The Design of Real-time Message Middleware Based on Event Service

Pengjie Wang[1], Houjie Li[2*], Haiyu Song[1], Wei Li[1] and Chengxue Yu[3]

[1]*College of Computer Science & Engineering, Dalian Nationalities University, Dalian, 116600, China*

[2]*College of Information & Communication Engineering, Dalian Nationalities University, Dalian, 116600, China*

[3]*College of International Business, Dalian Nationalities University, Dalian, 116600, China*

Abstract: The Event Service defined in Common Object Request Broker Architecture provides an asynchronous, multicast communication model among distributed objects. However, the standard CORBA Event Service and previous methods lack important features required by real-time applications. For instance, message transferring programs for cooperating design groups may have requirements of real-time processing and persistent storage of Event data. To address these problems, we propose a real-time message middleware design based on Event service. First, we extend Standard Event Service to a real-time Message Middleware by improving the QoS (Quality of Service) of Event Channel. Second, we propose a model that can persistently store the Event data and recover after the system crushed. Finally, we introduce an Event Channel Manager object to well manage the Event Channels. By using this real-time message middleware, the Object Request Brokers can communicate stably with each other without caring whether the two or many communication sides have relation or whether the other communication side is ready.

Keywords: Distributed computing, message middleware, Event Service, asynchronous multicast communication, real-time system.

1. INTRODUCTION

In typical CORBA (Common Object Request Broker Architecture) based applications, most of the communications between client and server are one-way and synchronous [1]. This way is the default communication way between distributed objects, and it is widely used in lots of engineering scenarios. However, there are also scenarios, where a loosely-coupled communication way is demanded. For example, in a designing project, which may last long period of time and need elaborate cooperation between designers, there are often setup of new designing groups and teardown of an old designing groups. At the same time, the members of group are also often changed. If each member of the group communicate with any other member one-way and synchronously, the tight coupling communication program will be too complex, and this will affect the scalability of the system. A new communication way, which is asynchronous and multicast, is demanded. In CORBA, the Event Service [2] and Notification Service [3-5] can offer this kind of asynchronous and multicast communication.

From the specification [3] and some implementation of the Notification Service [4, 5], we can see that the Notification Service is too complex and bring much overhead. On the other hand, the Event Service is more simple and practical, and also has lots of engineering applications [6-8].

Event Service adopts the policy of "Trying the best" and it has lots of disadvantage as a message middleware. Thus it needs to be improved greatly, before it can be applied in real-time system. There are several extensions and implementations for Event Service [9-11]. Chen *et al.* [9] improved the Event Service by introducing filtering mechanism. This filtering mechanism can reduce unnecessary Event message and can reduce the number of Events sent to the consumers. Chen *et al.* also proposed to build Event repository structure in order to increase additional Event types and information about properties. Zhang *et al.* [10] address fault-tolerance issues of the CORBA Event Service. They proposed to use object replication technology to implement fault-tolerance CORBA Event Service. They also applied their fault-tolerance Event Service to one distributed power monitor system, and proved that their method is feasible. Gong *et al.* [11] proposed an asynchronous callback model for Event Service of CORBA. The key of this model was to separate the cycle of requesting and responding, in order to form two separate invoking routes. Through this model, the requests message and responses message were non-interference from each other. User can query response using IIOP Client at any time after sending a request. Thus they achieved asynchronous transmission of information.

