A general collaborative platform for mobile multi-user applications

Mingyu Lim, Niels Nijdam, Nadia Magnenat-Thalmann
MIRALab, University of Geneva
Battelle, Batiment A, Route de Drize 7, 1227 Carouge, Switzerland
{Mingyu.Lim, niels.nijdam, thalmann}@miralab.unige.ch

Abstract

The aim of this paper is to propose a communication middleware which makes it possible to easily and efficiently develop the networking support for multi-user applications. Even though existing middleware and development tools provide lots of functionalities to realize distributed applications, they are purely low-level services passing the most development to developers or too specialized for a specific application. It brings a challenging issue of how to support sufficient and general high-level mechanisms by the middleware. To meet different networking and interaction requirements of multiple users, our approach is to address various possibilities of the communication architecture, the user membership management, the content transmission mechanism and the event management. In each requiring feature, the proposed middleware provides several options with application-level APIs and configuration so that the different interaction needs of a multi-user application can be easily handled in the developers’ point of view.

1. Introduction

As the Internet has become a common infrastructure to connect lots of users interacting with each other with various way of sharing their contexts, it is becoming prevalent to develop multi-user applications in both academic and industrial societies such as instant messaging services, multi-user online games, networked virtual communities and so on. This major trend also makes existing stand-alone applications for single user be extended and evolved to ones for multiple users. If an application developer wants to develop a multi-user system or change a single-user application for the multiple-user purpose, he/she has to take into consideration not only intrinsic functionalities of the system but also how each system can be connected and communicate with each other to support multiple users. To realize this, he/she could do pure socket level network programming or use other application level communication toolkit or middleware. However, in the case of socket level programming, application developers have to take care of everything relating to communication from the low level socket management to the high level aspect of distributed user interaction. Although there have been existing communication middleware systems which provide high level wrappers of basic functionalities for network programming, they still need to be improved in terms of easy and simple development of multi-user systems. In one hand, some communication environments still put many responsibilities to developers instead of supporting diversities of fundamental communication means. On the other hand, other toolkits are focus on specific applications and it is difficult to use them for another systems. For supporting developers, it is a dilemma how to balance between the freedom and simple way of the development.

In this paper, we propose a common communication middleware which enables developers to make multi-user applications in an easy and simple way considering a common step to implement them. We extract general requirements to develop an application for multiple users in terms of communication architecture, user membership management, event management, message delivery and process and support for transmission of various contents. The proposed communication middleware which is separated from application provides high level APIs to support required functionalities. It consists of four main modules: communication manager, event manager, multi-session manager and stub module. The communication manager manages communication channels created by applications and provides send/receive methods of different socket wrappers (e.g. TCP and UDP) and file communication. The event manager converts application level events created by application to low level messages that are transmitted by the communication manager. When a message is delivered from the communication manager to the event manager, it parses the message to an application event and forwards it to both multi-session manager and application so that they can handle the event. The multi-session manager handles received events internally and automatically manages user membership in different levels of user groups. All the three managers are...
integrated with an intermediate stub module which directly interacts with an application layer. Developers use the stub module for high level control of the middleware or also can access each manager module for low level functionalities if needed. An overall structure of a communication node integrated with the middleware is easily set by a configuration file in which we can set the communication role of the application (server or client), session management (multiple sessions or single session) and user membership control (no user authentication or authentication by application).

The rest of the paper is organized as follows: section 2 describes the related work in terms of various communication frameworks for multi-user systems. In section 3, we discuss the requirements and design consideration of a common communication middleware. Following section 4 and 5 in which we propose a detailed architecture of the proposed communication middleware and its implementation, we illustrate the integration procedure of an exemplary application with the middleware and a case study of the work in section 6. In section 7, we conclude the paper with the description of future work.

2. Related work

In this section, we discuss and analyze several middleware tools which help users develop distributed and multi-user applications by providing various functionalities with different goals and views.

Adaptive Communication Environment (ACE) [1] is an open-source object-oriented framework that helps developers of high-performance and real-time communication applications. It provides a rich set of software modules, but it still puts much responsibility on developers to develop a multi-user purpose application because they focus more on software modules which concern an efficient way of managing relatively low level signal, process, thread and socket.

Common Object Request Broker Architecture (CORBA) [2] is an architecture standardized by Object Management Group (OMG) which enables software components in a same application or a remote host in a network to interact with each other. 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 should have additional knowledge of the internal CORBA architecture, components and protocols. Because of problems incurred by the heavy and complex specifications of CORBA, developers devised Internet Communications Engine (ICE) [3]. While it supports various ways of RPC style object interaction, ICE provides relatively low level supports and still overlooks much higher level of requirements for multi-user systems.

