Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Computer Standards & Interfaces 25 (2003) 345–355

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

A new development environment for an event-based distributed system

Tsun-Yu Hsiao[a,*], Nei-Chiung Perng[b], Winston Lo[c], Yue-Shan Chang[d], Shyan-Ming Yuan[a]

[a] *Department of Computer and Information Science, National Chiao-Tung University. No. 1001, Ta-Hsueh Road, Hsin-Chu, Taiwan, ROC*
[b] *Department of Computer Science and Information Engineering, National Taiwan University, No.1, Sec. 4, Roosevelt Road, Taipei, Taiwan, ROC*
[c] *Department of Computer Science and Information Engineering, Tung-Hai University, 181 Taichung-kang Rd., Sec. 3, Taichung, Taiwan, ROC*
[d] *Department of Electronic Engineering, Minghsin University of Science and Technology, Hsin-Chu, Taiwan, ROC*

Abstract

The rapid growth of data exchange on the Internet has created many critical problems that require an answer. Traditional data exchange systems based on client/server communication models are less scalable and incur especially high maintenance cost in the data exchange domain. For these reasons, many researchers have switched their interest to asynchronous communication models. Although Message-Oriented Middleware (MOM) is a middle-tier infrastructure that links operating systems and applications, such asynchronous communication APIs supported by middleware vendors are usually hard to use. For these reasons, in this study, we present a new development environment for asynchronous communication platforms which we term Ghostwriter. The keyword for our development environment is 'easy', that is, easy to use, easy to develop, and easy to deploy. Therefore, many researchers have switched their interest to asynchronous communication models. In addition, learning about and implementing the functions of the asynchronous communication's clients in Ghostwriter environment is simple. Other benefits are a lower technical learning curve, help for concentrate on system design, has easily reusable components, and easily integrated applications.

Keywords: Event-based; Middleware; Development environment

1. Introduction

Data exchange between applications in computer networks has become more and more popular, but such exchange on the rapidly expanding Internet has brought to light many unsolved issues. Generally, Internet data exchange systems are based on traditional client/server (peer-to-peer) architecture. Many models of Client/Server paradigm middleware, known as remote-invocation-based middleware that binds the complexities from the programming context, have recently become available. For example, we have Microsoft's DCOM/COM+ [16,22], OMG's (Object Management Group) CORBA (Common

* Corresponding author.
E-mail addresses: tyhsiao@cis.nctu.edu.tw (T.-Y. Hsiao), neil@rtlab.csie.ntu.edu.tw (N.-C. Perng), winston@mail.thu.edu.tw (W. Lo), ysc@must.edu.tw (Y.-S. Chang), smyuan@cis.nctu.edu.tw (S.-M. Yuan).

0920-5489/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0920-5489(03)00007-2

Object Request Broker Architecture) [12], and EJB (Enterprise Java Bean) [9,19,20], which are all based on client/service (request/response) architecture. However, these communication models are less scalable and incur especially high maintenance costs in the data exchange domain. For these reasons, many researchers in search of better Internet communications in this field have switched to asynchronous communication models (sometimes termed event-based or messaging-based communication models).

A communication infrastructure based on the asynchronous model has the advantage of being loosely coupled, many-to-many communication. Furthermore, such a model is usually considered more scalable than the traditional synchronous model. In general, an asynchronous communication model is part of a middleware set-up, which is the middle-tier infrastructure that links operating systems and applications [10]. The model also separates the underlying environment from applications and presents application programmers and users with a homogenous interface. The use of middleware in the development of large-scale systems is very common in enterprise data exchange systems [24]. Middleware that uses asynchronous communication as its underlying mechanism is termed Message Oriented Middleware (MOM). Many MOM-like platforms or standards are available, e.g., TIB/Rendezvous [23], ELVIN [5], Gryphon [7], SIENA [18], SOAP [17], the Java Message Service (JMS) [21], and the CORBA Event Service and Notification Service [13,14]. The JMS and the CORBA Event/Notification service, which are standard platforms in industry, have as their main objective an API that provides a reliable and flexible service for exchanging messages asynchronously between inter- or intra-enterprise applications.

However, many programmers choose not to use asynchronous communication models as data exchange platforms, for the simple reason that event-based programming is a difficult and complex work. The development of asynchronous data exchange programs requires a lot of domain knowledge about an event-based distributed system. Some research has revealed that protocols developed without programming models in mind lead to low-level service implementation that proves very cumbersome in use [6,8,15]. Remote-invocation-based middleware is intuitive but ties applications to rigid client/server communication. Moreover, standard messaging-based services, for example, the CORBA Event/Notification Service and the JMS, have different and complex APIs that programmers must get used to. For these reasons, such as in Ref. [2]; in this paper, we present a new development environment for an asynchronous communication platform, which we term Ghostwriter.

