CHAPTER 4
DESIGN ANALYSIS

Software and system design will have to cover many aspects. This thesis work will only cover the flexibility and tailor-ability issues of this groupware, while the other design, for example GUI, security, are beyond the scope of this work. The requirements defined in previous chapter are translated into several design steps and sections which will be described in this chapter. The main goal of this chapter is to bring a clear analysis of the system which is going to be developed.

4.1. Distribution Architecture

This system will provide 2 alternative of architecture of its components, namely centralized and hybrid/semi replicated architecture. This distributed system will also allow partitioning of the network. Network partition in this case will be based on client physical locations. The network partition can also be based on the version of this groupware (light version for low resource mobile device, and full version for PC) but that will be a future work.

4.1.1. Network Partitioning

Thus, based on requirements, in one partition of network, there will be one or more clients which the number of client is corresponds to the number of UI and one program replica should be provided. The example of scenario of network partitioning can be seen in Figure 9.

In this sample scenario, Main Server is located in Germany. There are 6 Clients want to join a session. The sequence of request is based on the number on the client names. Another scenario possible is the group leader registers all clients at the same time when requesting a session. Server recognizes location of the client. Then, it suggests partition of the user network by observing the location. It can be

seen in Figure 9 server makes 3 partitions; Region 1, 2 and 3. Server tests the network connection to each client and chooses the best connection in each region as the replicated master. Server asks the chosen clients if they are wiling to be a replicated master. If the chosen client is not willing to be replicated master, server asks another client in the same region to be the replicated master, only if the computer specification supported. If there is none of client in one region is able to be the replicated master, all clients in that region will be clients to a replicated master in the other region (start from the nearest neighbor). It can be the main server itself.

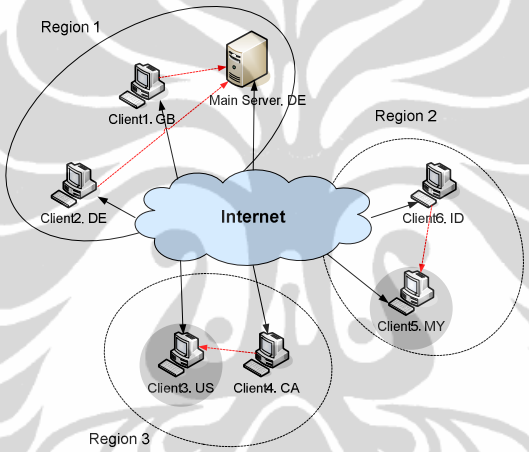

Figure 9. Network Partitioning Scenario

In this sample scenario, Main Server in Germany (DE) becomes a master for Client1 in United Kingdom (GB), and Client2 in Germany (DE). Client5 in Malaysia (MY) becomes a master for Client6 in Indonesia (ID), and Client2 in USA (US) becomes a master for Client4 in Canada (CA).

4.1.2. Late Joiners

When Late Joiners Mode is enabled, there are 3 possibilities of adaptability. The adaptability is based on the network connection to each user and condition of users. How server know this information will be described later. As usual, the system only suggests while the decision always at the user side. Continuing the above example, there is a late joiner, Client7 in Australia (AU) who wants to join

a session. There will be 3 scenarios possible for adaptability of the system regarding this late joiner.

7. Late Joiner uses nearest replicated master. Because the nearest partition is Region 2, Client7 join Region 2 and become a client of Client5, as can be seen in Figure 10.

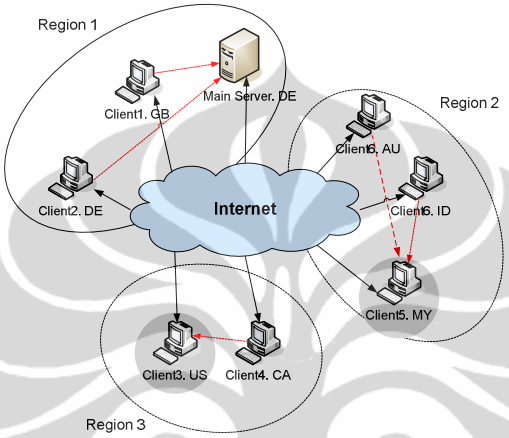

Figure 10. Late Joiner uses nearest replicated master

8. Late Joiner becomes a new replicated master in a new partition. The system suggests Client 7 to make a new partition, because the network condition form AU to every existing replicated master is not good. The illustration can be seen in Figure 11.