SOAP [4] is a message passing protocol for exchanging XML-based messages using HTTP/HTTPS and is widely used in Web applications. As SOAP uses application layer transport protocols such as HTTP and SMTP, it allows relatively easier way of transmitting messages rather than transport layer protocols (e.g. TCP and UDP). However, there have been critics about the performance due to verbose XML message format. In our view, it also does not provide any high level features required for efficient support of distributed multi-user applications because it is originally designed as a messaging protocol, not a middleware or toolkit level.

There are also many application level middleware approaches which supports different application specific functionalities according to specific needs for multi-user systems. Among compelling multi-user applications are online games and many research works [5][6][7][8] have devised dedicated middlewares and technical schemes that are tightly coupled with their own games. Many open source or proprietary game engines supporting multi-user capability [9][10][11][12][13] focus on 3D graphics engine and various components stuck to game application and provide still limited networking support with fixed client/server model. Even though such middleware approaches support well customized functionalities meeting their own requirements, it is not suitable to use them in other multi-user applications that have different purposes.

3. Design consideration

When we design a middleware or architecture with a layered approach, there is a trade-off between the freedom of users and simplicity of development as discussed in the previous section. In one hand, when we provide only low-level functions and put most responsibilities to users, they have more freedom to develop and improve functionalities using the provided functions. However instead, they should implement details of the new modules. On the other hand, if a system provides users with high-level APIs which have such functionalities, it allows developers to develop a module in an efficient way. In this case, however, APIs sometimes restricts flexibility to change the supported modules to the limited number of possible options. Extremely speaking, this is the problem of using a pure programming language or a high-level toolkit. Therefore, we focus on multi-user applications possibly in general and analyze fundamental requirements which could efficiently help developers both with high and low level features.

From the experience of work on network framework for distributed virtual environments (DVEs) [14], we have found several fundamental features (communication architecture, user membership management, transmission scheme and event
management) required for developing multi-user systems which is discussed in following sections.

4. Proposed architecture

In this section, we propose a communication middleware, CM which takes into consideration on design issues in the previous section. Our approach includes various features such as communication architecture, user membership management, content transmission and event management, all of which are required for developing multi-user applications.

4.1. Overall architecture

The proposed communication middleware (CM) aims to provide an easy and efficient way of developing a multi-user application. It supports various functionalities with options for different requirements of developers. The middleware deals with multi-user issues which have to be implemented by developers if it has only fundamental networking supports. Our system has a role of bridge between an application and underlying network infrastructure. Among basics to do this is to deliver messages and contents between these two entities, by which communicating nodes can interact with each another. With APIs provided by the proposed middleware (CM), application developers can create, send, receive, and process a event. In addition to dealing with events, it supports other operations which detect a specialized event and conduct a dedicated service according to the event type. To support them, we follow the layered approach to design the middleware.

Figure 1 shows an overall architecture of the proposed middleware (CM). In the point of view of applications, the proposed middleware (CM) consists of four main modules in order: CM stub, multi-session manager, event manager, and communication manager.

4.1.1. CM stub

The CM stub is a core module which interfaces an application. A developer can access most of supporting functionalities of the proposed middleware with this module. In general, it provides APIs to start and stop the CM, register and deregister an event to be used among CM nodes, send an event, and assign an event handling callback function which is called by the system whenever it receives an event from a remote node. All events created in an application and needed to be sent pass through the CM stub until the underlying network layer. The CM stub then notifies an application of every received event from a remote node through the registered callback function so that the application can deal with incoming events.

In addition to these fundamental operations, depending on the application type (a client or a server) and the communication architecture (client/server, peer-to-peer, or hybrid), which are assigned in a configuration file by a developer, the CM stub provides appropriate useful functions with various options. For example, if the application is a client type in the client/server architecture, the CM stub offers a function to connect to a server. What a developer has to do is to set the server information like the IP address and port in the configuration file and call the simple connection function. All the rest operations required for creating a new channel, initializing it, and binding addresses are performed by other underlying responsible module such as the communication manager and the event manager.