However, the QoS (Quality of Service), real-time issue and persistent storage have not been the focus in these designs and implementations. In this paper, based on our previous work [12], we propose a message middleware design based on Event Service. In this message middleware, we introduce the persistent storage mechanism and improve the

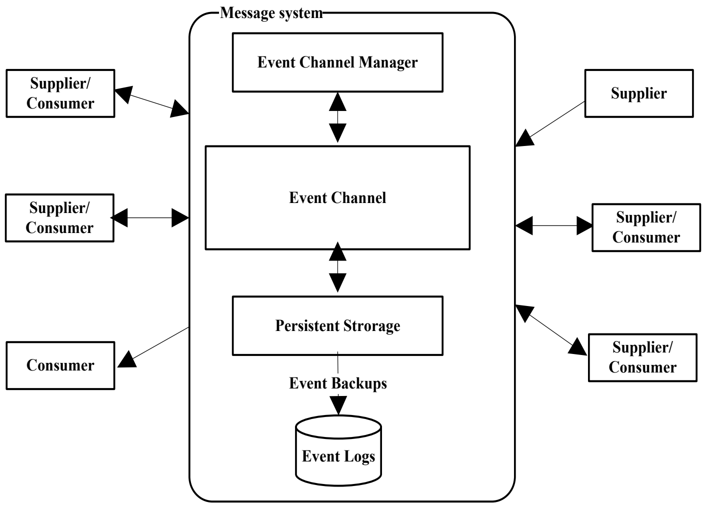

Fig. (1). The overall structure of message middleware system.

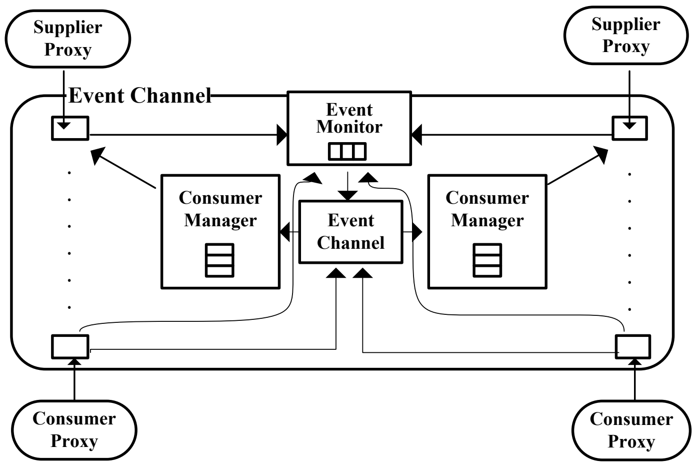

Fig. (2). The structure of Event Channel.

Quality of Service for the real-time Event message transferring.

Fig. (**1**) shows the overall structure of our message middleware system. Event Channel Manager is responsible for managing Event Channels in the message system, as Fig. (**2**) shows. Please refer Section 2 for the details. Event Channel is responsible for transferring Events data from the Supplier to the Consumer, as Fig. (**2**) shows. Please refer Section 3 for the details. The unit of Persistent Storage is introduced here to permanently store the Events data in the Event Channel. It can start the recovery process after the Event Channel fails unexpectedly, as Fig. (**3**) shows. Please refer Section 4 for the details. Section 5 present a application case and Section 6 give the conclusions.

2. EVENT CHANNEL MANAGER

The extended Event Channel Manager is response for the creating, destroying, suspending, transferring and searching for the Event Channel. Any design group can request to create a new Event Channel, and the one who start a new Event Channel becomes the owner of this Event Channel. Other designers can search and browser all the Event Channels and they can also request to join any Event Channel which they are interested in. If the project is suspended, the owner of the Event Channel can request to suspend the Event Channel temporarily. After the project is finished, the group will be dismissed, and the owner can request Event Channel Manager to destroy the channel. When the Channel Manager received the request, it will check the status of the Event Channel and approve destroying the channel after it make sure that there is no transferring Event data. Normally, the owner cannot logout from the Event Channel until the Channel is destroyed. However, if the owner does want to logout from the Channel, he has to request transferring the ownership of the channel to other member of this channel. After the Event Channel Manager receives the request, it will randomly pick one member as the owner of the Channel.

Event Channel Manager also has the responsibility of destroying the Orphan Channel, which is defined as a Channel that has no communication ends attached. The reason that leads to the orphan channel is that the owner of the Event Channel unexpectedly fails and aborts from the system before it has enough time to destroy the Channel.

