>
<rate> 20</rate>
</connection_spec>
:
:
</configuration>
```

Figure 9: XML metadata for Event Channel

release) and it includes the generation of the configuration code and XML metadata. The metadata configures the container to interact with the ECP, and configures the ECP to make the appropriate connections with the AES. As discussed in a later section, XML metadata is also generated to configure AES for correlation, mode-awareness and distributed event notifications. Thus, the designer essentially designs the system by specifying the component instances (by either first defining the components in IDL3 or by taking components from the library) and the interconnections. All of the subsequent steps for configuration and deployment are automated.

5 Event Channel Optimizations

Although AES provides support for implementing event connections, alternative (lightweight) implementations of the event service may be preferable in specific application contexts. In this section, we explore such alternative implementations and evaluate their performance. Our goal is to add several alternative implementations to ECF and develop algorithms that can analyze the application scenario to determine the most suitable implementation available in the ECF for the different event connections.

5.1 Direct-Dispatching

A basic event service (for example, without correlation support) can be implemented by maintaining subscriber lists in the container of the producer components and making notifications by direct dispatching (direct method calls from producer containers to consumer containers). In the following, we give a brief overview of this mechanism in OpenCCM [GOA02]. In the OpenCCM code generation process, for an output event port, say `Out_e` of GPS, a method `subscribe_Out_e` is added in GPSCCM (context) that can be invoked to subscribe to this event (see Figure 3). In addition, a method `push_Out_e` is added to GPScontext that is invoked by GPS to publish the event. At the consumer side, for an event sink port, say `In_e` of the AirFrame in Figure 3, a method get_consumer_In_e is added to AirFrameContext that returns the stub reference to that object. This reference is passed to the subscriber to receive the notification.

To configure the containers to use direct dispatching, we add to the configuration file the method call, `subscribe_Out_e` of GPS with the Corba stub reference of AirFrame, obtained via method get_consumer_In_e, as a parameter (see solid lines in Figure 6). The GPS container stores these references in a subscriber list. When the GPS is ready to push an event, its container delegates the call to the direct dispatch plug-in module (a plug-in is needed to integrate this scheme). This module retrieves the list of subscribers from the container by invoking _get_subscribers(publisher name). Subsequently, it cycles through the retrieved list and makes a push_call on each of the consumers (see dashed lines in Figure 6). OpenCCM's colocation strategy determines if the call is remote or a local call and delivers the event.

In the BoldStroke environment, direct dispatching is also used in addition to event channel notification [Sha99]. The notifications via direct dispatch are termed ERM (Entity Resource Manager) calls. In the Boldstroke applications, if the producer and consumer ports are on the same server and are at the same priority level, and the notification does not involve correlation, then the call can be tagged as ERM. With smaller scenarios (10-20 components), it may be possible to perform such tagging via visual inspection. This however becomes impractical with larger scenarios, and analysis algorithms are needed that can identify such oppurtunities. An algorithm has been implemented in Cadena that identifies the notifications that can be performed by direct dispatching using the criteria used the BoldStroke system. The algorithm performs analysis on the application scenarios and annotates each event connection in the CAD XML file with appropriate tags (ERM or AES). The container then reads the XML file and interacts with the appropriate plug-ins (AES plug-in or DD plug-in).

Figure 10 gives the performance results for an application consisting of one producer and

Consumers	1	4	8	16	32	64	128
DD	172	235	468	907	1797	3656	7422
AES	266	703	1078	1875	3517	6703	13453

Figure 10: Event Notification within a single component server

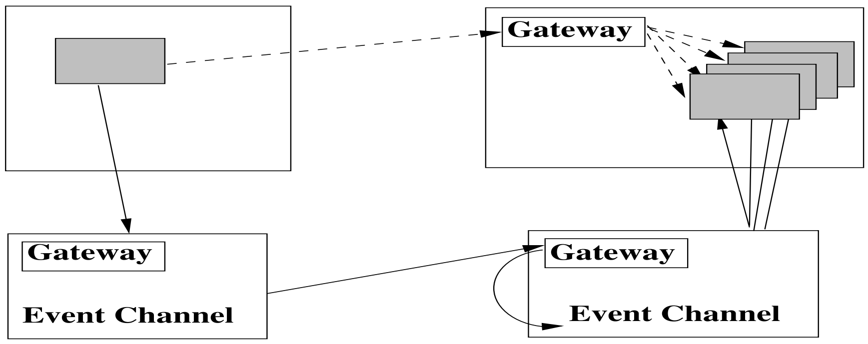

Figure 11: Event Distribution with gateways

multiple consumers, all on the same component server, for direct dispatching (DD) and AES-based notification. The experiment (all subsequent experiments in the paper) involves the producer generating 1000 events and values are time taken in milliseconds from the first event generated to the last notification received (the measurements actually start from 500th event up to event number 1500). The experiments were carried out on machines running XP with Intel P4, 2.52GHz processor. For method calls between components on the same component server, OpenCCM optimizes them as local calls (that is, they are not ORB calls). Thus, all notifications via DD are local calls in this experiment. Similarly, since an event channel is created in each component server, all method calls for AES are also local. However, a notification in AES involves a number of method calls to traverse the various event channel layers. Thus, as can be seen in Figure 10, AES notification is more expensive as compared to direct dispatching. Thus, it is advantageous to tag calls as ERM whenever possible.