4.1.2. Multi-session manager

The multi-session manager manages a different level of user interaction area according to the application requirement. It defines the number and organization of sessions and regions in a hierarchical structure. The multi-session manager contains session managers, each of which handles a session and a session manager can have more than one region manager which manages a region. The developer can organize the structure of sessions and regions using the configuration file or APIs provided by manager modules. In the proposed architecture, we need at least one session and region to allow users to interact with each other. It means that the minimal unit of interaction is a region. A user always has to belong to a region. By default, we define a session as an independent interaction area and regions inside a session as an interrelated interaction area. While users in different session usually do not exchange events, those in different regions in the same session frequently interact with each other. However, this assignment between a session and a region is just logical concept,
and a developer can define his/her own relationship among them through APIs. For example, a multi-session manager provides APIs which can distribute an event to different sessions and regions. A developer can assign and distribute an event to specific sessions or regions. In addition, although static configuration of the session and region is given by the configuration file, the manager module also provides functions to manage them dynamically such as creation/deletion of a session or a region. The session and region structure is tightly coupled with the user membership management which we discuss in another section.

4.1.3. Event manager

The event manager is in charge of an event in the system. An event is a high-level form of exchange method between application nodes. As it includes semantics to be exchanged so that it can be understandable at the application layer, high-level manager modules (the multi-session manager, session manager, and region manager) and an application uses an event as a way of information and content exchange. One of key role of the event manager is to change an event to a low-level message (sequence of byte array) which is delivered to the underlying communication manager, and vice versa. To the contrary, an incoming message given by the communication manager is converted to an event in the event manager which then delivers it to the multi-session manager for the internal operation. The event manager also forwards the converted event to the CM stub module which eventually delivers it to an application for the event process at the application layer as well.

4.1.4. Communication manager

The communication manager controls messages and provides APIs to manage communication channels. The main role is to send and receive messages to/from underlying network. To support this, the communication manager runs a separate thread for waiting any incoming message. Whenever it receives a message from the network, the communication manager forwards it to the event manager. It also uses its own send functions embedded in each channel when a message to be sent is delivered from the event manager. If an application creates a channel, it is maintained in the communication manager as a channel list. Every event channel in the list is checked by the receiving thread and any received message is notified to the event manager. Therefore, the communication manager supports asynchronous communication among nodes as it does not guarantee for a channel in the list to receive an immediate reply in a blocking state. However, the communication middleware also allows a developer to make synchronous communication by managing each channel in a separate way. The system provides different dedicated communication sockets which wrap native socket APIs and makes it possible an easy way to open and close a channel depending on demand. Supported socket types are a server socket, a stream socket, datagram socket, and a multicast socket. The server socket is used in the server side which waits and opens connection with requesting clients. The counterpart of the server socket is the stream socket which is the wrapper of a normal TCP socket. The datagram socket is for UDP connection of either a server or a client. The multicast socket is on top of the datagram socket for multicast communication. It is useful when routers enable multicast routing. Developers can enable the synchronous communication by directly using the wrapper sockets instead of the communication manager.

4.2. Communication architecture

Our communication middleware supports three models of communication architecture: client/server, peer-to-peer and hybrid. As each model has pros and cons according to the purpose of an application, system environment, etc., a developer can choose which communication architecture he/she will adopt. The communication architecture is determined in a global configuration file of the communication middleware. A developer just specifies one of three architectures with pre-defined terms (CM_CS for the client/server, CM_PP for the peer-to-peer and CM_PS for the hybrid model). If we choose the client/server or the hybrid model, it is also required to specify an application type because there are two types of communicating nodes: a server or a client. SYSTEM_TYPE field in the configuration file distinguishes such two application types by putting SERVER or CLIENT. The communication middleware then initializes in a different way according to the determined system type.

4.3. User membership management

The communication middleware supports different levels of user membership management based on the session and region concepts. The multi-session manager maintains all connected users, a session manager manages users who belong to the corresponding session and a region manager deals with region members. Whenever there is a change of user membership in any level, the middleware notifies an application of the membership change so that it can keep track the up-to-date information. In addition, each manager modules allow for the application to access current membership in a different level by providing APIs. Thus, developers have options of whether they follow the default notification from the middleware or make their own membership management policy in the application layer.

