Lessening Forwarding Load of Application Server in Distributed Multi-User Systems

Mingyu Lim, Hyungseok Kim, Jee-In Kim

Department of Internet & Multimedia Engineering, Konkuk University
{mlim, hyuskim, jnkm}@konkuk.ac.kr
doi:10.4156/jnit.vol2.issue2.4

Abstract

This paper proposes a communication middleware that provides application developers with simple event transmission methods in an efficient and consistent way regardless of the communication architecture. Consistent end-to-end transmission is achieved via an internal forwarding scheme. Using the proposed scheme, the communication middleware under an application internally forwards an event to one node or to a group of nodes. Qualitative and quantitative experiments show that the proposed internal forwarding scheme reduces the overhead of the application’s forwarding tasks.

Keywords: Message Forwarding, Communication Middleware, Communication Architecture

1. Introduction

The Internet has become a common infrastructure for connecting many users (or nodes) interacting with each other in various ways to share their contexts. The Internet has become prevalent with developing multi-user applications in both academic and industrial societies, such as instant messaging services, multi-user online games and networked virtual communities. These applications frequently need to distribute messages to multiple nodes. In traditional network programming models using pure sockets, an application can establish a connection with remote nodes, and it can send and receive a message to/from them. There have been various underlying communication support by framework or middleware approaches. However, these approaches still assign the responsibilities of implementing various high-level transmission modes on developers, and they overlook the efficiency of the frequent message distribution.

First, developers must take care of socket-level channels and implement various transmission mechanisms according to application requirements. For sufficient distribution mechanisms, what they should think about is the number of transmission destinations and the socket type. Existing approaches do not provide sufficient distribution mechanisms because they are focus on their specific applications.

Second, applications can be burdened by distributing messages among end nodes as the size of messages or receiving nodes increases in specific communication architecture such as the client-server model. In this model, two client applications can interact with each other only with the mediation of a server application regardless of the transmission mode used. A server checks incoming messages, and if the destination is other clients, then the server forwards them to the destination. A server application would not be concerned with message forwarding if the underlying communication module performs the forwarding. It would also be more efficient if a client can designate the target nodes instead of a server, and if a message is internally forwarded through a communication system at the server side. Then, client applications in the client-server model can communicate as if they are directly connected like in peer-to-peer model.

In this paper, we propose a communication middleware (CM) that provides application developers with simple transmission methods that address the aforementioned issues. Applications can easily send events to a selected node or a group of nodes regardless of the adopted communication model by calling high-level transmission functions with different options. Consistent end-to-end event transmission is achieved via an internal forwarding scheme. A node can call the same APIs of CM with the same destination nodes no matter which communication model is used. In the client-server model, a client application can send an event with directly designating destination nodes. The CM under the server application then internally forwards the received event to one or a group of nodes, and this reduces the forwarding overhead of the application. Qualitative analysis and experiments show that the
The proposed internal forwarding scheme makes the message transmission in multi-user applications much easier than the existing middleware approaches, and it eases the applications’ burden of forwarding tasks.

The remainder of this paper is organized as follows: section II describes related work in terms of various communication frameworks for multi-user systems. In section III, we discuss the design consideration of consistent message transmission mechanisms and its forwarding overhead. Section IV and V introduce our communication middleware, and propose the detailed internal forwarding scheme. We analyze the proposed scheme in sections VI. Finally, in section VII, we conclude the paper with the description of future work.

2. Related work

Adaptive Communication Environment (ACE) [1] is an open-source object-oriented framework that implements many core patterns for concurrent communication software. ACE is designed to help developers of high-performance and real-time communication applications, and simplifies a way to develop network applications. However, it puts much responsibility on developers to develop a distributed application, because they focus more on software modules that concern an efficient way to manage relatively low-level signal, process, thread and socket. Using ACE, developers still have to take care of and implement the different requirements of message distribution mechanisms.