5.2 Distributed Event Notification

AES uses the notion of gateways for notification of events across processor boundaries (which has also been used in the ACE/TAO event service [HLS97]). A *gateway* is a module that subscribes to remote events and then distributes them locally. In Cadena, the CAD file is scanned to determine the set, $subscribe(X, Y)$, of events from components on server Y that are consumed by components on server X. The gateway at server X is then configured to subscribe to the event channel at server Y for each event in $subscribe(X, Y)$. Thus, when an event e in $subscribe(X, Y)$ is published by a component residing in Y, the gateway at X is notified of this occurrence. This gateway then locally publishes the events with its event channel which in turn notifies the local components. The solid lines in Figure 11 gives the event notification flow with gateways.

Consumers	1	4	8	16	32	64	128
DD	1453	2423	4625	9016	16437	32281	63578
AES with gateway	2953	3281	3704	4516	6234	9875	17500
Gateway as a component	1640	1750	1891	2375	3171	5203	9969

Figure 12: Event Notification with Distribution

Consumers	1	4	8
DD	2891	9593	18203
AES with gateway	10206	12250	14562

Figure 13: Event Notification with Distribution with multiple producers

Let a be a producer of an event e on component server Y and and B be the set of consumers of event e on server X. As discussed above, with a gateway-based implementation, the gateway will get a $single$ notification from a, which is then locally distributed to the components in the set B. An alternative mechanism is to simply perform direct dispatching via ORB calls. In this case, each component in B will be individually notified via direct dispatching whenever the event e is published by a. Figure 12 gives the performance of direct-dispatching vs gateway-based implementation. As can be seen, the gateway-based implementation (second row) performs better as compared to direct dispatching (first row) as the number of consumers increase on server X as there is only one notification across the server boundary. As the number of consumers increase, the only increase incurred is the cost of additional local notifications (as one can see, the increments in the second row in Figure 12 are the same as in the second row in Figure 10). We also ran the experiment with 4 producers on one machine with varying number of consumers (1,4 and 8) for each of the producers. The performance results, given in Figure 13, also validate the same conclusion.

It is possible to incorporate the gateway concept in the direct-dispatching scheme. From the CAD file, a "gateway component" can be synthesized that subscribes to remote events and then locally distributes the notifications. The dashed lines in Figure 11 illustrates the flow of notification in this scheme. For experimentation purposes, the steps to generate the gateway component were carried out manually. The last row in Figure 12 gives the performance of this implementation with varying number of consumers. Clearly, this scheme outperforms the other two. Although the gateway component can be generated automatically, it does require significant "rewiring" of the connections (which can add complexity to the code). In the presence of multiple component servers, however, several techniques for efficient event distribution based on arranging component servers in topologies such as trees and star have been studied and are available [CRW99]. Designing gateway components and arranging them in such topologies is likely to be complex. Thus, we feel that "gateway as a component" optimization can definitely be exploited in scenarios with small number of component servers. More experimentation is needed to study this scheme for large number of servers.