The middleware also provides flexible ways of users to connect to the system in order to interact with each other. We have to take into account various login policies depending on the application requirements. An application could require a strict user
authentication/authorization or allow any user to connect without specific identification procedure. It could have multiple sessions and regions to manage a huge number of users or could not need any session or region management due to a few numbers of users. The middleware configuration file has two options (LOGIN_SCHEME and SESSION_SCHEME) for developers to set them enable or disable according to which type of membership policy they would like to use. If the LOGIN_SCHEME field is enabled, it means that the system requires user authentication/authorization procedure. As the middleware does not provide any specific mechanism, it puts the responsibility of detailed procedure to the application layer. When a server receives a login request from a user, it forwards the request to the application, which conducts its own user authentication process. Instead, the middleware provides an API so that the application informs the system if the requester is a valid user or not. The middleware then replies to the requester with the result. If the LOGIN_SCHEME is disabled, the system requires no authentication process. The user requests login with a user name and the server just accepts this request. After login process is completed, the next step is to join a session and a region of the system. We can enable or disable the SESSION_SCHEME field for this phase. If it is disabled, the system does not require multiple sessions and regions and a login user immediately joins a default session and region. On the other hand, the enabled SESSION_SCHEME means that the system manages multiple sessions and regions and a user needs to select which session and region he/she would like to join. A client can get current available session lists from a server and join/leave a session by provided APIs.

4.4. Event management

Once users (client or server applications) are connected to a system, they interact with each other by sending messages and processing received messages. In the application developers’ point of view, it is a crucial factor 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, the middleware wraps such required steps in the event manager introducing a high-level event. Not only the middleware itself internally uses events as means of communication, but also an application can simply create, send and receive an event.

![Figure 2. CM event format.](image)

As shown in Figure 2, an event consists of two parts: a header and a body. The event header contains internal characteristics of an event. The byte number field indicates 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. Event type and ID fields give a way of classifying events. Events are classified into different types depending on a common purpose, and each event belonging to a type is identified by the event ID. The communication middleware defines several event types (the system event types) for internal use and another type (the user event type) which developers can explicitly use to define their own events on demand. An application can use not only user-defined events but also some system events if needed. 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 in the system, which makes it possible for an application to catch the system events. The handler session and handler region fields let the system knows which module of the received node has to handle an event. Default value of the both fields is null string which means that the multi-session manager take care 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 care 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.
4.5. Event and content transmission

In the previous section, we discussed how an event is organized and how it is delivered to each module inside the communication middleware. In this section, we describe how we send an event to remote nodes. The middleware provides several APIs to efficiently send an event to one or more users. When we send an event to a user, we take a user name as a destination node instead of the IP address or the domain name because the middleware maintains the user name and channel information and can find which channel is used by which user. We can set an option whether the event will be sent via TCP or UDP as the middleware keeps both channels. To send an event to more than one user, there are several possibilities according to the communication architecture as depicted in Figure 3. In the client/server architecture (Figure 3(a)), a user send an event to a server with setting the distribution fields in the event header which will enable the server to automatically distribute the event to other users. In the peer/peer or hybrid architecture, we can call a dedicated function “multicast” which by default sends an event to users in a region which the sender currently belongs to. It is available when the network supports multicast transmission (Figure 3(b)). We can then assign a multicast address to a region and this function sends through the assigned address. If the multicast is not available or a user wants to send to another region or session, the function takes additional parameters like the distribution fields in the event header. We can set the destination session or region in this function, which then sends the events to users in the corresponding group (Figure 3(c)).

**Figure 3. Event transmission.**

In addition to the event transmission, communicating nodes can exchange content to interact with each other. For example, the middleware supports the simple file transfer capability by the file manager module. For the future work, we plan to add multimedia streaming support as well because it is another important functionality in multimedia-based multi-user applications.

5. Integration into an application

In this section, we describe how to use the communication middleware library inside a multi-user application. The development environment is the MS Visual Studio 2005 on Windows XP. Although it is not yet implemented for the multi-platform environment, it can be ported to other platform without significant effort because the communication middleware is implemented mostly with standard C++ except some platform specific features like the thread.

Various multi-user applications can be built based on the CM modules. Depending on whether an application has a role of client or server, it uses the APIs of CM client stub or server stub, respectively. A server and client application integrated with CM can select any type of application, rendering subsystem, and user interface as the communication middleware is independent of an application.

First, the application developer should add the CM library to the application, assign a (client/server) stub instance, and register a function which will process the receiving events. Once the function is registered to CM, any event received by CM is forwarded to it. Events can be identified with their ID. The function should be implemented by the developer according to events that he/she would like to handle.

The second step is to set up the configuration file. At the client side, all the developer can do is to assign the IP address and port number of the server machine so that the client application can connect to it. In case of the server application, the developer decides the communication architecture, the application type, IP address, port number, the login and session scheme and other detailed fields.

