Designing the Communications Infrastructure of Groupware Systems

Sergio F. Ochoa[1], Luis A. Guerrero[1], David A. Fuller[2], and Oriel Herrera[3]

[1] Department of Computer Science, Universidad de Chile.
Blanco Encalada 2120, Santiago, Chile
{sochoa, luguerre}@dcc.uchile.cl

[2] Computer Science Department, Pontificia Universidad Católica de Chile.
V. Mackenna 4860, Santiago, Chile.
dfuller@ing.puc.cl

[3] Informatics School, Universidad Católica de Temuco.
Manuel Montt 56 Temuco, Chile.
oherrera@uct.cl

Abstract. In the development of groupware systems a well designed communications infrastructure is required, due to the high complexity of the communication scenario. Also, the design and implementation of coordination and collaboration mechanisms depends on the communications infrastructure. Actually there are no well known guidelines to design this infrastructure. Therefore, this paper proposes an architectural pattern that helps carry out the design of this communications infrastructure. The proposed pattern supports all the groupware systems communication scenarios, taking in account their particularities. This pattern has been used in the design of several groupware applications and a groupware framework with very good results.

1 Introduction

A collaborative application or groupware system supports the work performed by a group of collaborators who pursue a common goal. For the attainment of this goal, the groupware system must provide support in three essential aspects: coordination, collaboration, and communication, better known as the three groupware Cs [4]. Communication is the base for reaching coordination and collaboration.

Unlike distributed systems, collaborative applications share a common goal, which has to be reached by the collaborators through the coordination of their individual contributions and activities. Due to the existence of this common goal, communication becomes a way of supporting the coordination and collaboration mechanism that permits the group work. Therefore, the design of the communications infrastructure is very important, since there are many other design aspects that depend on it [14]. These are, for example, awareness, sessions and user administration, floor control, and notifications.

Considering the essential aspects that have been defined by Ellis, et al. [4], the groupware systems should be designed using a layered architecture [2] (see Figure 1). The reason is that each layer uses services from lower layers to do its work. This is

J.M. Haake and J.A. Pino (Eds.): CRIWG 2002, LNCS 2440, pp. 114–133, 2002.

similar to what happens with the data communication protocols or with operating systems. In this scenario, each layer carries out a specific function and communicates with the other layers through well-defined interfaces.

In groupware systems the communication layer should be in charge of providing the communication among the applications. The coordination layer should generate a shared vision of the group work. It coordinates the actions that are carried out individually, and generates a consistent vision of the group activities. Finally, the two superior layers correspond to the definition and the use of the elements that permit collaborative work, from the user viewpoint. Stratified architectures have many advantages, which have been widely discussed [2].

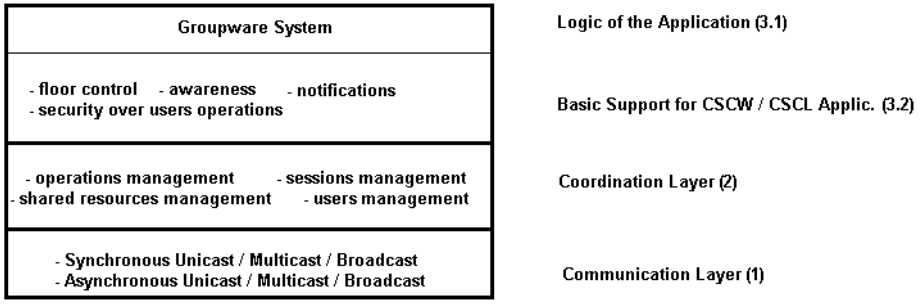

Fig. 1. Layers of a typical groupware system

Typically, in groupware systems, each component of the layers two and three should be designed and implemented over the defined communications infrastructure (or communication layer). Therefore, the implementation, extension or modification capacities of these components depend greatly on: (a) the services provided by the communications infrastructure and (b) the quality of they. Therefore, the infrastructure must be well designed, and isolated from the rest of the collaborative work support media. In this way, it is always possible to add new services to the communication interface, incorporate new communication technologies, and maintain and improve the existing infrastructure, without changing the superior modules of the application. To achieve this, an isolated, modular, and flexible communications infrastructure that permits communication on all the typical scenarios is required.

Today, there are no communication architectures that take in consideration the particularities of the groupware systems. Therefore, this article presents an architectural pattern called CAGS -Communication Architecture for Groupware Systems- which proposes a flexible and modular architecture to manage the communication in groupware systems. This pattern takes in account the particularities of this kind of communication which are described in the next section. Section 2 also presents the communication scenarios of groupware systems. Section 3 presents the related works. Section 4 describes the CAGS pattern. Section 5 shows how this pattern supports the typical communication scenarios. Section 6 shows one example of the application of this pattern. Section 7 presents the obtained results. Finally, section 8 states the conclusions of this work.

2 Communication in Groupware Systems

In CMC – Computer-Mediated Communication – scenarios there are at least four elements involved: a sender, a message, a channel, and a recipient. The *sender* is the process that sends messages. The *messages* are flows of bytes transmitted through a channel. The *recipient* is a process that receives, and possibly processes the messages. Finally, all these messages are sent through the *channel*.

Usually, every groupware system needs to implement several ways of messages addressing, such as: point-to-point, multicast, and broadcast. Moreover, each one of they can be implemented using a different ways of message delivering, such as: synchronous or asynchronous. We call *synchronous communication* those instances of communication where the sender can deliver messages to the recipient without explicitly storing these messages in the channel. A special case of this type of communication is audio and video streaming, also called *isochronous communication* [20], which requires that the messages be generated, transmitted, and received in fixed time intervals. We refer to *asynchronous communication* when messages sent by the sender may need to be stored by the channel before being delivered to the recipient. Summarizing, the possible communication scenarios in groupware systems are a combination of one message addressing way and one message delivering way.