Common Object Request Broker Architecture (CORBA) [2] is an architecture standardized by the Object Management Group (OMG) that enables software components in a same application or a remote host in a network to interact with each other. Developers use an interface definition language (IDL) to specify the interfaces through which objects communicate by way of a normal method call. While CORBA provides a standardized and normalized means of easy way for objects in distributed systems to communicate, developers who would like to make a distributed application need additional knowledge of the internal CORBA architecture, components and protocols. Due to problems incurred by the heavy and complex specifications of CORBA, developers devised Internet Communications Engine (ICE) [3] that has similar functionalities, but is much lighter than CORBA. It, however, supports various ways of remote-procedure-call (RPC) style object interaction that are relatively low-level and still overlooks much higher level requirements for communication architecture and end-to-end message distribution.

Generic Communication Middleware (GCM) [4] addresses a generic solution for distributed message based communications between different types of applications using different transport techniques and communication paradigms. The goal of GCM is to facilitate the development of distributed applications by providing simple and easy communication middleware. GCM operates in both client-server and peer-to-peer models, and provides different message delivery models for one-to-one and one-to-many transmission. However, an application has message forwarding overhead as it checks all incoming messages if they need to be forwarded or not. The one-to-many delivery does not provide sufficient modes.

Mockets [5] is a communication middleware specifically designed for wireless networking scenarios. Applications using mockets can use several delivery services with different communication semantics such as reliable/unreliable and sequenced/unsequenced message delivery. The mockets middleware also provides the semantic message classification such as cancellation/replacement of some messages in a queue. It also supports other communication services that are required for wireless communication environments. However, the mockets middleware has the same problem with other existing middleware approaches, as it does not consider the various types of distribution mechanisms and the forwarding overhead.

Multiple Network Interface Socket (MuniSocket) [6] is a more efficient TCP-based socket model. The MuniSocket supports middleware-level parallel message transmission by utilizing multiple physical network connections on a cluster, that is transparent to applications. User messages are partitioned into uniformly sized segments that are transferred in parallel through multiple network interfaces. As it provides only another type of socket APIs, it does not support high-level message distribution mechanisms.
Lessening Forwarding Load of Application Server in Distributed Multi-User Systems  
Mingyu Lim, Hyungseok Kim, Jee-In Kim  
Journal of Next Generation Information Technology. Volume 2, Number 2, May 2011

User-centric Communication Middleware (UCM) [7] is another approach which provides high-level abstraction especially for the multimedia communication applications. It supports user membership, session management, call-back notification, and self-management interface. Although the goal and requirements of UCM are similar to our work, its minimum requirements of high-level communication support are limited to the multi-user multimedia applications. For example, it assumes only client-server model and insufficient message delivery schemes.

FlexFeed [8] is a mobile-agent based communication framework for the battlefield scenario. It provides an efficient and adaptive communications infrastructure to support and extend edge warrior capabilities and provide access to critical information at any time, while at the same time ensuring optimal resource utilization and security. FlexFeed is designed for future combat systems which has specific application requirements. It focuses on data streaming in heterogeneous systems with limited computational and communication resources. But, FlexFeed applications overlook their forwarding overhead.

Communication middleware is used even in humanoid robot architecture [9]. Ng-Thow-Hing et al. developed communication subsystems for delivering sensory information data to distributed modules. They used three different subsystems: the Cognitive Map (CogMap), Distributed Operation via Discrete Events (DiODE), and Multimodal Communication (MC). As these subsystems are used in one humanoid robot, they do not contain any multi-user aspects. It will need to be considered if more than one robot starts to communicate with each other.

In summary, existing middleware approaches for supporting communications provides low-level message delivery features and connection management, or focus on application-specific functionalities. Application developers need to implement different details of message distribution according to application requirements, which causes the message forwarding overhead. If a communication middleware provides consistent message distribution mechanisms, it reduces such overhead.

3. Consistent end-to-end message transmission

When we consider end-to-end message transmission provided by a communication middleware for multi-user applications, there are different transmission requirements according to application characteristics. Table 1 shows a possible transmission mode with the combination of reliability and destination numbers. One-to-one transmission can be done by reliable (TCP) or unreliable (UDP) channels. One-to-many transmission may require a socket type, too. Unreliable one-to-many transmissions can save network bandwidth if the underlying network supports multicasting. To set multiple destinations, an application requires a means to manage groups of remote nodes. If an application needs to add more than one channel at the same transmission mode, then what should be considered becomes more complicated. In this case, an application should manage the sockets per node in order to avoid duplicate message delivery to the same node.