```
<correlation_spec>
  <correlation name="first" type="|">
    <out id="7" location="ComponentServer3"/>
       <in1 id="10" location="ComponentServer1" simpleevent="false">
          <correlation name="firstsub1" type="+">
             <out id="10" location="ComponentServer1"/>
               <in1 id="3" location="ComponentServer1" simpleevent="true"/>
               <in2 id="4" location="ComponentServer1" simpleevent="true"/>
           </correlation>
        </in1>
        <in2 id="20" location="ComponentServer2" simpleevent="false">
           <correlation name="firstsub2" type="+">
              <out id="20" location="ComponentServer2"/>
              <in1 id="5" location="ComponentServer2" simpleevent="true"/>
              <in2 id="6" location="ComponentServer2" simpleevent="true"/>
            </correlation>
        </in2>
  </correlation>
</correlation_spec>
```

Figure 14: Correlation Specification

6 Correlation

Correlation provides a mechanism for components to monitor occurrence of event patterns. For example, And-correlation (e1 and e2) and Or-correlation (e1 or e2), are two commonly available correlation expressions [HLS97]. A component subscribing to "e1 and e2" receives a notification when both e1 and e2 have been published (Or-correlation is defined similarly). In general, a correlation expression may be a nested expression involving and and or operators. In Cadena, we allow the designer to specify special components identified as correlators and associate correlation expressions with them. The set of correlation expressions, CE, is defined as follows:

- If e1 and e2 are events then e1 and e2, e1 or e2 $\in$ CE.
- If exp1, exp2 $\in$ CE then exp1 and exp2, exp1 or exp2 in CE.

We plan to add more operators such as those for sequence and exclusive-or in the future. A correlation component has one input port for each possible input event to the correlation expression and an output port to publish the correlated event.

Given an application scenario with correlation, Cadena generates an XML file that contains the representation of each of the correlation expressions in the XML format. Figure 14 shows the XML file for the expression (e3 and e4) or (e5 and e6). At configuration time, the XML file is used to instantiate filters for each of the correlation expressions in the correlation and synchronization module (CSM) of AES. For each correlation subexpression, the filter_builder in the CSM creates a filter and registers itself as a consumer (internally in the event channel) for all events in the expression, and also acts as the publisher of the correlated event. Thus, when an event arrives at the event channel, it is first directed to the CSM if it is involved in a correlation expression (this information is derived from the synch_id tag in the XML file (see Figure 9)).

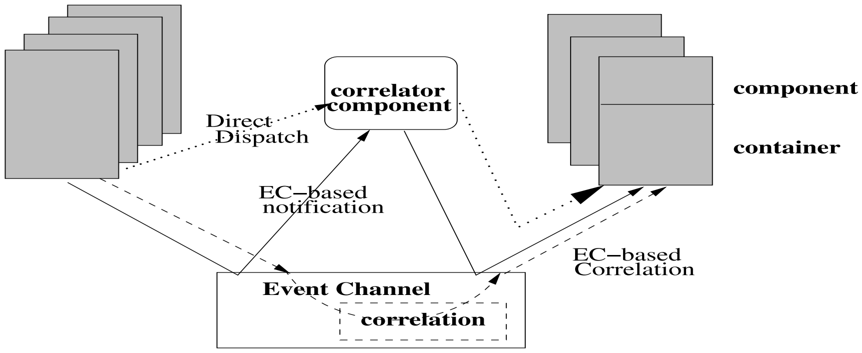

Figure 15: Event Correlation Experiments

The appropriate filters are then triggered in the CSM. If the event results in the satisfaction of a filter expression, the corresponding correlated event is generated and published. For example, for Figure 14, three filters are created, one for e3 and e4, one for e5 and e6, and an OR filter for the two intermediate correlated events generated by the first two filters (with ids 10 and 20). The scheme described above, event-channel based correlation (ECC), is the default scheme used in Cadena. The Boldstroke system also uses an event channel scheme to implement all correlation [Sha99].

We have also experimented with the following alternative schemes for correlation:
• *Correlator as a component (CCom)*: In this approach, as shown in Figure 15, each correlator is implemented as a CCM component. The correlator component subscribes to each of its input events. After the correlator receives notification for the events involved in its correlation expression, it emits the correlated events if the expression is satisfied.
• *Correlation in the container (CCon)*: In this approach, as shown in Figure 16, the correlation is done in the container at the consumer side. The container is initialized with the correlation expression from the XML file. The container subscribes to all events in the correlation expressions. As the notifications arrive, they are recorded in the container and a notification is generated to the component when the expression is satisfied.
The $CCom$ approach is currently supported in Cadena. For $CCon$, we have identified the code to the added to the container. For the purposes of experimentation, this code was inserted in the container manually. This however can be automated using code weaving techniques [KLM$^+$97].