All the activities of user will be written into the logs according to data structure defined in Section 4.2. Based on these backups, we can restart and rebuild the Event Channel, even after the breakdown of message middleware system. Section 4 gives the details.

In order to meet the requirement of our message middleware system, we need to extend the Event Service specification by adding the IDL (Interface Description Language) of Event Channel Manager, COSChannelManager, as following.

```
module COSChannelManager
{
interface EventChannelManager
{
CosEventChannelAdmin::EventChannel CreateChannel
(in string ChannelName )
Raises   (NameAlreadyUsed);
CosEventChannelAdmin::EventChannel SearchChannel
 (in string ChannelNname)
Raises   (EventChannelNotFound);
CosEventChannelAdmin::EventChannel RestartChannel
 (in string ChannelName)
Raises   (EventChannelNotFound);
 CosEventChannelAdmin::EventChannel  DestroyChannel
  (in string ChannelName)
 Raises   (EventChannelNotFound);
 CosEventChannelAdmin::EventChannel  SuspendChannel
  (in string ChannelName)
 Raises   (EventChannelNotFound);
 CosEventChannelAdmin::EventChannel  TransferChannel
  (in string ChannelName)
 Raise   (EventChannelNotFound);
exception NameAlreadyUsed { };
exception EventChannelNotFound{ };
};
};
```

3. EVENT CHANNEL

The Event Channel is shown as Fig. (**2**).

3.1. Two-way Communication

When one designer requests to join a particular existing Event Channel, he/she can choose the access type between read-only and two-way (read and write). This leads to three types of user in our message middleware system. One is "Supplier/Consumer", which is not only the Event supplier but also the Event consumer, and its connection to the middleware system is two-way. Another two types is the "Supplier" which only sends the Event data to the middleware system, and "Consumer" which only receives the Event data from the middleware system. Fig. (**2**) shows the two-way communication scenario.

Two-way communication will make the race condition in Event Channel more serious. This happens when more than one Supplier send Event data in the same Event Channel. There are two solutions to address this problem. Solution one is that each supplier builds his own Event Array in the Event Channel in order to store the Event data produced by one specific Supplier. The advantage of this solution is that it can avoid the race condition and lock mechanisms completely. However, we have to build a new Event Array every time a new Supplier is added. To maintain these arrays will be a problem, especially when there are too many designers in a cooperating design group.

Solution two is to keep only one Event Array for all Suppliers. In this situation, we have to deal with the concurrency problem. The request of the Supplier may be suspended for waiting if the lock mechanism is introduced to our middleware system. To avoid this, more temporal memory need to be allocated for the Supplier Proxies. This also leads to memory overhead.

Based on above discussion, we choose Solution one. As Fig. (**2**) shows, the Event data is put into Consumer Manager. In Fig. (**2**), we only show one Event Array.

3.2. The Data Structure of Event Array

The structure of Event Array is (PushSupplierID, EvDataIOR, EvData, State, Priority, ElapseTime). Here, "PushSupplierID" is the ID of the Supplier. "EvDataIOR" is the IOR (Interoperable Object Reference) of an Event. "State" is used to record whether the Event has been received by all the "Consumer" of this Channel. If yes, "State" will be set, and at the same time, this Event will be put into destroying list, waiting for destroying. "EventData" is the Event data, and "Priority" records the priority of the Event. "ElapseTime" is the time duration that the Event exists in the Event Channel (Seciton 4.4 will give the details for the last two parameters).

3.3. Communication Quality Control

The Event Monitor is introduced to improve the communication quality, as Fig. (**2**) shows. It monitors the communication process and makes sure that all Event data is transferred to all consumers in time and with high quality.

3.3.1. Event Monitoring Process

(1) The Event data is sent from Supplier to Consumer Proxy, which further sends the Event to Event Channel and notifies the Event Monitor at the same time.