First, we extract the key ideas from various event-based middlewares. Then, we analysis the functions that the software development team will have to handle, including the system administration, the application programming, and the middleware configuration. We use this analysis of the functions to design the Ghostwriter architecture. Next, we define Event Markup Language (EventML) Document Type Definition (DTD) [25] files, which help the system administrator to define the roles of the applications that will be part of the Ghostwriter development process. Application programmers need only concern themselves with the data logic and how messages are sent and received in their programs. They need know nothing about event-based programming APIs or models. We also define the development flow for Ghostwriter and how to facilitate rapid deployment of applications onto the middleware.

The remainder of the paper is organized as follows: Section 2 discusses the design concepts of the Ghostwriter system. Section 3 presents the system architecture. In Section 4, we describe what has been done with Ghostwriter and discuss plans for future work on improving the Ghostwriter environment. Section 5 contains Conclusions.

2. The design concepts for the system

We entitle our system Ghostwriter. In ordinary usage, a ghostwriter is someone who is employed to write under someone else's name, that is, one who does the hard work for someone who does not wish to or cannot do it himself. Our Ghostwriter is thus a development environment that provides an easy means for system programmers who do not wish to or cannot create for themselves programs in an event-based distributed system.

2.1. The main concept of the Ghostwriter engine

As mentioned above, most Internet program communication still uses custom, socket-based, HTTP solutions. We believe that event-based distributed systems are more efficient data exchange models than traditional ones. However, many programmers choose not to use event-based programming models as the underlying mechanism because of the difficulties they present, in particular, because of the amount of knowledge about the domain that they demand from the programmer wishing to develop data exchange programs.

The keyword to describe the Ghostwriter development environment is 'easy', i.e., easy to use, easy to develop, easy to deploy. It is also easy to learn, to explore, and to implement the functions of Ghostwriter clients. There are several additional benefits with the infrastructure: a lower technical learning curve; help with concentrating on the system design; easily reusable components; easy integration with application. We describe these benefits below, respectively.

2.1.1. Lowering the technical learning curve

In the Ghostwriter development environment, programmers program data exchange logic just like writing a single function call. They need only know the type of incoming parameters to handle and return the data in a predefined format. Thus, they handle the data logic and leave the rest to Ghostwriter. What the system does is to provide a shell, which is a class containing the abstract methods, such as send and receive. A client application inherits this class and implements the abstract methods. The communication issues are already written into the shell. The application programmers are not even aware of the existence of the middleware.

2.1.2. Help with concentrating on the system design

Ghostwriter handles all the network communications for distributed computing, including socket and port management, protocols, semantics, and message transportation. In other words, application developers do not need to concern their client applications with the communication infrastructure. The same is true for system analysts, who need only to learn a little event-based communication knowledge. As a result, they do not need to write complicated system specifications. This simplicity helps the application developer to write clients. The role of programmers and system analysts is discussed in Section 2.2.

2.1.3. A sophisticated system of cooperation

Although in team work it is always import ant for individuals to do their separate task well, it is equally important for them to work well in cooperation. The best way is to assign tasks clearly. For this reason, Ghostwriter separates the whole system development into three parts: system administration, application programming, and communication configuration. In huge software development, cooperation is the best way to decrease working time. Ghostwriter is capable of sophisticated cooperation. It assigns the work into five roles: end users, application engineers, system analysts, middleware providers, and Ghostwriter providers. We describe these five roles in Section 2.2.

2.1.4. Easily reusable components

When customer requirements change, they need to change the underlying middleware. For example, an enterprise may wish to switch from the JMS to the CORBA Event Service. In that case, the action of updating the Ghostwriter program should not involve rewriting the whole system and applications, which would be unreasonable. In our system, all the developers need to do is to choose another Ghostwriter engine suitable to the new middleware. Furthermore, the Ghostwriter client applications need no rewriting. It is said that distributed systems are capable of location transparency, but Ghostwriter is even more capable of middleware transparency. No matter what middleware enterprise is chosen, Ghostwriter treats it in the same manner.

2.1.5. Easily integrated applications

Another important goal of using middleware is enterprise application integration (EAI). In an international enterprise, since many legacy applications work perfectly well everyday, it is not reasonable to rewrite them for a new task that is only slightly different. The features in our middleware mean that we need only add a wrapper to an application to make it operable in a new system. Ghostwriter is a client of middleware, so it has all the capabilities that middleware owned. Developers can integrate the application in two ways;

either integrate it directly into the underlying middleware or wrap it in a Ghostwriter client.