The communications infrastructure of groupware systems is similar to that of the distributed systems, all though there are various differences. The more relevant differences are visible from the design viewpoint. Unlike distributed systems, the communications infrastructure of the groupware systems should be:

Isolated. To let the application be independent from the communication medium used. Also, to separate the communication tasks from coordination tasks. In that way, it is easier to design, implement and maintain a good communications infrastructure. Besides, due to the relationship that exists between communication, coordination, and collaboration, it is natural that the first be managed in an isolated way.

Complete. To support all the communications scenarios that may be present. In groupware systems, the communications scenarios that need to be supported can change along the time. For example, due to extensions of the application functionality. It can also be due to changes or to the incorporation of aspects that support the collaborative work, for example: awareness, performance, operation monitoring, floor control or security, among others. If the communications infrastructure does not support all the typical communication scenarios of the groupware systems, it will limit the functionality and the expansion capacity of any application implemented over it. For example, to implement a simple chat it is necessary to support only synchronous communication. If the chat implements connection awareness, it is possible to use the same infrastructure to support it. It is possible that after using it, problems of performance are encountered due the high amount of awareness messages in the network. The most obvious solution for this problem is to change the awareness from synchronous to asynchronous. It can be easy to do only if the communications infrastructure is modular and extensible.

Flexible and modular. To activate and deactivate communication scenarios according to the application necessity, without prejudice to it. Typically, the

groupware systems do not use all the known communications scenarios, and use only a subset of they. Incorporating the support needed for only the scenarios involved in the application generally is necessary for performance reasons, resource usage, maintenance and application simplicity. Therefore, the communications infrastructure must be complete, but must also be modular and flexible to support the different communications scenarios. This means that it must be possible to activate o deactivate the support for certain communications scenarios, without the rest of the application suffering because of this. In this way, only communication scenarios that are needed will be incorporated, without limiting functionality, nor the extension capacities of the collaborative application.

Open. So as not to condition its use to the fact that certain application architectures (e.g. Client/Server) can be used. In that way, the solution viability is guaranteed in any work scenario. Today, the majority of the proposed communications infrastructures condition their usage to certain type of applications, specific domains, or the use of certain software architectures.

These characteristics make communications infrastructures of groupware systems are different, but also more demanding than most communications infrastructures of current applications. This is one of the reasons why collaborative systems are difficult to implement, maintain and expand.

3 Related Works

In CMC the groupware systems need to manage communication among processes in any of the defined scenarios, but in a modular way. Building mechanisms that make this possible is not an easy task, for until this moment there are no known guidelines to support the design of communications infrastructure in groupware systems.

A very used way to represent the design of this kind infrastructure is through an architectural pattern [2]. There are no well known architectural patterns to design the communication in groupware systems, but in distributed systems there are a number of choices (see [18]). Schmidt et al. proposes a pattern language for middleware and distributed application. It is the most complete pattern system in this area. However, it is of little use in the groupware systems area, because, for example, it does not separate the communication services from the coordination services. Typically these architectural patterns do not include the restrictions of groupware systems, because they are designed for distributed systems.

On the other hand, there are middleware technologies that carry out the communication in this scenario, like for example, RPC -Remote Procedure Call-, MOM -Message Oriented Middleware- or TP-distributed Transaction Processing- [11]. These are not complete, flexible, modular nor isolated, but there are open. This is due to that these technologies were designed to support distributed systems.

Actually, the ORB -Object Request Broker- technology is the most used for distributed systems. There are two standards for it: CORBA – Common Object Request Broker Architecture- and OLE/DCOM – Object Linking and Embedding / Distributed Common Object Model –. Similar to those mentioned before, they do not consider the communication restrictions of the groupware systems because they propose solutions for distributed systems.

As a consequence of the lack of support in the development of groupware systems in general, over the last few years, a great number of frameworks to support the collaborative application development have appeared. Among the more known frameworks are: TOP-Ten Objects Platform – [9], Habanero [3], GroupKit [8,17], COCHI – Collaborative Objects for Communication and Human Interaction – [13], and JSDT – Java Shared Data Toolkit – [1]. These implement some elements of CAGS, and therefore, do not incorporate all the communications restrictions mentioned before. In section 4.8 a more detailed analysis of this framework is presented.

In groupware systems there are no well known patterns for designing the communications infrastructure. The few patterns that have been defined in this area are focused on the collaborative application design [10]. Therefore, this paper proposes the CAGS architectural pattern as a guideline to design and implement the communications infrastructure of groupware systems. This pattern is the result of more than six years of experience in designing and implementing groupware systems.

4 The CAGS Pattern

As Buschmann [2] said, an architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities and includes rules and guidelines for organizing the relationships between them. The CAGS architectural pattern establishes a modular and extensible structure for the overall management of the communication in groupware systems. To specify this pattern the structure proposed by Buschmann will be used [2].

4.1 Pattern Name

CAGS – Communication Architecture for Groupware Systems.

4.2 Context

In a groupware system, both the collaborators and some system processes need to exchange messages among themselves. In addition, sometimes it is necessary to keep a record of messages in order to subsequently operate on them. The types of communication that may be found in this scenario are point-to-point, multicast and broadcast. At the same time, these can be synchronous or asynchronous.

4.3 Problem

Unlike distributed systems, the groupware systems need communications infrastructures designed in a modular way, that potentially are capable of including all the communication scenarios possible. This infrastructure must also be capable of providing a minimal set of services that permit the implementation of coordination

and collaboration mechanisms of the application. The communication scenarios supported by this infrastructure can change along the time.

4.4 Solution

In order to manage the different communication scenarios, taking in account the restriction mentioned in section 2, the architectural pattern called CAGS is proposed. It is based on the layer pattern [2] and its main goal is to facilitate the delivery of a set of communication services that permit the implementation of coordination and collaboration mechanisms over it.

To define CAGS we consider a groupware application composed by a *work interface*, *background processes* and *a communications infrastructure*. The communications infrastructure permits the sending and receiving of messages among processes in several communication scenarios. The background processes permit the coordination of all the operations that form part of the collaborative work, directly or indirectly. Examples of these processes are the following: the shared-memory manager, the floor control manager or the session manager, among others. The user interface permits the collaborative work by using the background processes and the communications infrastructure.