(2) A timer is opened for this Event by the Event Monitor.

(3) The Event Channel passes the Event to Consumer Manager, and then to Supplier Proxy, which finally pushes the Event to Consumer.

(4) Consumer notifies Supplier Proxy that it has successfully received Event.

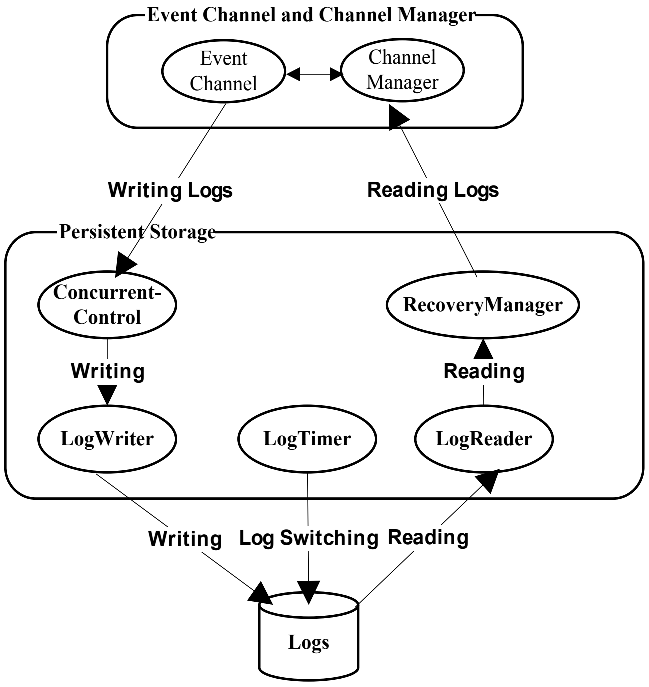

Fig. (3). Persistent Storage unit of our message middleware system.

(5) The Supplier Proxy sends the success message further to Event Monitor.

(6) If Event Monitor received all the success messages from the Supplier Proxy before the timer's alarming time, we can delete the Event from the Monitor Array. Otherwise, Event Monitor will request Event Channel to resend the Event.

We have to change the return value of push method to meet the requirement of Event Monitor. In addition, we have to design the data structure of the Monitor Array.

3.3.2. Changing the Return Value of Push Method

In the standard IDL, the return value of push method is "void". In above Event Monitor, we can see that Supplier Proxy need feedback message. Thus, we change the return value of push method from "void" to "Boolean" in order to meet this requirement. That is,

boolean push (in any data) raises(Disconnected)

where returned value being true means that the Event has been sent successfully, and false unsuccessfully.

3.3.3. The Data Structure of Monitor Array

The data structure of Monitor Array is (PushSupplierID, EvDataIOR, EventState). In the structure, "PushSupplierID" is the ID of Supplier. "EvDataIOR" is the IOR of the Event. "EventState" is an array which records whether the Consumer has successfully received the Event. Its specific definition is,

typedef boolean EventState[n]

where n is the number of Consumers that is connected to Event Channel.

Many Suppliers may share one Event Monitor. We can handle this race condition by two means. One is to build one Monitor array for each Supplier, and the other is to introduce lock mechanism to share the Event Monitor in a mutual exclusion manner. Because each row of Monitor Array has little information, it takes little time and overhead to maintain these locks. Therefore, we adopt the latter method to address this concurrency problem.

3.4. Priority Control

We add new overloaded method in the IDL of Proxy-PushConsumer,

Boolean push (in any Data, in short Priority)

Raises (Disconnected)

where we have 5 levels "Priority" from 1 to 5, and 5 is the top priority. When Supplier send the Event tagged with priority, the push method should be invoked to push Event to Event Channel.

The Events in Event Array of the Consumer Manager is ordered by a product between "Priority" and "ElapseTime", which shows how long the Event has been stayed in the Event Array. This order can assure that high priority Event is sent immediately and at the same time, low priority Event is not starved.