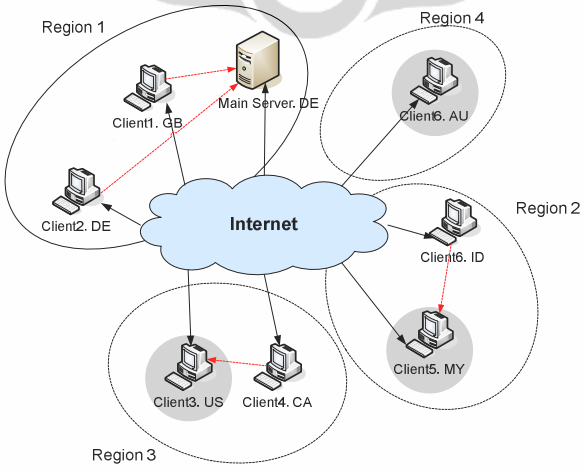

Figure 11. Late Joiner becomes new replicated master in a new partition

9. Late Joiner becomes a new replicated master in existing partition. The system suggest Late Joiner to become a new replicated master in existing region, because the connection of Late Joiner is better than the old replicated master in that region. Thus the network partitioning becomes like Figure 12.

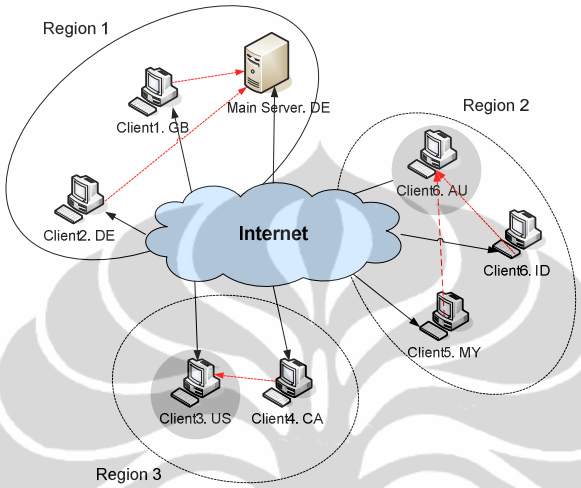

Figure 12. Late Joiner become a new replicated master in existing region

4.2. Role of User

There are 4 roles that are available in a session; moderator, member, floor handler, observer. Take into notice that moderator will have double role at the same time, which is a role as moderator and another role from 3 roles available. Class diagram of Role can be seen in Figure 13.

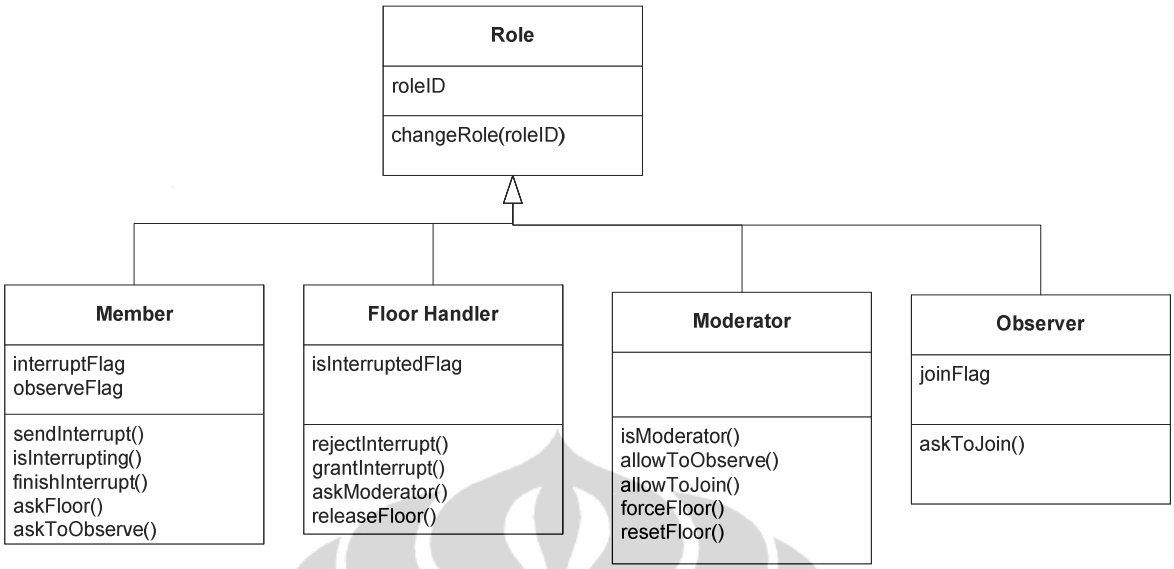

Figure 13. Role Class Diagram

The following are the roles and their corresponding privileges in a session:

- Moderator
  - Choose type of floor control model to be used in the session
  - Force Member to leave a session (kick out)
  - Reset floor list in Sequential Control Model
  - Force to free the token from Floor Holder in Free Control Model
  - Become the first Floor Holder in a session
  - Grant or reject request to become an Observer
  - Grant or reject latecomer
- Member
  - Ask to interrupt **Floor Holder**
  - Ask a token before it is released by the **Floor Holder**
  - Ask to become an **Observer**
  - Leave a session early
  - Join a session late
- **Floor Holder**
  - Can grant or reject interruption
  - Can grant or reject request of token
  - Release token

- **Observer**
  - Can ask to become a **Member** (rejoin a session)

In Figure 14, the state transition diagram of role transformation is illustrated.

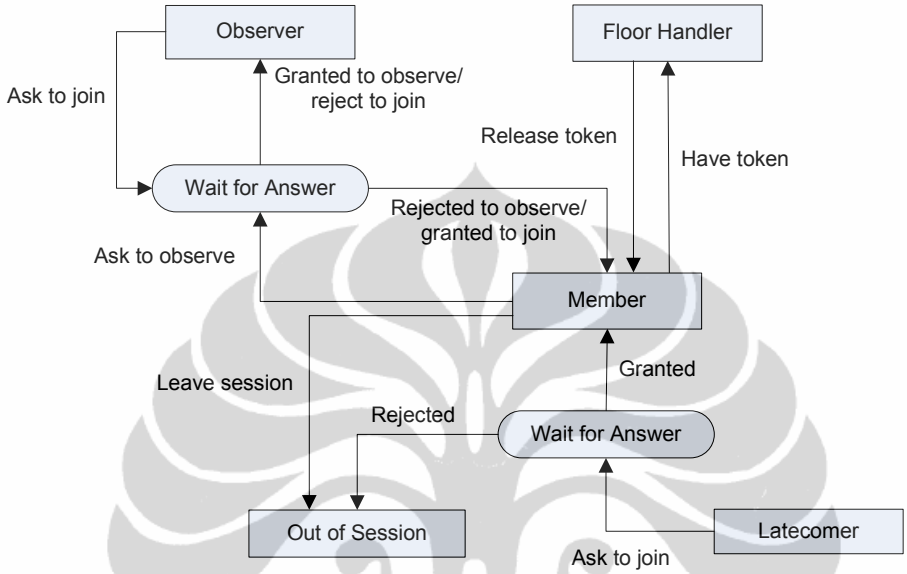

Figure 14. Role State Transition Diagram

4.3. Floor Control Model

At start up, moderator are given the right to choose between three types of floor control available from the system. To bring flexibility in more advanced way, moderator can also change the type of floor between these types anytime when the session is running as shown in Figure 15. The three type of floor control model available are Free Flow, Sequential Flow and Shared Floor Control Model.

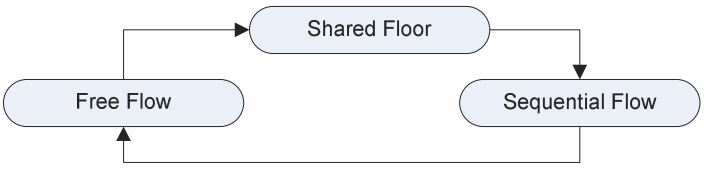

Figure 15. Control Model State Transition Diagram

4.3.1. Sequential Floor Control Model

In sequential flow control, user can access the system only if it is his turn according to the floor list. This floor list can also be analogically thought as token ring network. By default, a Member who is also a Moderator will be the first person in the list. The next sequence in the list is according to the sequence of Member asking for the floor, until all Members in the session is on the list. Notice that Observer will not be in the list, because he can not ask to have the floor unless he is a common Member.

To prevent the floor being possessed too long by the Floor Holder, the Moderator has the privilege to force the Floor Holder to release the floor and allow the next Member in the list to become Floor Holder.