2.2. Separate the different tasks

Building an event-based system is a large undertaking with many problems to be solved. We believe that there are considerations: system administration, application programming, and communication configuration.

2.2.1. System administration

Usually, once the system requirements are clarified, the next step is to specify suitable system architecture. For instance, decisions must be made on how many event channels (or the JMS topic) are to be used and how many clients are programmed. At this stage, system analysts configure the settings into an EventML file, which we now describe in more details.

There may be senior engineers who prefer to analyze the system deployment rather than do the trivial coding work. There may be system analysts who only concern themselves with business logic. In addition, there may be employees in the marking department that may know about customer/enterprise requirements, but do not have the talent to write client applications. Although such people can do nothing in terms of traditional software development, with Ghostwriter they can write simple EventML files to accomplish the system administration.

2.2.2. Application programming

Data logic, which is the most important part in the client application, is written by application engineers. We consider the engineers, who write the codes, to handle what messages are received and what are messages sent. Complicated network programming, such as sockets, middleware, and event-based distributed system should be transparent from engineers. Moreover, engineers do not need to know the detail of the communication technology, that is, whether it is the CORBA event/notification service or the JMS. Data logic is the only thing needed at this application-programming phase.

End user is the top role in Ghostwriter system. An end user is the one who uses the application above the communication system. For instance, nurses and doctors are the end users of a health care monitoring system and the investors are the end users of a financial analysis system.

2.2.3. Communication configuration

The system administration uses business logic to configure a system, and the application programming uses data logic to program a client application. Those two phases are both separate from communications, so we must have some component to handle the communication configuration. Those who are familiar with network environment configure the physical communication. As already stated, our research presents an infrastructure, which is a development environment for event-based distributed system. Anyone who understands the concept of Ghostwriter system can implement his own Ghostwriter engine.

In our research, we use Middleware products from providers such as JavaORB and SonicMQ. There are many other products that use the CORBA Event/Notification and Java Message Service standard. The Ghostwriter provides a suitable engine for enterprise use, such as the CORBA version Ghostwriter engine or the JMS version (Fig. 1).

2.3. XML and CORBA mapping

This section describes our XML definition, which corresponds to the general characteristics of event-

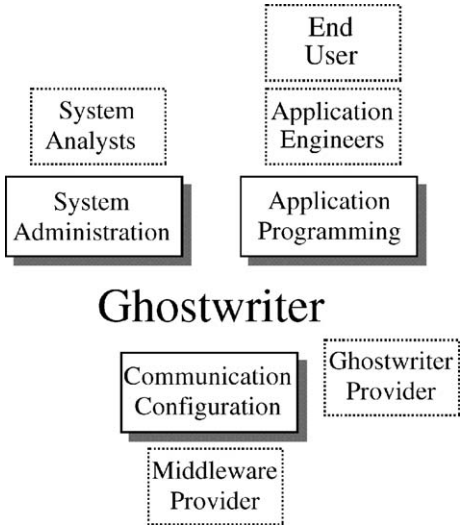

Fig. 1. Roles and tasks in the Ghostwriter development environment.

based middleware. First, we extract the key ideas from various event-based middlewares. Then, we define the DTD of an EventML file that features the general characteristic of event-based middleware. In the EventML DTD, we give an example definition of a client subscribing or publishing to a channel. The system administrator can use an EventML to describe the application's role in the data exchange environment, and the ghostwriter application can join for data exchange.

2.3.1. Extract the key ideas from event-based middleware

Nowadays, many event-based models have been defined, but all of them share the same central ideas. There is absolutely nothing new in the technologies pertaining to the CORBA Event/Notification Service, the Java Message Service, or anything similar. They are simply slightly different variations of applications that we are familiar with.

For this reasons, we did not design EventML to exactly match all the middlewares. Instead, we extract the key ideas from those event-based distributed systems. We believe that there are four key ideas in the event-based model: channels, clients, the relationships between channels and clients, and communication models.

Although there is a comfortable overlap between the JMS and the CORBA Notification Service communication models and their capabilities, there are several differences (Table 1).

We select CORBA and Java solutions for comparison because the CORBA Event/Notification Server and Java Message Service are well-known standard models of asynchronous communication. The Notification Service is a new, improved version of the Event Service, but both are still very similar.