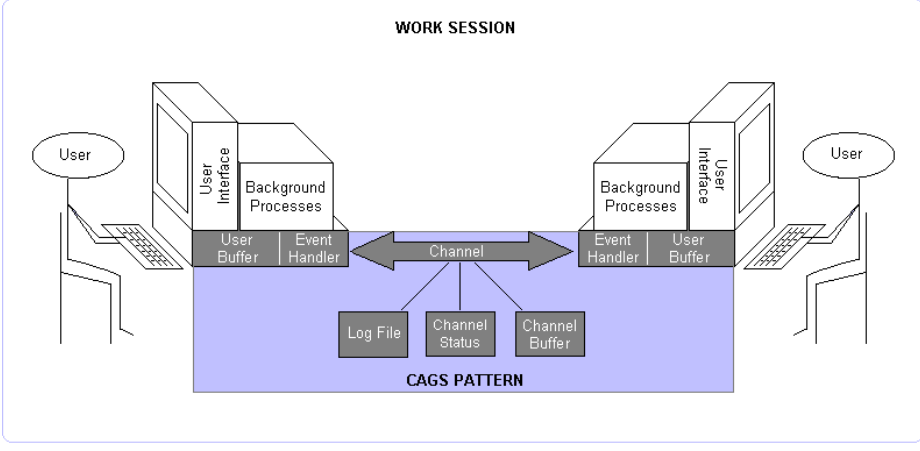

Fig. 2. Communication elements of CAGS pattern

The CAGS pattern proposes an architecture with six components: *user buffer*, *event handler*, *channel*, *log file*, *channel status* and *channel buffer*. The user buffer is used for buffering. It is only necessary in applications that synchronize video and audio transmissions. The event handler is the interface manager between the communications layer and the superior layers. It is in charge of receiving those messages whose recipient is a process running on the same computer. Also, it is in charge of capturing the events that should be transmitted, and sending them through the channel to the proper recipient processes. The channel takes care of the message transportation between the event handlers.

As shown in Figure 2, the channel should have three associated elements: a log file, the channel status and a channel buffer. Every message that travels through the channel could be properly stored in a log file. This component has a similar behavior to that of the command pattern [6], and it is useful, for example, to show the messages to a user who logs on late into the work session or to allow for operations such as "undo" and "redo". The channel status stores and maintains information about active applications, computers, processes and recipient groups that are using the communications infrastructure. This information is registered through background processes and is maintained through the event handlers. Once the information is registered as part of the channel status, it can be used to access the groups directly, using a group identifier, for example. If multicast is not available, this is a good way of simulating it. Finally, if during a message distribution, one of the users is not active, and the communication mode is asynchronous, the channel must save the message in the channel buffer in order to attempt a subsequent delivery. This buffer has a similar behavior to that of the *producers and consumers* pattern [7]. It is essentially used to give persistence to the messages in asynchronous scenarios.

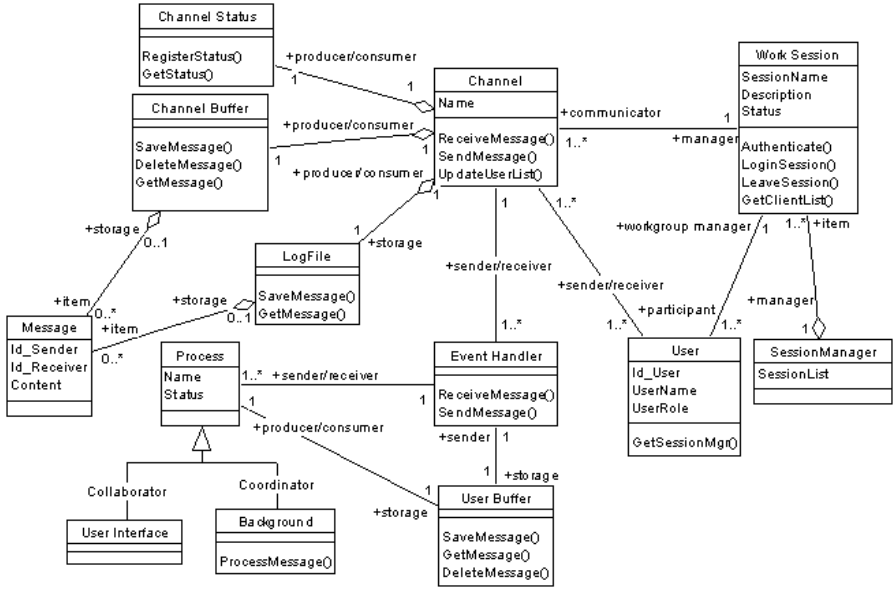

Fig. 3. Object classes diagram for the CAGS pattern

4.5 Structure

Figure 3 shows the UML class diagram of the CAGS pattern structure. In this diagram, the six components that form part of the pattern can be identified, as well as the relationships between them, and the relationships with other external components.

The external components, such as work sessions and sessions managers, belong to coordination layer and they show an example of how one coordination component can be linked to the components of the communication layer.

The communications infrastructure interacts with the user interface and background processes through the event handlers. On the other hand, these event handlers capture the events and act in consequence. Also, they receive the messages from the channel and send them to the corresponding destination processes. The event handler uses an API to encapsulate all functionality of the communication layer. Thus, when changing the implementation of the communication layer, the superior layer of the collaborative applications will suffer no changes.

The user buffer provides temporary storage of messages in order to synchronize the reception of some data types like audio and video. Only in this kind of application this component makes sense. The channel transmits/receives the messages to/from an event handler. It provides a communication media by means of messages for each defined communication scenario (see section 5). There are a number of basic features which the channel should provide, such as:

Correct message delivery and receipt: refers not only to the integrity of data, but also to the message generation and acceptance sequence. For this purpose, the event handler should interact with the channel to guarantee the reliable delivery and retrieval of messages.