The first experiment involves comparing the performance of the schemes ECC and $CCom$. The correlation expression is for the XML file shown in Figure 14 with all components on the same component server. Clearly, if the correlator component communicates with other components via the event channel (as shown by solid lines in Figure 15), then ECC will perform better than $CCom$ (as $CCom$ will incur the additional overhead of having the notifications traverse all the way up to the component whereas in ECC, they only need to travel up to the CSM module). One of the reasons a connection is tagged "AES" is if it is involved in a correlation. However, if the option of implementing correlation in a component is available, then we do not have this restriction anymore. This is an important consequence as DD notification is less expensive than AES notification. As mentioned earlier, the Boldstroke system

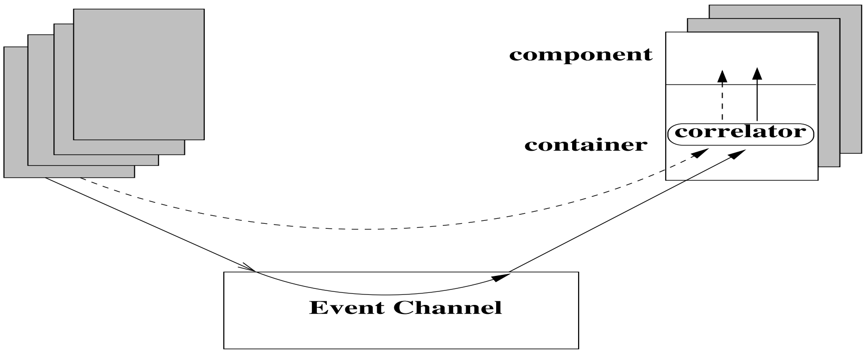

Figure 16: Event Correlation in the Container

Consumers	1	4	8	16
CCom with direct-dispatch	469	703	906	1546
ECC	515	874	1360	2707
CCon with direct-dispatch	313	610	1235	1984
CCon with EC	750	1562	2703	4906

Figure 17: Event Correlation within a component server

implements all correlation via the event channel and therefore, the $CCom$ scheme with DD opens the possibility of optimized correlation implementation. For example, consider the case where the correlated event is e1 and e2. In this case, $e1$ and $e2$ can be notified to the correlator component via direct dispatching, and the correlator component can also notify the correlated event via direct dispatching (rather than all notifications going via the event channel). Figure 17 gives the results of performance comparison of $CCom$ with DD (with event flow shown by dotted lines in Figure 15) and ECC (shown by dashed lines in Figure 15). As can be seen in Figure 17, $CCom$ with direct dispatching (first row) performs better than ECC (second row).

The second experiment involves comparing the performance for the case where the correlation is implemented in the container ($CCon$). The third row in Figure 17 gives the time taken for the case when event notification takes place via direct dispatching to the consumer component, and its container performs the correlation (dashed lines in Figure 16). For small number of consumers, the $CCon$ scheme with direct dispatching performs better than $CCom$ (since $CCom$ has the overhead of an additional component in the flow chain). However, as the number of consumers increase, in $CCon$, each consumer performs correlation independently for the same expression whereas in $CCom$, correlation is performed once in the correlator component. Hence, as the number of consumers increase, $CCom$ performs better. Thus, if a scenario involves a number of correlations with few consumers (1 or 2) of each correlated event then $CCon$ with direct-dispatching is a better alternative. For example, this scheme would perform better for the scenario given in Figure 1 as compared to $CCom$. As the number of consumers for a particular correlated event increase, $CCon$ is a better alternative. We also measured the performance for the case when the event notification is via the event channel rather than

DD (solid lines in Figure 16). In this case, the $CCon$ scheme incurs additional overhead as compared to the ECC scheme and hence, it is less efficient (fourth row in Figure 17).

6.1 Distributed Correlation

For scenarios involving multiple component servers, we have explored several alternative implementations for event correlation. Consider the case wherein the correlation expression is e1 AND e2, where e1 and e2 are produced by components in component server A and the consumer of the correlated event is on component server B. One possibility is to allow notifications for e1 and e2 to independently reach B and perform correlation at B. An alternative, more efficient way is to perform correlation at A itself (by placing the filter in the CSM of the event channel in A) and then send a single notification across the network to B. Thus, we need to determine the location of the correlation filters. For more complex expression such as the one in Figure 14, the placement of the filters for the intermediate correlated events also need to the decided. We have implemented an algorithm that optimizes the location of the correlation filters. The input to this program is the XML file with location for each source event specified (the "location" tag in Figure 14) and the topology of the underlying component servers (only two types of topologies, tree and complete graphs, are currently implemented). The algorithm, in turn, computes the location for all intermediate and final filters (the details of the algorithms have been omitted due to lack of space). The output is an XML file of type shown in Figure 14 annotated with the location of each filter. The configuration program in Cadena reads the XML file and instantiates the filters in the appropriate servers.