<table>
<thead>
<tr>
<th>Table 1. Transmission mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable Communication</td>
</tr>
<tr>
<td>One-to-one TCP one-to-one</td>
</tr>
<tr>
<td>One-to-many TCP one-to-many</td>
</tr>
</tbody>
</table>

Various types of distribution mechanisms can be adopted differently depending on communication architecture. Communication models are divided into three categories: client-server, peer-to-peer and hybrid models. The hybrid model means a peer has a connection both with a server and with other peers. Communication among end nodes (a client or a peer) can be direct or indirect based on the existence of a server. In the peer-to-peer model, all the communication of peers is direct because there is no server. However, in the client-server and hybrid models, there is a case of indirect communication if a client would like to send a message to other clients. Especially in the client-server model, a client should send a message to a server that forwards it to the final destination clients because there is no direct connection among clients. Another problem is that a client needs to define the event type that
will be forwarded by the server. This means that a server application checks the incoming messages and distinguishes which message should be forwarded. However, it is more intuitive for a client to set another client rather than a server as a target node. Instead, the message delivery to final destinations is undertaken by an underlying communication system. CM supports consistent event transmission methods in different communication models through the internal forwarding scheme.

4. Communication middleware

4.1. CM architecture

CM is a communication middleware that enables developers to make multi-user applications in an easy and simple way [4]. CM is separated from application and it provides high level APIs for supporting various communication architecture, user membership management, event management and transmission of various contents. CM consists of three main manager modules: communication manager, event manager and multi-session manager. The communication manager manages communication channels created by the applications, and it provides send/receive methods of different socket wrappers. The event manager converts application level events created by the application to low level messages that are transmitted by the communication manager. The multi-session manager internally handles the received events and automatically manages users’ memberships in different levels of a session and a group. These manager modules are integrated with an intermediate stub module that directly interacts with an application layer. Developers use the stub module for high level control of CM.

4.2. Event management

Once users (client or server applications) are connected to a system, they interact with one another by sending messages and processing received messages. From the application developers' viewpoint, it is crucial to be able to easily create, send and receive an arbitrary message. To reduce the effort of defining a message with the primitive byte arrays, sending and receiving via socket functions and parsing received messages, CM wraps such required steps in the event manager introducing a high-level event. The middleware itself internally uses events as a means of communication, and an application can simply create, send and receive an event.

An event consists of two parts: a header and a body as shown in Figure 1. The event header contains internal characteristics of an event. The byte number field indicates the total number of bytes of an event. It is required when the event manager separates an event from the sequence of bytes delivered by the communication manager.

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event number</td>
<td>8</td>
</tr>
<tr>
<td>Event type</td>
<td>16</td>
</tr>
<tr>
<td>Event ID</td>
<td>24</td>
</tr>
<tr>
<td>Handler session ID</td>
<td>32</td>
</tr>
<tr>
<td>Handler group ID</td>
<td></td>
</tr>
<tr>
<td>Distribution session ID</td>
<td></td>
</tr>
<tr>
<td>Distribution group ID</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 1.** CM event format

Event type and ID fields provide a way to classify events. Events are classified into different types, depending on a common purpose. Each event belonging to a type is identified by an event ID. The
communication middleware defines several event types (the system event types) for internal use and another type (the user event type) that developers can explicitly use to define their own events on demand. An application can use not only user-defined events but also system events if required. For example, the middleware provides pre-defined update events such as user membership, user location and object addition, removal and change. They are delivered to the application layer after being processed by the system. This makes it possible for an application to capture the system events.

The handler session and handler region fields let the system know which module of the received node has to handle an event. The default value for both fields is the null string that means that the multi-session manager takes care of an event. If the handler session field is set to a specific session name, the event is delivered to the corresponding session manager. These two fields are used only for the middleware operations and the application developers do not need to concern themselves about them.