Table 1
Characteristics of the CORBA event/notification service and the JMS

	CORBA ES	CORBA NS	JMS
Channel	Channels	Channel Factory	Topics
Communication	Pull/push model	Pull/push model	Push model
Data type	Any and typed Event	Structured event	Stream, map, text, object, Bytes Message
Filtering	No	Yes	Yes

```
<!ELEMENT EventML (EventManager, Clients)>
<!ELEMENT EventManager (Channel*, Middleware)>
<!ELEMENT Channel EMPTY>
<!ATTLIST Channel
      Name CDATA #REQUIRED>
<!ELEMENT Middleware (CORBA, JMS)>
<!ELEMENT CORBA EMPTY>
<!ELEMENT JMS EMPTY>
<!ELEMENT Clients (Supplier*, Consumer*)>
<!ELEMENT Supplier (Publish*)>
<!ATTLIST Supplier
      ClientClass CDATA #REQUIRED>
<!ELEMENT Publish EMPTY>
<!ATTLIST Publish
      ChannelName CDATA #REQUIRED>
<!ELEMENT Consumer (Subscribe*)>
<!ATTLIST Consumer
      ClientClass CDATA #REQUIRED>
<!ELEMENT Subscribe EMPTY>
<!ATTLIST Subscribe
      ChannelName CDATA #REQUIRED>
```

Fig. 2. EventML DTD.

Comparing the Notification Service and the JMS, the first has a pull/push model of communication–the second is only a push model; the first has a publish/subscribe model–the second is both a publish/subscribe model and a point-to-point model. Last, they both have channel management object. But in event service, there is no channel management object.

The Notification Service supports a message, which is a structured event that is defined in IDL. The JMS supports six different message formats with different message bodies. Both services have filtering mechanisms.

2.3.2. Details of the EventML

After extracting the key ideas and comparing the difference, we designed the EventML (Fig. 2).

The < EventML>tag is the root which has two parts: < EventManager>and < Clients>. The first of these tags manages the main setting in the event manager, such as channels and middleware-specified properties, while the second manages the clients. We define the suppliers and consumers in EventML file, and indicate the relationship between clients and channels.

```
<?xml version='1.0' standalone="no"?>
<!DOCTYPE EventML SYSTEM "EventML.dtd">
<EventML>
    <EventManager>
        <Channel Name="NCTU"/>
        <Channel Name="DCS Lab"/>
    </EventManager>
    <Clients>
        <Supplier ClientClass="News">
            <Publish ChannelName="NCTU"/>
            <Publish ChannelName="DCS Lab"/>
        </Supplier>
        <Consumer ClientClass="Neil">
            <Subscribe ChannelName="DCS Lab"/>
        </Consumer>
    </Clients>
</EventML>
```

Fig. 3. EventML sample.

2.3.3. An EventML sample

The following is a small sample of EventML (Fig. 3):

The description in EventML asks the event manager to add the channels, NCTU and DCS Lab. Then, this configuration file assigns one supplier News and one consumer Neil. The supplier News publishes the events to the NCTU and DCS Lab channels. The consumer Neil subscribes only to the DCS Lab channel, so it will receive only the events on the DCS Lab channel.

3. Implementing the Ghostwriter system

The architecture of Ghostwriter is described in this section. We give concrete form to our ideas in the form of the simple Ghostwriter engine we present with a description of implementation.

3.1. System architecture and sample

Fig. 4 shows an overview of Ghostwriter. Each of the hosts in its system follows this architecture.

The communication components are as follows. At the bottom of the figure comes the middleware, which handles communication between all the hosts, i.e., it receives/sends events or messages from all suppliers/consumers. It can be any one of CORBA with Event/Notification, the JMS, or any non-standard Message-Oriented Middleware. It receives events and passes them to the Ghostwriter engine. On the other hand, the engine sends events to the middleware, which then delivers them through middleware's channels.

At the top, there are client applications. These are wrapped as Ghostwriter clients for separating the tasks. The clients communicate with the Ghostwriter engine, i.e., they receive events or messages from and send them to the engine. This interaction between engine and clients is predefined in the Ghostwriter client class, which allows application engineers to program clients and concentrate on data logic only.

Configuration files describe the system setting. The system configuration is written in an EventML files. The system analysts write their semantics in these files, and then register them in the Ghostwriter engine. There is a mapping table in the engine that retains, after registration, all the relationships between the clients and channels written in the files. With the mapping table, the Ghostwriter knows which event to send to which client and vice versa.

3.2. The Ghostwriter development flow

There are several steps to initializing Ghostwriter communication. We begin by configuring the underlying middleware. At this point, the appropriate engine for the system is specified by the provider (Fig. 5).