4. PERSISTENT STORAGE

Fig. (**3**) shows structure of the unit of Persistent Storage of our message middleware system. This unit is responsible for the persistent storage of the Event data (backup of the Event data) and the recovery of the Event data. Event Channel will write logs in specific time (please refer Section 4.1).

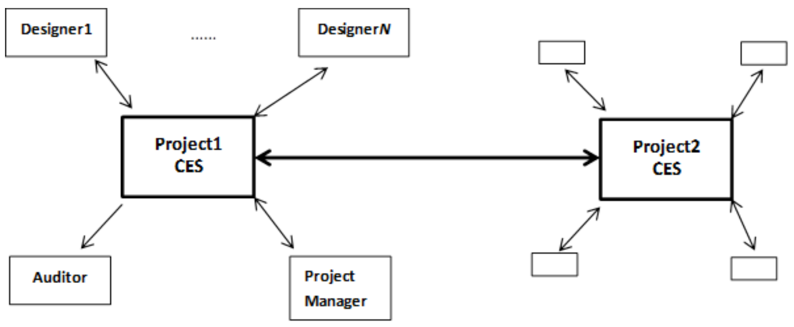

Fig. (4). Application and communication between CES.

The detail process has two steps. In the first step, ConcurrentControl schedules the Event Channels by introducing lock mechanism. In the second step, LogWriter will be invoked to write logs to hard disk. Log has to follow one specific data structure, as shown in Section 4.2. LogTimer is used to trigger Event log switching every a specific time interval. Event Channel Manager will initial the recovery process, after Event Channel fails and breakdowns unexpectedly. First, LogReader sends the requested Events to RecoveryManager, and, RecoveryManager sends them further to Event Channel Manager. Second, based on the user information written in logs, the Event Channel will be recreated by RecoveryManage and Event Channel Manager, and then Events will be recovered.

In order to get the backup of Events data, two problems should be addressed. One is when to write logs to disk from memory, and the other is how to structure the log file. We will give the details in the following subsections.

4.1. Log Writing Time

(1) When a user joins, logouts, suspends, transfers and destroys an Event Channel, its activities should be logged.

(2) In the following 3 situations, we will write Event logs. The first is when the Event Channel memory is 30% full. The second is when the thread is timed out. For example, we can set a timer, which will alarm every 5 seconds. The third is just before one specific Event will be removed from the Event array.

4.2. Log Specification

(1) The structure of user activities logs is as (UserType, UserIOR, EvChannelIOR, Time). Here “UserType” has three options, “Event Consumer”, “Event Supplier” and “Supplier/Consumer”. “UserIOR” is the IOR of the user, and “EvChannelIOR” is the IOR of Event Channel.

(2) The structure of Event data is (PushSupplierID, EvChannelIOR, Any, EventState, Time). Here, “PushSupplierID” is the ID of Supplier, and “EvChannelIOR” is the IOR of Event Channel. “Any” is the Event data. “EventState” is the state whether the Event data has been successfully received by all consumers (please refer 3.3.3). With this state, after recovery, we can resend the Event to the consumers which have not received the Event successfully. “Time” is the time when the Event was sent.

5. APPLICATION AND EXTENSION

In this Section, we give an application for our message middleware. In a designing group, there are many roles, like designer, auditor and project manager. In Fig. (**4**), project 1 has N designer, one auditor and one project manager. When the designer chooses to join project 1, he chooses the two-way mode. That is, he can receive the Event message from the CES (CORBA Event Service) or send the Event to CES. On the other hand, the auditor chooses read-only mode. That means he can only receive the Event message from CES. During the project, if a new designer applies to join in CES, he can choose to receive past Event messages, since our CES has backed up the Event data.