The session and region distribution fields aim to support an efficient method of one-to-many event redistribution. A delivered event sometimes should be forwarded to other nodes. In this case, a sender can set the distribution fields to a session or a region where an event is distributed. If a session name is set to the session distribution field of an event, the middleware of the recipient node forwards the event to every node connected to the session. The distribution fields are especially useful in the client/server architecture where a server always forwards events sent by a client to a set of other clients.

The event body has different fields that are defined by the system or a developer based on the event type and ID. While system type events have pre-defined body fields, the user type event allow developers to define arbitrary body fields on demand. We describe details of user-defined event in the case-study section.

5. Internal forwarding

A notable feature of CM is that it supports one-to-one and one-to-many communication among client nodes regardless of the adopted communication architecture. Especially in the client-server model, if a sending client sets a receiver client as a destination, then the CM at the server side forwards an event to the destination client. The client CM sends an event to a server CM which then searches for a channel of the destination client, and forwards it. Therefore, a server application does not need to be involved in the forwarding process any more. This feature gives a client application an illusion of direct connection with other clients even in the client-server model. Figure 2 shows the difference between typical forwarding and the internal forwarding scheme in the one-to-one case. CM’s internal forwarding eases server application’s burden of message forwarding management.

![Typical vs. internal forwarding](image)

Figure 2. Typical vs. internal forwarding

Considering various transmission modes and end-to-end transmission, the Stub module in CM provides four modes of event transmission methods that an application can call, as is shown in Figure 3. A send() method sends an event to a specified remote node represented by a string name; it provides unicast transmission using a stream (CM_STREAM) or a datagram channel (CM_DATAGRAM) set by an option parameter. An application can easily send an event to a target node because it does not need to care for the channels and the target node information. The application only specifies a target
node as a string name, and it sets a transmission mode as a parameter. An event is delivered to a target client even in the client-server model by the internal forwarding scheme of CM.

![Figure 3. Event transmission methods of CM](image)

A cast() method is called for one-to-many transmission. The required parameters are the same as those in the send() method, except the second and third parameters. As this method performs multiple one-to-one transmissions, it requires a name of a destination group rather than a specific node name. As a group belongs to a session in CM, a session name is also required to restrict a target group. Therefore, the cast() method needs two parameters for a destination group. If both the session and group name are set, then CM sends an event to all members of a group in a specific session. If the value of the group name is null, then an event is sent to members of all groups in a designated session. If both the session and group names are null, then the event is sent the same way as that of the broadcast() method; sending to all connected nodes. This method also supports the internal forwarding similar to that of the send() method. A client application, even in the client-server or hybrid models, can call the cast() method with a target session and group name that are different from the client’s current session and group. As a client does not know who is a member of different groups, CM internally sends an event to the server. The server CM then finds the target members and forwards the event to them.

A broadcast() method is called when a node sends an event to all connected nodes, so it does not need a target parameter. When a client invokes this method in the client-server or hybrid models, CM also performs the internal forwarding similar to the previous transmission methods.

A multicast() method can be used only if multicast communication is enabled in the underlying network. An application can add a multicast channel with a multicast group name, or each group in the hybrid model opens a default multicast channel with a group name. Therefore, a node can send an event via the registered multicast channel name.

In addition to event transmission, communicating nodes can exchange content to interact with one another. For example, the middleware supports the simple file transfer capability using the file manager module. Using the file manager of the middleware, we can configure the source or destination folder, and request a file through a simple API with a requested file name and the file owner name as parameters. Since the file transfer operation is conducted as a background task, sending and receiving nodes can continue other interactions simultaneously.

The middleware also supports multimedia streaming as it is another important functionality in multimedia-based multi-user applications. For the streaming feature, developers can set bytes of memory in the data field of the user-defined event, and send it to more than one receiving node. Streaming can be implemented by sending and receiving the sequence of bytes. If the assigned bytes size exceeds the buffer limit of an underlying socket, the middleware divides the entire byte stream into smaller packets, under the limit, and sends them to their destinations. The middleware at the receiving side then recovers the entire byte stream and delivers them to an application.