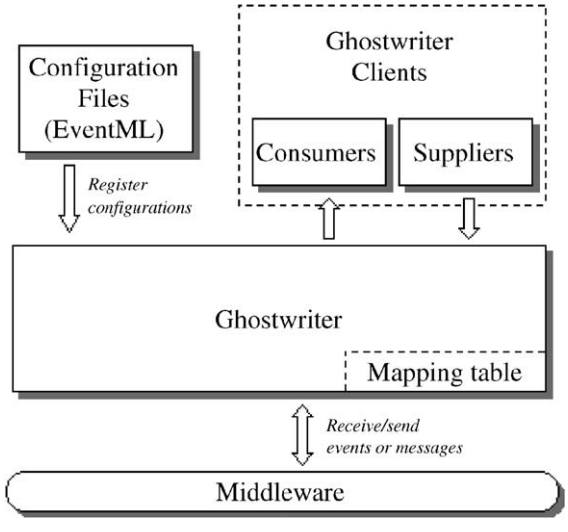

Fig. 4. Ghostwriter architecture.

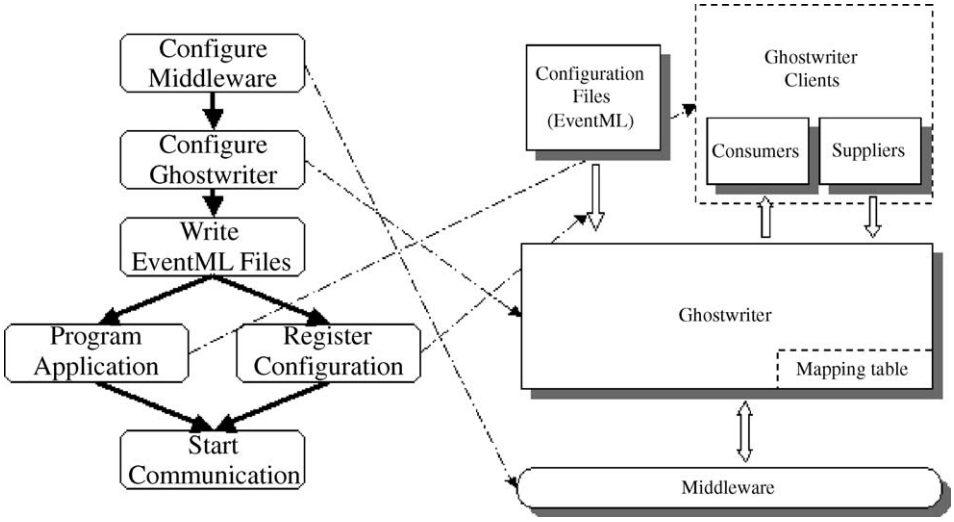

Fig. 5. The development flow of the Ghostwriter environment.

Second, the engine is configured and linked to the middleware. Of course, a CORBA version Ghostwriter engine is provided for the CORBA Event Service and a JMS version for the Java Message Service.

The third step involves writing the configuration in the EventML. System analysts set up the channels, suppliers, consumers, relationships, and specify the system settings they want. That step is followed by two more important tasks. First, the application engineers program the client applications, ensuring that suppliers and consumers are included as Ghostwriter clients. Second, the configuration files are registered in the Ghostwriter engine and a mapping table is created. At this point, we are ready to exploit the communication functions of Ghostwriter use through event-based distributed systems.

3.3. The platform for Ghostwriter implementation

The Ghostwriter engine is implemented on the Microsoft Windows 2000 Advanced Server version and the Java Development Kit (JDK) for win32 version 1.3.0_02. The CORBA Notification Service, JavaORB [4] v2.2.7 product from the Distributed Object Group (DOG) is the chosen product. It is a full implementation of the CORBA 2.3 and provides all the features specified by the OMG and others such as Naming, Event, Transaction, Property, and Collection. A new complete version of the JavaORB, OpenORB 1.0 [3,11], is already available. The XML parser, Xerces 1.3.1, is used to parse and validate the EventML configuration files. The Xerces Java Parser 1.3.1 supports the XML 1.0 [25] recommendation and contains advanced parser functionality, such as the XML Schema, DOM Level 2 version 1.0, and the SAX version 2, in addition to supporting the industry-standard DOM Level 1 and the SAX version 1 APIs.

4. Discussion and future work

In this section, we describe what has been done with Ghostwriter and what our plans for future steps with the environment are.

4.1. What has been done with Ghostwriter

Standard JavaORB and SonicMQ programming are two samples that perform many trivial tasks. Both CORBA Event/Notification Service and Java Message Service have several preparations. What we have done with Ghostwriter is to extract those tasks from the two programs and incorporate them into the Ghostwriter engine configured by EventML files.