Message storage: due to the fact that the recipient process is not always available at the moment when it is required, the channel should serve as a message buffer. This service uses the channel buffer to store the messages.

Message distribution: refers to the delivery of messages to the proper recipient processes, which conform a work session. It is carried out using the channel status information.

Message persistence: if it is necessary, it should be possible to store messages for additional operations upon them. This service can be implemented using the channel buffer and the log file.

The channel buffer allows the storage of messages for subsequent delivery. If it is necessary to keep the history of sent messages, the channel sends a copy of the message to the log file component. This log file component maintains all the work history.

Finally, the channel status component stores the information about applications, processes, computers and recipient groups that are using the communications infrastructure. The channel status organizes the information in a hierarchical way. The first level corresponds to active applications. The second level corresponds to the computers in which these applications are running. Finally, the third level corresponds to the active processes that are a part of the application. The way through these three information levels has a unique name associated to it and corresponds to a recipient identifier. This identifier is called *Recipient_Id* and is used to deliver the messages. On the other hand, it is also possible to associate a unique identifier to a set of Recipient_Ids. In that way, it is possible to setup an addressing strategy that permits efficient message distribution. All messages given by background processes or by user

interface to the channel should have a *Recipient_Id*. The channel on the other hand will be the one in charge of distributing them in the most appropriate way.

4.6 Dynamics

Next there is a description of some of the communications scenarios that may be found when using CAGS. These scenarios are part of the pattern dynamics.

Scenario I. A user enters a work session through a login process (Figure 4). To start this process *Proc1* sends a message to the event handler (E.H.) that requests a login for a work session. The event handler tells the work session that a new user wants to log. The work session validates and in case of success, registers the new user, and returns a response to the event handler. This registration involves the channel notification for it includes the new client and processes into the channel status component. Finally, the event handler tells the process the result of the login operation. To exit a work session, the scenario is similar.

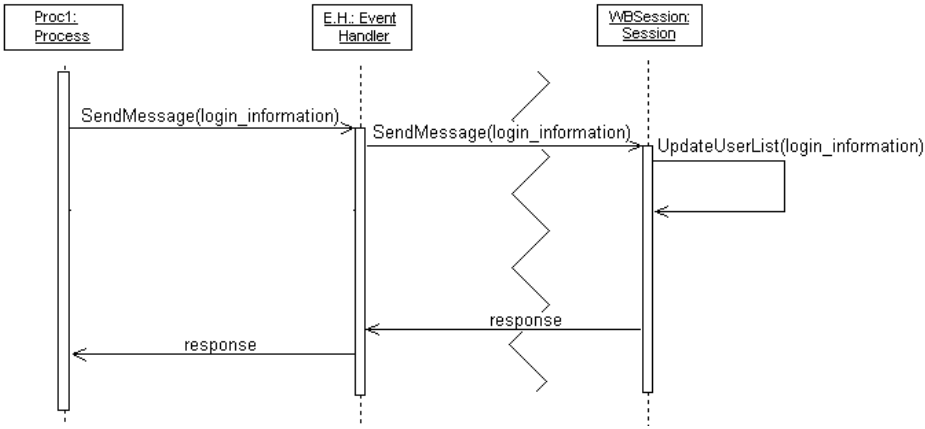

Fig. 4. Entry of a user into a work session through a login process

Scenario II. A background process sends a message to a set of background processes. In Figure 5, a process *Proc1* sends a message to its event handler. This event handler sends the message to the channel. The channel responds that it received the message, and proceeds to distribute it to every recipient event handler. Every event handler receives the message and delivers it to the proper process. To carry out an effective deliver of the message, the channel uses the information stored in the channel status.

Scenario III. The channel stores all the messages received in the log file. When the channel receives a message, it sends it to the log file before sending it to the recipient event handlers. The log file receives the message and stores it. Then, the channel uses the channel status information to carry out an effective delivery to the recipient event handlers.

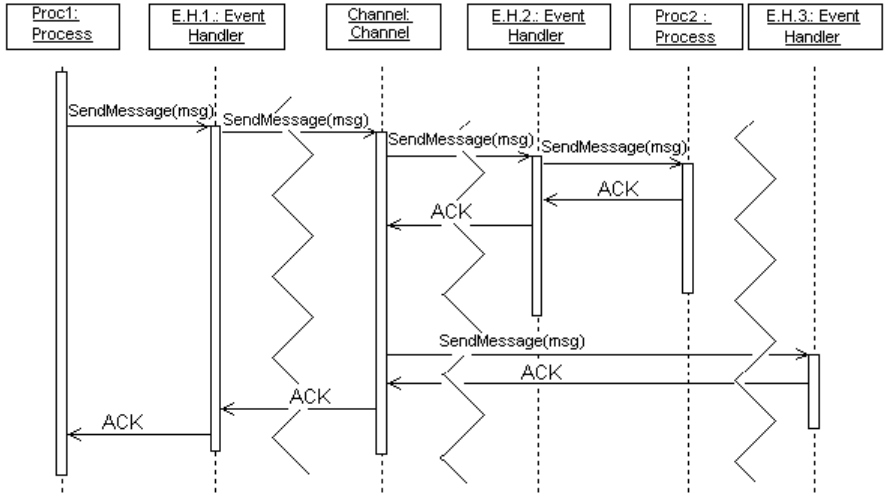

Fig. 5. A process sends a message to a set of processes

4.7 Implementation

To carry out a CAGS implementation it is necessary to take some decisions. The most important decisions relate to the following:

1. the required functionality of the pattern. It must be decided if the log file is wanted or not, as well as the storage time for the messages in it. For example, last hour, last week, last 100 messages, etc.
2. whether the communication is synchronous or asynchronous, in order to decide if the *channel buffer* component will be used or not.
3. if audio or video reception (*isochronous communication)* will be involved. A *user buffer* could be needed.
4. what messages can be generated both by user interface (UI) and by background processes (BP). It is also necessary to specify what messages can arrive at each kind of process (UI and BP), and what to do with them.
5. whether to use a client-server or a peer-to-peer architecture. The decision made will determine where to store the list of users logged into the work session, as well as their corresponding IP numbers.
6. If a *channel buffer* or a *log file* is used, where will they be implemented, and where will the messages be stored. For example, in client-server architecture these two components, as well as the *channel* component, can be implemented as part of the server, or else, they can be implemented as part of the client.