After configuring the required communication features, the developer can start CM which then initializes all internal channel and communication information and waits events from remote nodes. In the case of a client application, the further steps are required to connect to a server. Different connection procedure is applied depending on the login and session scheme defined at the server side. Once at least two communicating nodes are connected, the application can send events via the stub module and can process receiving events by the registered function. The developer can define an application specific event using the user event where he/she can add arbitrary event fields with primitive data types. Therefore, when the registered receiving function in the application receives an event, it is totally up to the application developers how the application handles it. An event can update application specific data, or call a rendering subsystem to present the effect of the event to users. In addition, the client application can use other CM APIs to manage the connection with the server when it changes the session, logs out from the server, or disconnects itself from the server. Figure 4 illustrates overall integration sequences in the application’s point of view.
6. Case study

In this section, we describe the usability of the communication middleware with simple examples to make sure how easily we can use the networking support when we build a multi-user application. Linking with the communication middleware library, an application is allowed to use APIs and set the supporting features as discussed in previous sections. For a simple example, we assume a client/server based application without any login and session scheme. In the server application, a developer can assign the communication architecture, the login scheme and the session scheme in a configuration file as shown in Figure 5. SYS_TYPE designates the type of an application which can be SERVER or CLIENT. COMM_ARCH is the communication architecture which the application will use. CM_CS means the client/server model. LOGIN_SCHEME and SESSION_SCHEME are indicating whether the application uses a specific user authentication and more than one session. In this example, we do not perform any user identification process and use only one default session. At the client side, what we should set is only the server information (IP address and port number).

![Figure 4. Integration procedure.](image)

**Figure 4. Integration procedure.**

To start the communication middleware in an application, what we can do is to call a start function. Then the server and client application initializes all required channels and user membership information and waits for events from a remote node. The client application should call a login function to connect to the server before sending and receiving any event. After the establishment of connection between the server and the client, we can send an event to interact with each other. A developer can assign an event type, ID and required field and send it through APIs provided by the middleware. Figure 6 shows an example source code of a client which connects to a server and sends a CM user event.

```plaintext
CCMClientStub cm; // assign the CM client stub
cm.registerReceive(processEvent); // register to CM
cm.startCM(); // start the CM
cm.loginCM("user"); // login to a server
CCMUserEvent ue; // assign a CM user event
ue.setID("testEvent");
int val = 1;
ue.setField(CM_INT, "testInt", &val);
cm.send(&ue, "SERVER"); // send an event
```

**Figure 5. Server configuration**

To start the communication middleware in an application, what we can do is to call a start function. Then the server and client application initializes all required channels and user membership information and waits for events from a remote node. The client application should call a login function to connect to the server before sending and receiving any event. After the establishment of connection between the server and the client, we can send an event to interact with each other. A developer can assign an event type, ID and required field and send it through APIs provided by the middleware. Figure 6 shows an example source code of a client which connects to a server and sends a CM user event.

![Figure 6. Client example](image)
use other supporting classes of the middleware to extend their applications as well. It means that developers can still access to other information provided by underlying CM classes if they want to develop new functionalities that current CM does not support. The full description of how to use the middleware and event definition is described in separate technical document [15][16].

7. Conclusion and future work

In this paper, we proposed a generic communication middleware for multi-user applications. The goal of the proposed middleware is for developers to easily take care of application-level networking supports while keeping the flexibility of development. To clarify common requirements for building a multi-user application, we classified possible required features into several sub-topics: communication architecture, user membership management, content transmission and event management. Although an application needs all these features, we can have varying possibilities of detailed mechanisms. Thus, we proposed several possible options in each multi-user requirement. For the communication architecture, the middleware support client/server, peer-to-peer and hybrid approaches. We support the hierarchical management of user membership mechanism. Developers can benefit from various content transmission schemes from the simple one-to-one to one-to-many transfer by simple APIs. To exchange any message among communicating nodes, the middleware makes it possible for developers to define an arbitrary event with the efficient and intuitive interface.

To extend the current implementation of text and file transmission, we are working on integrating multimedia streaming facilities as the multimedia service like audio and video transmission among multiple nodes has been of great importance in the multi-user application. Another topic for the future work is to take consideration of the scalability of the communication architecture. We are researching on an efficient way to organize more complicated communication architecture with more than one server.

Acknowledgement

The work presented was supported by the EU project INTERMEDIA (38419), in the frame of the EU IST FP6 Programme.

References