To program a CORBA application, first, we need to resolve the initial reference of the ORB and the

object adapter, either BOA or POA. Then, we activate the object adapter, resolve the Event Service or Notification Service, bind the channel, ask for proxies, and finally, initiate communications. The line number of the code is about 50. Also, the CORBA programmers need to understand the event-based programming concept before they can write the CORBA event-based programs. (Please see the CORBA specifications [12,13,14] for more detail).

The left side of Fig. 6 shows the programming model of the CORBA. The gray dialog there is a partial code segment, which applies only to binding the JavaORB and event service. In a complete event-based program, some other segment or segments will be needed, such as for sending an event to the Event Channel. The right side of the figure shows the programming style of the Ghostwriter client. The gray dialog there is the code segment for send/receive communications between the Ghostwriter Engine and any Message-Oriented Middleware. Any complicated codes, such as bind to CORBA ORB, locate event channel, bind event channel, send event object … are omitted in Ghostwriter client. Ghostwriter clients need only to bind to the Ghostwriter Engine, and simply send or receive any event defined in business logic. The engine handles all the dirty work and the rule of communication is to obey the EventML already registered in it.

Similarly, to program a JMS application, we must first check the connection factories. With this factory, no matter whether it is a queue connection factory or a topic connection factory, we can create a connection. Next, we create a session with a connection, and then a publisher or a subscriber with the session. Finally, we publish or subscribe messages to a queue or to a topic. In other words, JMS programmers need to understand the JMS programming concept. (Please see the JMS specifications [21] for more detail).

Using Ghostwriter as the middle-tier between the application and the middleware, we can omit the complicated codes for initializing, middleware/locate channel/bind channel, etc. Programmers need only implement the receive method in their Ghostwriter client and the send method when they have messages for sending to the middleware. Only system analysts (those who specify the EventML files) need to understand the event-based programming concept.

We establish that the Ghostwriter infrastructure works in the Highly Confidential Information System* (HiCIS). The purpose of this HiCIS project is to research and implement an information system that is robust and secure enough to survive information war-

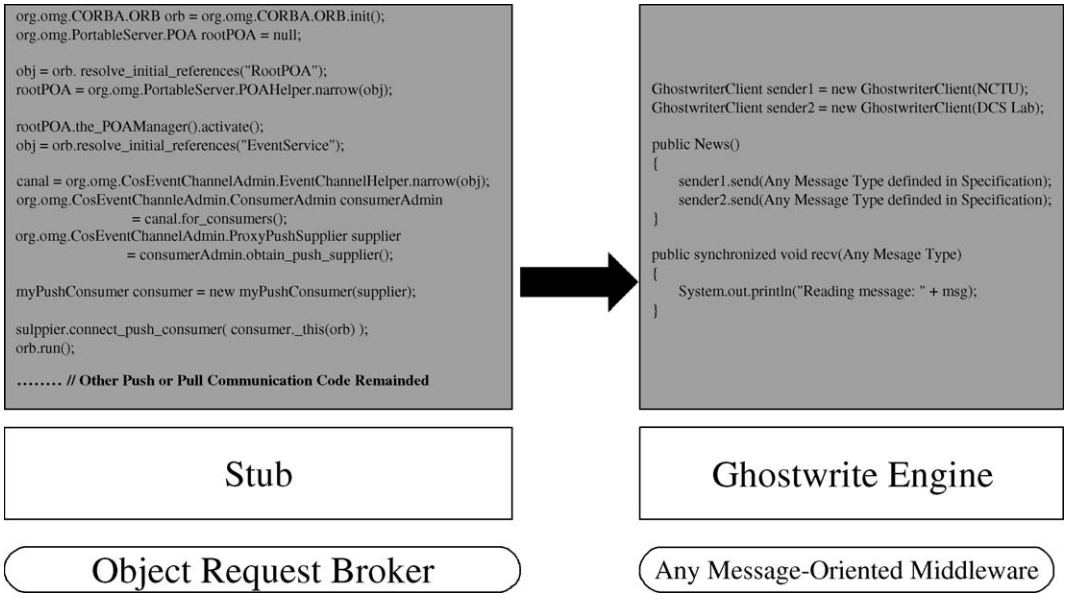

Fig. 6. Comparing the traditional CORBA event-based application and the Ghostwriter Client.

fare. The project has five teams, each headed by a core technology specialist in the field of information system protection. The core areas are information concealment, encryption, monitoring, adaptation and reconfiguration. Although the team leaders are experts in their individual core areas, some are not familiar with network programming. The project is collaborative and the programs are integrative, so some kind of integrative methodology and tool that will allow other users to exploit the event-based data exchange platform are needed. Our choice for that task is the Ghostwriter Infrastructure, which, because of its message-oriented middleware, provides a data exchange platform for data transfer, e.g., encrypted hidden image.