On the other hand, every work session must have at least one channel. It is advisable to create a default channel when a work session is created. The majority of the applications have only one channel. If an application requires several channels, it is necessary to indicate the channel by which the message is going to be sent. To

implement the pattern components, it is necessary to use distributed objects. The consequences of this pattern are part of the section 7.

4.8 Adaptations

The CAGS pattern can be used for the development of groupware systems as well as for the creation of frameworks for the development of such systems. Also, several components of the CAGS pattern are present in most frameworks for groupware application development kits, such as: TOP -Ten Object Platform- [9], Habanero [3], GroupKit [8, 17], COCHI -Collaborative Objects for Communication and Human Interaction- [13], and JSDT -Java Shared Data Toolkit- [1]. Below is a brief description of how these frameworks incorporate some components of the CAGS pattern to implement their communications infrastructure.

TOP. The TOP platform operates according to a client/server architecture [9] and it was designed taking in account the CAGS pattern. It does not implement multicast, but it can be easily simulated. TOP makes the channel component act as a server responsible for the distribution of messages. There are also modules in charge of providing: the shared objects persistence, session management, user and floor control. These modules are in the coordination layer and use channel services. The *event handler* that is part of the channel is divided into two parts, one in the client application and the other in the server. The latter uses an in/out port to allow communication through messages, using the server as an intermediary. TOP allows the development of synchronous and asynchronous, but not isochronous applications. Therefore, the channel implements the *channel buffer*, but not the *log file*. This framework separates the communication layer from de coordination layer. Also, it permits the access to the communications infrastructure through a set of predefined services.

Habanero. This platform allows only synchronous work, and is used to transform single-user applications into shared applications, and to develop new groupware systems. Habanero does not support multicast, and the *channel* component is implemented through a server, which is responsible for managing communications. The collaborative applications are implemented as a set of client objects, which interact through a *wrapped object*. This *wrapped object* is responsible not only for managing the application events, but also for coordinating the rest of the wrapped objects that form part of it. Communication is not performed by messages but by *actions*, which are orders distributed for execution. To support asynchronous work, ISAAC – Integrated Synchronous And Asynchronous Collaboration – [12] was developed, which implements an extension of the Habanero's *session model*. This new session model implements the *log file* and the *buffers* as a part of the model and not as a part of the channel. This framework also does not separate de communication layer from the coordination layer.

GroupKit. This allows the development of applications which only support distributed synchronous work. The *channel* component is implemented by a server,

which manages the transmission of messages by means of a process called *registrar*. The event handlers are in the client application and they interact directly with the server through an API. Since GroupKit supports only synchronous work, it does not implement either the *buffers* or the *log file*. In this framework the communication layer is mixed with the coordination layer. It does not implement multicast, but it can be simulated.

COCHI. This framework permits the use of synchronous applications and only does not implement multicast. It uses a client/server architecture so the *channel* component is implemented through a server, represented by the *communication media* component. The *channel buffer*, *user buffer* and *log file* components are not part of the framework since it only supports synchronous (and not isochronous) work. The *event handler* component is divided into two parts: one part is in the client application and the other in the channel (in this case, the server). This framework separates the communication layer from the coordination layer.

JSDT. It implements several CAGS components such as the: *channel*, *event handler*, *user buffer*, and *log file*. The channel is implemented as a server, and access to it is carried out through a client application. In JDST, the *event handler*, is divided into two parts: one sends messages and the other receives messages. JSDT is part of the message receipt only. Sending messages is the responsibility of the application, because processes are allowed to write directly on the channel, without going through the event handler. The message receipt function of the channel depends on the type of conversation (*Channel, ByteArray* or *Token*). In this framework, a *user buffer* exists for isochronous communication handling, such as audio and video, and can be used in both conversation modes: *Channel*, and *ByteArray*. JSDT does not provide asynchronous communication. However, this type of communication can be implemented, creating a channel buffer with the elements supplied by the framework. Typically JSDT does not separate the communication layer from de coordination layer, but a well designed application could simulate the existence of these layers.

5 Support to Communications Scenarios

The CAGS pattern should provide a modular and flexible support to all communication scenarios that were defined in section 2. Taking in account the ways of message delivering, the communication can be synchronous or asynchronous. Next, each one of these scenarios is analyzed.

Synchronous Communication Scenario. To support synchronous communication, the following components are needed, as a minimum: an event handler, a channel and the channel status. This is the minimal communications infrastructure to support synchronous communication with message discarding. This means that if the recipient process is not active at the time when the message is distributed, the message will be discarded.

On the other hand, if one works in a synchronous way but allowing replies of small sets of operations, it is necessary to include a log file as part of the communications infrastructure. An example of this is when one works using synchronization points. In this scenario, the shared-object manager is in charge of delivering the states of the shared objects until the last synchronization point, to any collaborator that enters the work session. Then, the communications infrastructure must deliver all the operations done from the last synchronization point to the collaborator.

In the case of isochronous communication [20] it is only necessary to add the user buffer to the minimal communications infrastructure. This buffer serves to store the messages received from the channel, since then they will be transmitted to the interface in fixed time intervals.

The message addressing way supported in this scenario is the same as that in the asynchronous communication scenario, they are: point-to-point, multicast and broadcast. To carry out each of the addressing ways, the channel status information is used. The point-to-point and multicast communication can be implemented through the Recipient_Ids stored in the status channel. Additionally, each application (not each instance of an application) has by default a special Recipient_Id which it is used to carry out the messages broadcasting.

Asynchronous Communication Scenario. On the other hand, in asynchronous communication the minimal infrastructure is composed by: an event handler, a channel, the channel status, a channel buffer. Depending on the information that the messages in the channel buffer have, a log file might be or not be necessary. The way that the messages are addressed is the same as in synchronous scenarios.