Figure 18 gives the performance comparison for different implementations of distributed correlation. The correlation expression is "(eA and eB) or (eC and eD)", where eA, eB, eC and eD are produced by components A, B, C and D respectively. Components A and B reside on component server 1, C and D on component server 2 and all consumers are on component server 3. The first row gives the timing figures for the case where correlation is implemented in a correlator component in component server 3 and notifications are delivered from A, B, C and D to the correlator component via direct dispatch. The second row is for the same configuration except that notifications are delivered via the event channel and correlation is done in the event channel. In this case, DD with correlator component scheme outperforms the second one as direct dispatching involves less overhead. The third row gives the figures when the filter placement optimization algorithm is applied to the event channel-based correlation. As compared to the second row, we see a significant drop in the time required. We believe for scenarios involving a large number of component servers, the filter placement algorithm will result is significant optimization and will outperform the direct dispatching scheme as well. However, further experimentation is required to validate this claim. The filter placement algorithm could also be applied to the correlator component scheme, but it would require significant rewiring of connections at the IDL3 level.

Consumers	1	4	8	16
C_ERM	2375	2454	2656	3204
E_ECC	6265	6656	7360	8375
D_ECC	3156	3781	4156	5500

Figure 18: Event Correlation with Distribution

7 Mode-aware Event Filtering

In Cadena, all event connections between components are configured at the initialization time. However, the flow of events over these connections may have to be constrained dynamically. For example, in Figure 1, the application operates in two modes: In the TacticalSteering mode, component TacticalSteering is enabled and NavSteering is disabled (and the opposite is true in mode NavSteering). Thus, in the TacticalSteering mode, NavSteering still receives all the notifications (as it is registered as a consumer of events irrespective of the mode), and these notifications are simply dropped. An alternative scheme is to make the event channel mode-aware and have the event channel constrain the flow of notifications based on modes. For example, in the TacticalSteering mode, the event channel will not deliver any notifications to NavSteering. AES is a mode-aware event service and is configured to implement mode-awareness using XML metadata generated from the mode-specifications of the component. We have shown that mode-aware AES adapt to the application modes, and improves the performance of the system by quenching the flow of unnecessary notifications. Due to lack of space, the details have been omitted.

8 Discussion

We have proposed an event communication framework that include a number of different alternative implementations for the different features of the event service middleware. We have evaluated the performance of these implementations and our study has shown that different implementations are suitable in different application contexts. This provides us with the flexibility of analyzing the application scenario to identify the most appropriate implementation for each of the event connections in the scenario. We have implemented some of the algorithms in Cadena. For example, programs to identify the ERM events, gateway generation for distribution, optimization of the filter placement, and different correlation implementations have been coded. These algorithms automatically identify suitable implementations for the different event connections, and generate the metadata needed to configure the containers to use the appropriate implementation for each of the event connections. Our future work will concentrate to further evaluation of the different event channel implementations in ECF for larger scenarios (specially, scenarios from the Boeing BoldStoke system), and enriching ECF with more schemes.

References

[CNF98] G. Cugola, E. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *IEEE International Conference on Software Engineering*, 1998.

[CRW99] A. Carzaniga, D. Rosenblum, and A. Wolf. Interfaces and algorithms for a wide-area event notification service. In *Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado*, 1999.

[GOA02] GOAL. The OpenCCM platform. *http://www.corbaweb.lifl.fr/OpenCCM*, 2002.

[Gro02] OMG CCM Implementators Group. Tutorial: Corba component model. In *OMG Meeting*, 2002.

[HBBM96] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using events to build large scale distributed applications. In *SIGOPS European Workshop*, 1996.

[HDD+03] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and R. Venkatesh. Cadena: An integrated development, analysis and verification environment for comba component systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2003.

[HLS97] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of OOPLSA*, 1997.

[KLM+97] G. Kiczales, J. Lampoing, A. Mendhekar, C. Maeda, C. Lopez, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241*, 1997.

[KR95] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21, 1995.

[MSS97] M. Mansouri-Samani and M. Sloman. Gem: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), 1997.

[Sch94] D. Schmidt. ACE: an object oriented framework for developing distributed applications. In *Proceedings of the USENIX C++ Technical Conference*, 1994.

[Sha99] D. Sharp. Avionics product line software architecture flow policies. In *Proceedings of the Digital Avionics Systems Conference*, 1999.

[WR02] N. Wang and C. Rodriques. Tutorial: Corba component model and QoS enabled CCM. In *PCES PI Meeting*, 2002.