The concept and role of Ghostwriter in the project are to facilitate collaboration and integration and thus reduce the difficulty of building an application in an event-based distributed system. Programmers need only be concerned with their research domain program. When there is data to be transferred through our data exchange platform, programmers need simply to use the send/receive method. They have no need of any domain knowledge of the event-based system. Table 2 shows a comparison of the CORBA event/notification service, the JMS 1.1 and our Ghostwriter programming models. As we can see, the Ghostwriter client programmer has no need to know how the event-based programming model functions. They can simply connect with Ghostwriter and send or receive the events required.

4.2. Future work

Having established that the Ghostwriter infrastructure works in HiCIS and makes building applications for event-based distributed systems simpler, the next consideration is commercialization. This does not mean that we ourselves have plans to produce and market Ghostwriter, but we are convinced that Ghostwriter has commercial possibilities.

We intend to add an event filtering mechanism to EventML. Although the CORBA Event Service does not have this function, both the Notification Service and the Java Message Service feature event filtering and message selection. Furthermore, because of the information explosion on the Internet, such a mechanism will constitute an important part of middleware.

A further development must be to improve EventML until it becomes a generic solution. The CORBA Event/Notification and the Java Message Service used in this study are but two examples of event-based systems. We are aware that Middleware

Table 2
Comparison of middleware programming models

	CORBA ES/NS	JMS 1.1	Ghostwriter
Event-based Domain Knowledge Needed	YES	YES	NO (Just System Analyst who writes EventML need to know about this)
Steps that Push Supplier need to do	(1) Bind to the EventChannel	(1) Lookup ConnectionFactory from JNDI	(1) Bind to Ghostwriter
	(2) Get a SupplierAdmin	(2) Create Connection from ConnectionFactory	(2) Implements Send()
	(3) Get a consumer proxy	(3) Create Session from Connection	
	(4) Add the supplier to the EventChannel	(4) Create MessageProducer from Session	
	(5) Data transfer	(5) Data transfer	
Steps that Push Consumer need to do	(1) Bind to the EventChannel	(1) Lookup ConnectionFactory from JNDI	(1) Bind to Ghostwriter
	(2) Get a ConsumerAdmin	(2) Create Connection from ConnectionFactory	(2) Implements Receive()
	(3) Get a supplier proxy	(3) Create Session from Connection	
	(4) Add the consumer to the EventChannel	(4) Create MessageConsumer from Session	
	(5) Implements Push()	(5) Implements setMessageListener()	

exists in the many forms devised by others elsewhere, so if the Ghostwriter development environment is to be a general solution, it must, as we intend, function with all possible forms.

The sheer volume of messages between the middleware and the clients that pass through Ghostwriter creates a data exchange bottleneck in the engine. According to our measurements, the data transfer time through the engine costs about 150 ms (In a Pentium2-450 PC with 256MB RAM). Because of this bottleneck [1], we propose the practical ghostwriter model shown in Fig. 7.

This model accepts configuration in the files, still written in EventML, just like the normal model does. However, it does not serve as a bridge between the clients and the middleware. Instead, it generates a wrapper class for clients. This wrapper class works as in a normal engine, but creates only one working thread for client, which is to say that one wrapper class works for one client only.

This model is the object of our future research. The reason for using the normal model was to establish how easily the Ghostwriter development environment did the work and how simple the model was to use. However, since it suffers from bottleneck, it is not efficient enough for our requirements.

The performance should be the same for native middleware clients because they are no different from the practical Ghostwriter clients.

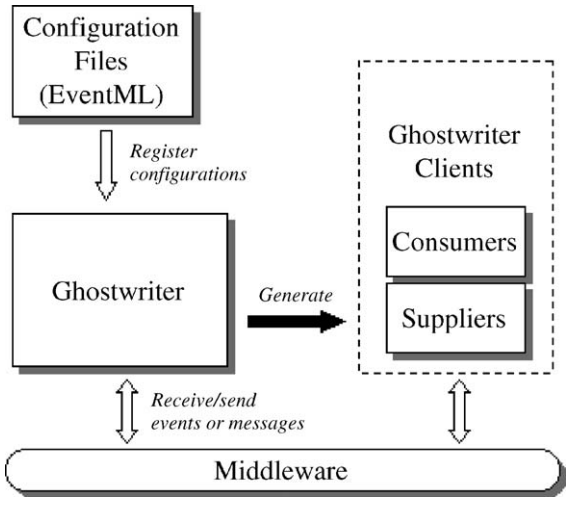

Fig. 7. Practical Ghostwriter architecture.