6 Pattern Application

In this section, it is shown how a simple groupware application, such as a shared whiteboard, can be designed using the pattern. In this example, it can be observed how some of the CAGS pattern elements, which are not present in the JSDT, can be implemented and integrated as specified by the pattern. The application has been built in Java, and was designed based on the following elements (see Figure 6): *user interface*, *GUI listener* (background process), *consumer* (event handler), *registry* (session manager), *work session* (session) and *channel* (channel).

The *user interface* (GUI) allows the user to interact using mouse and keyboard. The user operations are captured using the `KeyListener`, `MouseListener`, or `WindowListener` interfaces. A background process (GUI Listener) which is part of the coordination layer analyzes these operations. Depending on the type of event, and the object upon which the process was performed, this process will be in charge of whether to distribute the performed operation or not. Sending the information directly to the *channel* carries out this distribution. The channel transports the information, and then uses the recipient's *event handler* (consumers) as an intermediary between the target process and the channel. As the application is neither asynchronous nor isochronous, it does not need to implement buffers. Nor does it implement the log file, although in section 6.1, a variant of this application which does use a *log file*, is presented.

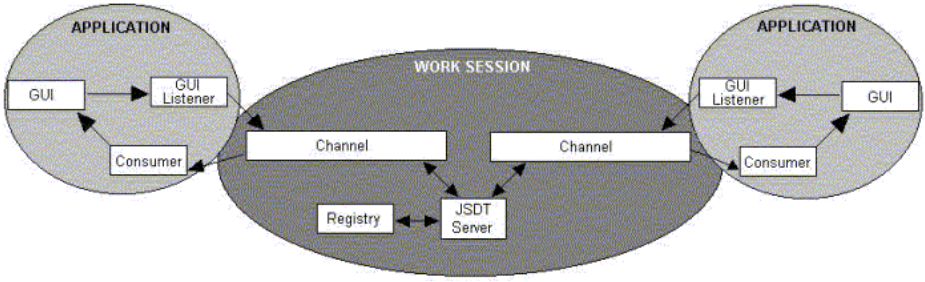

Fig. 6. Example of an application using the CAGS pattern and the JSDT tool

In figure 6, the channel, JSDT server and consumer are part of the communications infrastructure (communication layer). The Registry and the GUI listener are part of the coordination layer and they correspond to the background processes. Finally the GUI is part of the collaboration layer and it corresponds to the interface. It is the classification proposed in the section 4.4 (CAGS solution).

In order to support group work, our client application needs the server to have an active instance of the *session manager* known as *Registry*. Therefore, the server must verify if the *Registry* is running, and, if not, execute it. This allows work session, conversation, and user information recording. Then, at least one work session needs to be created, along with one or more conversations (communication channels) within the created session. These three operations can be observed in the following script segment, where the first task performed is the session ID building for the session that is going to be created (1). Later, it will verify if the *Registry* is running, and if it is not, it will be executed (2). Afterwards, a session (4) and a client for the session (3) are created. Finally, a *channel* type conversation is created (5), leaving the communications scenario ready to be used by any client application. Operations 1 to 5 correspond to coordination layer services. Operation 5 is the initialization of the communications infrastructure (communication layer).

```
(1)  url = URLString.createSessionURL(hostname, hostport, sessionType,
                                                          sessionName);
     try {
          /* Is the Registry running? */
(2)       if (RegistryFactory.registryExists(sessionType)==false) {
               RegistryFactory.startRegistry(sessionType);
          }

          /* Create a session */
(3)        client = new WhiteBoardClient("Server");
(4)        wbSession=SessionFactory.createSession(client,url,true);

           /* Create a channel type conversation. Add the client. */
(5)        wbSession.createChannel(client,"WBChannel",true,true,false);
           System.err.println("Setup and bound WhiteBoard server.");
      }
```

The *event handler* (consumer) will be activated every time a message arrives from the channel, and will execute the `dataReceived` method to process it:

```
      ...
      /** The WhiteBoardUser for this consumer. */
      WhiteBoardUser wbu;
      ...
(6)   public synchronized void dataReceived(Data data) {
        /* Handle the received data. */
(7)     wbu.commandLine = data.getDataAsString();
    }
```

The information that travels through the channel is of the *data* type (6), a proprietary JSDT format. In this application, what is sent/received are the commands or events into string format (7).

Once the first script segment is executed, it can be said that the channel is active, because the *Registry* is running, a session is created, and there is a conversation ready to be used. The only thing left to do is to log the client application into the channel. The simplest way to do this is the following: (a) enable the work interface (GUI), so that the users can use it; (b) log onto the channel, leaving the application interface ready to perform the collaborative work; (c) listen to the events that occur in the interface, and act accordingly (GUI Listener).

Then, it is necessary to carry out the login process. For it, the application should create a unique client ID (8). After this, the ID for the session is created to it can be accessed (9), a reference of this session is obtained (10), and the recently created client is added (10). Once the client is logged into the work session, it is necessary to obtain the references of the conversations to be used (11). Finally, it is necessary to register the client as a channel message consumer (12). Thus, the client application remains logged onto the channel, and is ready to begin the collaborative work.

```
private void connect() {
      ...
      /* Create a whiteboard client */
(8)   client = new WhiteBoardClient(name);
      /* Resolve the whiteboard session */
      try {
(9)     URLString url = URLString.createSessionURL(hostname,
                          hostport, sessionType, sessionName);
(10)     session=SessionFactory.createSession(client,url,true);
           ...
(11)     channel = session.createChannel(client,"WBChannel",
                                            true, true, true);
        wbConsumer = new WBConsumer(client.getName(), this);
(12)    channel.addConsumer(client, wbConsumer);
          ...
     }
```

Figure 7 shows two windows of the shared whiteboard application built, which are located on the same computer. It is the result of having used the CAGS pattern to define the communications infrastructure of the collaborative application. It can be observed that the application is modular, which simplifies its maintenance and extension.