### 6. Experimental results

As discussed in previous sections, CM supports intuitive event transmission modes regardless of the type of communication architecture. A client or peer can designate a target receiver or the target groups by a simple name, and it can set the reliability value as an option. The appropriate selection of channels and destination information is hidden by CM. Internal forwarding allows the application to use consistent event transmission methods for all possible communication models. Compared to other message transmission mechanisms, CM dramatically reduces the programming efforts for sending an event with various transmission modes. As shown in Table 2 and 3, the existing mechanisms require the codes of the server application in order to figure out the message type that should be forwarded to other clients, find the channels to be used and forward a message to the destination. All of the latter
steps are done by CM, and the server application explicitly needs to do nothing. From the client’s viewpoint, the client should send a message to different targets according to a communication model. For sending a message to another client in a typical communication model, a sending client sets a server as the target in the client-server model. In peer-to-peer or hybrid models, a sender directly sets a client as the destination. However, an application using CM can call the same functions and target nodes without considering the communication models.

<table>
<thead>
<tr>
<th>Table 2. Required codes of a server application</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Existing mechanisms</strong></td>
</tr>
<tr>
<td>- To figure out a forwarded message type</td>
</tr>
<tr>
<td>- To find an appropriate channel or a set of channels</td>
</tr>
<tr>
<td>- To send a message through the found channels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Transmission methods of a client application</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Existing mechanisms</strong></td>
</tr>
<tr>
<td>Client-server</td>
</tr>
<tr>
<td>- Destination: a target node or a set of nodes</td>
</tr>
<tr>
<td>Peer-to-peer and hybrid</td>
</tr>
</tbody>
</table>

We conducted an experiment in the client-server model to quantitatively measure the effectiveness of the internal forwarding scheme. In this experiment, we calculated the number of received events on the server application side. A client randomly generates two event types. One type is required to be forwarded to another client and the other type is to be handled by a server application. As the rate of forwarded events varies according to the applications, we experimented with different rate values. A client randomly generates 100 events with the forwarding rate and sends them to the server. The number of checked events is the average value of 100 times of iterations. As shown in Figure 4, a server application using the typical forwarding approach checks all received events regardless of the event type. However, the proposed scheme requires a smaller number of events because CM deals with the forwarded events instead of the application. In addition, the higher the rate of forwarded events is, the less the burden of the server application will be.

![Figure 4. Typical vs. internal forwarding (number of events)](image)

In the next experiment, we measured the transmission delay of events. In this experiment, all events are assumed to be forwarded events. A client sends a sequence of events to a server, which then sends
them back to the sender. An event has a field that records the moment at which the events are sent by a client. This field is used when a client receives an event back and calculates the transmission delay. We measured the average delay after a sequence of event transmissions with increasing the number of events. Figure 5 shows the comparison between the typical and internal forwarding schemes. The internal forwarding scheme reduces the end-to-end transmission delay by around 10 milliseconds.

![Figure 5. Typical vs. internal forwarding (transmission delay)](image)

7. Conclusion

In this paper, we proposed an internal forwarding scheme and consistent event transmission methods in different communication models. By the proposed approach, CM provides multi-user application developers with simple APIs to send an event with various transmission modes. As such, the proposed internal forwarding scheme reduces the programming efforts of the communication aspect and end-to-end transmission delay. Based on the proposed scheme, we plan to extend the current CM to support dynamic communication models in which a multi-user system changes its communication model at runtime depending on the number of users or other system requirements.

8. Acknowledgements

This research was supported by a grant (07High Tech A01) from High Tech Urban Development Program funded by Ministry of Land, Transport and Maritime Affairs of Korean government.

9. References

[2] Author1_Name, Author2_Name, Web Caching and Replication, Addison-Wesley(Publication_Name), USA, 2014
Lessening Forwarding Load of Application Server in Distributed Multi-User Systems
Mingyu Lim, Hyungseok Kim, Jee-In Kim
Journal of Next Generation Information Technology. Volume 2, Number 2, May 2011