5. Conclusions

The rapid growth of data exchange on the Internet has created many critical problems that require an answer. Traditional data exchange systems based on client/server communication models are less scalable and incur especially high-maintenance cost in the data exchange domain. For these reasons, many researchers have switched their interest to asynchronous communication models. Although Message-Oriented Middleware (MOM) is a middle-tier infrastructure that links operating systems and applications, such asynchronous communication APIs supported by middleware vendors are usually hard to use. For these reasons, we have presented a new development environment for asynchronous communication platforms which we term Ghostwriter. The keyword for this development environment is ‘easy’, that is, easy to use, easy to develop, and easy to deploy. In addition, learning about and implementing the functions of the asynchronous communication’s clients in Ghostwriter environment are simple. Other benefits are a lower technical learning curve, help for concentrate on system design, easily reusable components, and easily integrated applications.

A further key design point about the Ghostwriter environment is the separation of tasks and roles, i.e., between the different tasks of system administration, application programming, and communication configuration; and of the different development roles of end users, application engineers, system analysts, middleware providers, and Ghostwriter providers. But, linking them all, Ghostwriter provides engineers with a sophisticated system of cooperation. We have established that the Ghostwriter infrastructure works in a Highly Confidential Information System and reduces the difficulties in building applications for event-based distributed systems.

Acknowledgements

The authors thank anonymous reviewers for many useful comments. This work was partially supported by Ministry of Education of the Republic of China under Grant No. 89-E-FA04-1-4, High Confidence Information Systems.

References

[1] A. Arulanthu, C. O'Ryan, D. Schmidt, et al., The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging, Proc. Of IFIP/ACM Int'l Conf. on Distributed Systems Platforms and Open Distributed Processing, ACM, 2000, pp. 208–230.

[2] J. Bacon, K. Moody, J. Bates, Ma. Chaoying, A. McNeil, O. Seidel, M. Spiteri, Generic support for distributed applications, Computer 33 (3) (2000 March) 68–76.

[3] J. Daniel, C. Wood, OpenORB Programmers Guide, ExoLab. Group Org. (2000 November 6).

[4] Distributed Object Group, JavaORB Event Service, 1999. http://dog.team.free.fr/details_javaorb.html.

[5] Elvin, http://elvin.dstc.edu.au/.

[6] P.T. Eugster, R. Guerraoui, J. Sventek, Distributed asynchronous collections: Abstractions for publish/subscribe interaction, Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP-00), Cannes, France, Springer Verlag, 2000 (June), pp. 252–276.

[7] Gryphon, http://www.research.ibm.com/gryphon/.

[8] R. Guerraoui, What object-oriented distributed programming does not have to be, and what it may be, Informatik 2 (1999 April) 3–8.

[9] R. Monson-Haefel, Enterprise Java Beans, 2nd ed., O'reilly, 2000 (March).

[10] D. Milojicic, Middleware's role, today and tomorrow, IEEE Concurr. 7 (2) (1999 April–June) 70–80.

[11] O. Modica, OpenORB Event Service, Exolab. Org. (2000 September 9). http://openorb.exoLab.org.

[12] Object Management Group, CORBA: Common Object Request Broker Architecture and Specification, Revision 2.4 (2000 October). http://www.corba.org.

[13] Object Management Group, Event Service Specification, V 1.1, http://www.corba.org/.

[14] Object Management Group, Notification Service Specification, V 1.0, http://www.corba.org/.

[15] P.Th. Eugster, R. Boichat, R. Guerraoui, J. Sventek, Effective multicast programming in large scale distributed systems, in: Concurrency and Computation: Practice and Experience, vol. 13, Issue 6, Wiley & Sons, 2001 (May) pp. 421–447.

[16] A. Rofail, Y. Shohoud, Mastering COM and COM+, Sybex, 1999.

[17] SOAP, http://www.w3.org/TR/SOAP/.

[18] SIENA, http://www.research.ibm.com/gryphon/.

[19] Sun Microsystems, Enterprise JavaBeans™ Specification Version 2 (2000 October). http://java.sun.com/products/ejb.

[20] Sun Microsystems, Java 2 Platform, Enterprise Edition (J2EE), http://java.sun.com/j2ee/.

[21] Sun Microsystems, Java Message Service, Version 1.0.2 (1999 November). http://java.sun.com/products/jms.

[22] T.L. Thai, Learning DCOM, O'reilly, 1999.

[23] TIB/Rendezvous White Paper (1999), http://www.rv.tibco.com/whitepaper.html.

[24] S. Vinoski, Where is middleware? IEEE Internet Computing 6 (2) (2002 Mar./Apr.) 83–85.

[25] World Wide Web Consortium (W3C), Extensible Markup Language (XML), http://www.w3.org/XML/.