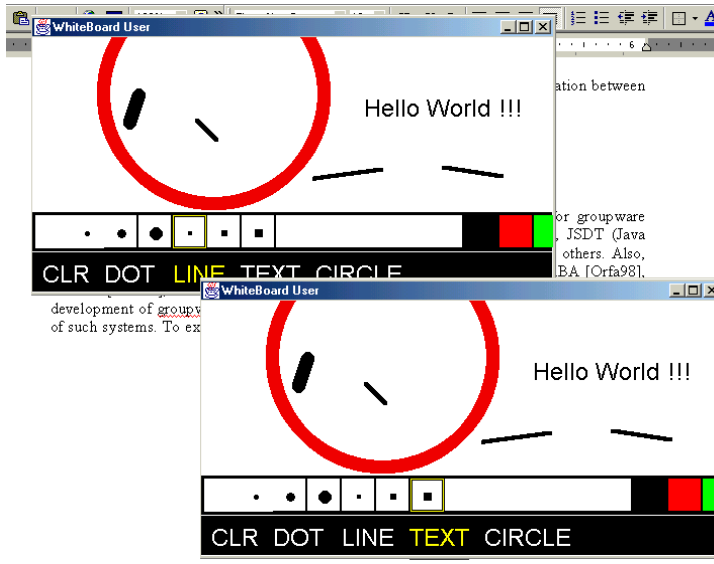

Fig. 7. Whiteboard Interface

6.1 Adding The *LogFile* Component to the Whiteboard

As previously stated, the shared whiteboard does not implement the *log file* and neither does JSDT. Therefore this is an example of how a CAGS component can be implemented or simulated through a design solution. The log file is important for the support of asynchronous work because it stores all the operations executed since the beginning of the session, in a chronological order. These operations are sent to each user that accesses the session later, so he/she can obtain an updated image of the current status of the shared objects. Figure 8 shows the incorporation of the log file to the communications infrastructure of the application.

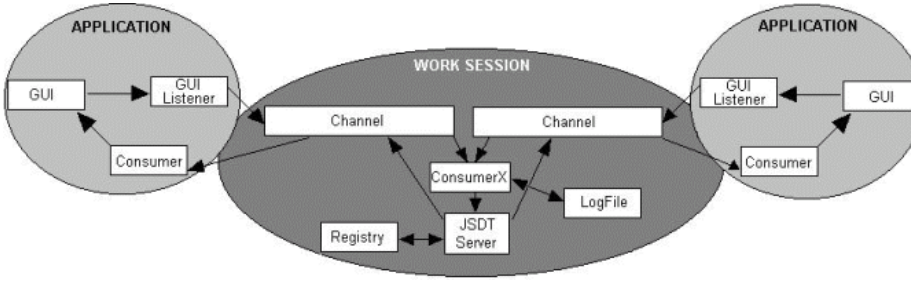

Fig. 8. Whiteboard application with log file

The log file implementation strategy is simple and consists of adding a "special" consumer to the server (*Consumer X).* This is a special consumer because it runs a task different from the rest of the consumers involved in the application. The function

of *Consumer* X is to analyze and conveniently store (in the log file) the messages that run through the channel, and to deliver these messages to any user who connects later onto a session. This strategy can be implemented using any framework that allows the consumer to listen to the commands that run through a channel.

The process for adding consumers is the same for clients and servers. To do this, it is necessary to incorporate into the channel, the client used to create it (18) and then register that client as a consumer (19). The rest of the work must be implemented inside the program of this consumer (Consumer X).

```
(18) wbSession.createChannel(client,"WBChannel",true,true,true);
       ...
     DrawingRecorder ConsumerX = new DrawingRecorder(ch);
(19) ch.addConsumer(client, ConsumerX);
       ...
```

One of the functions of this class is to store, in the log file, the messages that run through the channel (20). The other function is to send the stored messages to each new client that connects to the session (21).

```
class DrawingRecorder extends DrawingConsumer implements ChannelConsumer
         ...
         /** The list of all events or commands */
         private ArrayList event_list = null;
         public synchronized void dataReceived(Data data) {
         ...
         event_list = new ArrayList();
         ...
(20)     event_list.add(evt);
         ...
         If ((evt.operation = 1)&&(event_list.size() != 0)) {
            ...
(21)        SendToClient(evt.client, event_list);
            ...
```

7 Obtained Results

The CAGS pattern has been used to design several collaborative applications and also a framework for the development of such applications. This framework was called VisualTop [15] and it is based on TOP [9]. Also, it implements all components of CAGS and it separates the application in the three layers proposed in Figure 1. With VisualTOP, several collaborative applications were developed, like: an asynchronous collaborative editor, an electronic meeting system, a discussion forum and a synchronous desktop conferencing system. Some of these applications have been described in [15].

On the other hand, there are collaborative applications that have been developed using CAGS and others framework like TOP or JSDT. Some of the collaborative applications developed using CAGS and TOP are the following: an asynchronous collaborative editor, a discussion forum, an electronic meeting system, a brainstorming system and a voting system. All of them have been mentioned or briefly described in [9].

Among the collaborative applications developed with CAGS and JSDT are the following: a brainstorming system, a voting system, a chat, a shared whiteboard and a synchronous distributed slider. The major challenge that the developers faced during the development of these applications was giving communications support to the asynchronous scenarios. But, as was showed in section 6.1 this problem could be surpassed.

The best collaborative applications, regarding performance, maintainability and extendibility, were those developed with VisualTOP. The natural incorporation of CAGS to the platform causes that the designer does not have to worry about stratifying the application, nor designing the communications infrastructure. Among the advantages that CAGS has contributed to VisualTOP as much as the applications that use this pattern are:

Adaptability. The components of the communications infrastructure can be expanded or changed without having to modify the software modules that are supported by this infrastructure.

Reduced development time. The logical partitioning into a layered system and the modularity of construction using common services, offer efficiencies through the skill specialization and the component reusability. These practices will improve the system's quality and reduce lead times for their implementation and modification.