In a project, there are often lots of sub-projects. It is necessary to share the designer information between these projects CES. In order to achieve this goal, we should add a new method for interface EventChannelManager,

```
CosEventChannelAdmin:: transferEvent
 (in String SourceChannelName,
in String DestChannelName,
 in String EvDataIOR)
Raises  (EventChannelNotFound);
```

As Fig. (**4**) shows, when Designer1 want to get some Event message from project2 CES, he can invoke this method. Then, he can get any Event message in the Event channel or Event message that has been backed up.

CONCLUSION

Asynchronous and multicast communication is one of the most important topics in the application of distributed applications. In this paper, we give a design for a message middleware system based on Event Service. With this middleware system, distributed agents can communicate and cooperate with each other. Our message middleware can be embedded in the ERP system, taking the responsibility for the message transfer and storage.

In recent years, the Internet of Things technology develop quickly. Our real-time middleware can serve as the message middleware among distributed objects, such as a refrigerator or a table etc. However, the middleware should support protocols more than CORBA. In addition, the whole structure need to be more agile for communication between mobile device. As a future work, we would like to explore these issues.

CONFLICT OF INTEREST

The author confirms that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

This work is partially supported by NSFC (Project Number: 61300089, 61300084, 71373035), the Research Foundation of Education Bureau of Liaoning Province, (Project Number: L2013502, L2014540), the Doctoral Research Fund of Liaoning Province (Project Number: 20131023), Technology Innovation Fund from Ministry of Culture of China (Project Number: 2014KJCXXM12), Fundamental Research Funds for the Central Universities of 2015 entitled "Human motion retrieval and recognition based on sensors", China Postdoctoral Science Foundation (Project Number: 2014M561228), the Xiniuniao Innovation Foundation of CCF-Tencent 2014 (Project Number: CCF-TencentI AGR20140112), and Doctoral Research Fund of DLNU (0710-110005).

REFERENCES

[1] S. Tanenbaum, *Distributed Operating System*, Delhi: Dorling Kindersley, 2009.

[2] Object Management Group, Event Service Specification, 2002.

[3] Object Management Group, Notification Service Specification, 2002.

[4] T. Lei, W. D. Yang, and T. Wang, "Design and Optimization of Application Integration Based on Notification Service," In: *Proceedings of IEEE 3rd International Conference on Software Engineering and Service Science*, pp. 224-227, 2012.

[5] P. Gore, R. Cytron, and D. Schmidt, "Designing and Optimizing a Scalable CORBA Notification Service," In: *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, pp. 196-204, 2001.

[6] S. J. Xue, and W. Z. Sun, "Real-time radar data transfer based on corba event service," *Journal of Wuhan University of Technology*, vol. 32, no. 6, pp. 93-97, 2010.

[7] N. N. Yang, and J. Yang, "Networking schema of safety production monitoring and control system for coal mine based on CORBA event service," *Mining R & D*, vol. 28, no. 4, pp. 56-57, 79, 2008.

[8] H. Wang, J. F. Xu, and Z. Gao, "The application of CORBA middleware to the distributed flight control," *Computer Application and Software*, vol. 23, no. 5, pp. 15-18, 2006.

[9] Y. Chen, Y. Q. Wang, and Y. Liu, "Extended and distributed event service design based on CORBA," *Journal of Computer Application*, vol. 31, no. S1, pp. 138-143, 2011.

[10] J. S. Zhang, and J. Wu, "Research and implementation of fault-tolerant event service based on CORBA," *Microelectronics & Computer*, vol. 24, no. 6, pp. 166-169, 2007.

[11] F. H. Gong, and M. L. Qi, "Design of asynchronous communication based on event service of CORBA," *Electronic Design Engineering*, vol. 18, no. 5, pp. 22-28, 2010.

[12] H. Y. Song, P. J. Wang, W. Li, and X. F. Li, "Implementation of extended and light CORBA event service," *Computer Engineering and Design*, vol. 28, no. 20, pp. 4847-4849, 2007.

Received: November 28, 2014 Revised: January 09, 2015 Accepted: January 20, 2015