Flexibility. The features and capabilities of applications can be modified (e.g., they are scaleable and expandable) without changing the communications infrastructure. On the other hand, the communication scenarios that are parts of the infrastructure can be activated and deactivated according to the application's necessity, without prejudice to it

Maximize Flexibility in Managing the Communications Infrastructure. Changes to the communications infrastructure must be transparent to application programmers. This model permits changes in this infrastructure with little or no modification to the supported applications.

Reduce Complexity for Application Developers. Since much of the functionality in new applications will be provided through pre-written and pre-tested software services, the complexity of the collaborative applications development is reduced.

Complete Support. The CAGS pattern proposes a communication infrastructure that involves all communication scenarios of groupware systems and it takes in account the restriction mentioned in the section 2. In that sense, any application could be supported.

Open Architecture. In order to use the pattern, it is not necessary to use any software architecture in particular. In that way, the solution viability is guaranteed in any work scenario.

8 Conclusions

The communications in groupware systems are different to communications in distributed systems. The fact is that the design guidelines for communications

infrastructures of distributed systems can not be applied in groupware systems. In groupware systems the communications scenario is more complex and demanding, due that the communications infrastructure should be: flexible, modular, open, isolated and complete.

On the other hand, these characteristics make collaborative systems difficult to implement, maintain and expand. Therefore, a well-designed communications infrastructure is needed. Unfortunately, there are no well-known patterns for designing this infrastructure. The few patterns that have been defined in this area are focused on the collaborative application design [10]. Therefore, this paper proposes an architectural pattern called CAGS – Communication Architecture for Groupware Systems – It is a guideline for designing and implementing the communications infrastructure of groupware systems.

This pattern is based on the layer pattern [2]. Due its architecture, CAGS facilitates the delivery of a set of communication services that permit the implementation of a coordination and collaboration mechanism. The CAGS architecture is composed by six components: *user buffer*, *event handler*, *channel*, *log file*, *channel status* and *channel buffer*. With these six components the CAGS proposes a solution to manage the different communication scenarios of groupware systems, taking in account the restriction mentioned in the section 2.

In order to demonstrate the applicability of the proposed pattern, this paper shows how the most popular frameworks used for developing groupware systems implement several components of CAGS. Also, we explain step by step how this pattern can be used in order to design the communication aspects of a common groupware application. Several collaborative applications have been designed using this pattern, and also a framework for the development of such applications. This paper presents not only a pattern to handle these scenarios, but also a way to organize computer-mediated communication in order to easily implement the design aspects that depend on this communication (e.g.: awareness, process coordination or notification). Many of these aspects are critical for the success or failure of a collaborative application.

Acknowledgements. This work was partially supported by the Chilean Science and Technology Fund (FONDECYT), under grants 198-0960 and 100-0870, and also by the Scholarship for Doctoral Thesis Completion of FONDECYT.

References

1. Burridge, R. Java shared data toolkit: user guide. Sun Microsystems, Inc., 1998.
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, S. Pattern-oriented software architecture: a system of patterns. John Wiley & Sons, 1996.
3. Chabert, A., Grossman, E., Jackson, L., Pietrowicz, S., and Seguin, C. Java object-sharing in habanero. Comm. of the ACM 41, 6. 69-76. June, 1998.
4. Ellis, C. A., Gibbs, S., Rein, G. Groupware Some issues and experiences. Communications of the ACM 34, 1. 38-58. January 1991.

5. Fuchs, L., Pankoke-Babatz, U., Prinz, W. Supporting cooperative awareness with local event mechanisms: the GroupDesk system. Procs. of ECSCW'95, (Kluwer Academic Publishers), Stockholm, Sweden. 247-262. Sept.11-15, 1995.
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.
7. Grand, M. Patterns in java. John Wiley & Sons. 1998.
8. Greenberg, S., Roseman, M. Groupware toolkits for synchronous work. Beaudouin-Lafon, ed., Computer-Supported Cooperative Work, Chapt. 6, John Wiley & Sons, 135-168. 1999.
9. Guerrero, L., Fuller, D. A web-based OO platform for the development of multimedia collaborative applications. Decision Support Systems Journal 27, 3. 257-270. 1999.
10. Guerrero, L., Fuller, D. A pattern system for the development of collaborative applications. Information and Software Technology, Elsevier Science B.V., 43, 7, 457-467. May 2001.
11. Information Resource Management. Statewide technical architecture. Chapter 6: application communication middleware architecture. State of North Carolina. Revision July 2001.
12. Jackson, L., Grossman, E. Integration of synchronous and asynchronous collaboration activities, ACM Computing Surveys 31, 2. June 1999.
13. Licea, G., Favela, J. An extensible platform for the development of synchronous groupware. Information and Software Technology, Elsevier Science B.V, 42, 6. 389-406. April 2000.
14. Miranda, H. and Rodrigues, L. Flexible communication suppport for CSCW applications. Procs. of the CRIWG'99, Cancún, Mexico, 338-342. Sept. 21-24, 1999.
15. Pastor, M. VisualTop: a framework to develop groupware systems. Master of Science Thesis. Computer Science Department. Pontif. Universidad Católica de Chile. Nov. 2000.
16. Rhee, I, Cheung, S. Hutto, P. and Sunderan, B. Group communication support for distributed communication systems. Procs. of the 17th Int. Conf. on Distributed Computing Systems, IEEE CS Press. Baltimore, USA. 43-50. May 27-30, 1997.
17. Roseman, M., Greenberg, S. Building groupware with GroupKit. In M. Harrison (Ed.) Tcl/Tk Tools, O'Reilly Press. 535-564. 1997.
18. Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. Pattern-oriented software architecture. Vol..2: Patterns for concurrent and networked objects. J.Wiley & Sons. 2000.
19. Tanenbaum, A. Distributed operating systems. Prentice Hall. 1995.
20. Yi, J., Pastor, E. Communication support for cooperative application in open distributed processing systems. Procs. of CRIWG'96, Puerto Varas, Chile, 61-76. Sept., 25-27, 1996.