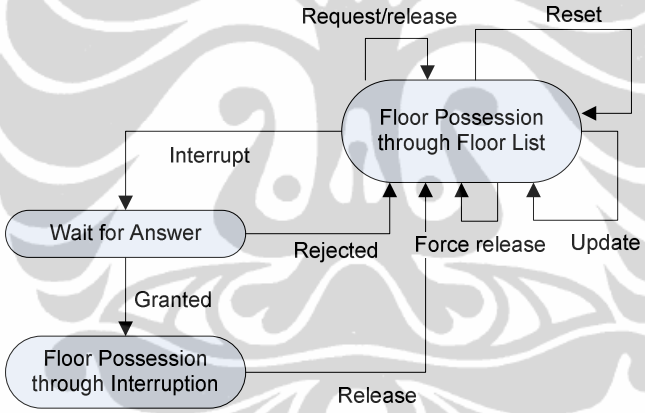

Figure 16. Sequential Flow Control Model State Transition Diagram

Member has privilege to make interruption, and possess the floor even though he is not the actual Floor Holder. But the actual Floor Holder has the right to grant or reject the request of this interruption. After this Member finish his interruption, the floor is passed to the actual Floor Holder and continue the floor possession based on floor list. Figure 16 illustrates this model.

Whenever there is a change in the number of Member, the floor list will be updated, but not being reset. The change of number of Member in the session can be caused by early leaver, latecomer, a Member becomes Observer, or an Observer becomes a Member. The new Member should ask the floor to be

updated in the floor list. If a Member leave, either he become an Observer or leave the session, his place on the list is deleted.

Whenever Moderator thinks that it is necessary, he has the privilege to reset the floor list. In this case, the list will be empty. The mechanism to fill the floor list is based on the sequence of request of the floor.

This model is best suited for all active users session, for example in brainstorming phase.

4.3.2. Free Flow Control Model

In this flow control, the system provides 1 token for all user on the session to possess a floor, or in other words to change his role from Member to Floor Holder.

To prevent the token being possessed too long by one Member, the Moderator has the privilege to force the Floor Holder to release the token. Member has privilege to ask the actual Floor Holder to pass the token, but Floor Holder can grant or reject his request. Figure 17 illustrates this model. Regarding the latecomer or early leaver, this flow control will not face a problem like in the previous model, since the token is independent. Update is not needed like in Sequential Flow Control Model.

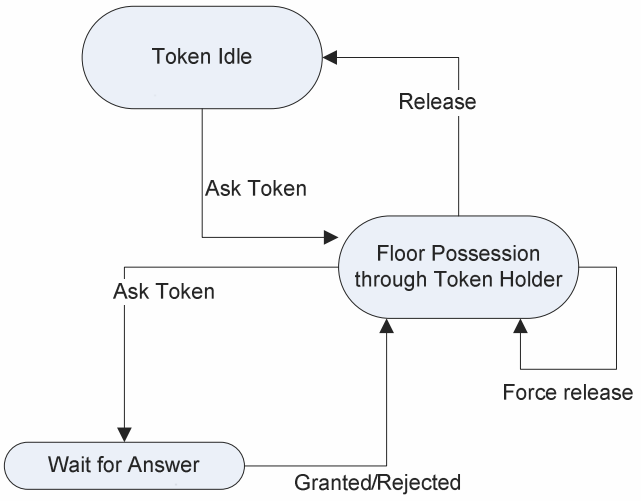

Figure 17. Free Flow Control Model State Transition Diagram

This flow control model is appropriate for presentation-like session, where only one person holds a session for a long time.

4.3.3. Shared Floor Control Model

In this model, there is no different role. All users are members and have the same privileges. In this flow control model, rigorous 2-phase locking, which will be explained later, is applied. Shared floor control model leads to a pure distributed system. It may be suitable for some applications, but not all. One example is as example Electronic Brainstorming System (EBS).

The main idea of this floor control model is to provide comfortability to users to express their ideas in a session, without having to wait someone else to finish his work. It will also save the session time. But trade off should be taken into account, that a more complex locking and updating mechanism should be provided.

4.4. Component

In component based software, applications are developed from components. These components themselves might be built up from several small components. Components might be added or removed during runtime. This solved the tailor-ability issue that we have. Since we use hybrid-replication architecture, a replication mechanism of the components, should also be determined.

4.4.1. Separation of Components

To support flexibility, in this system we will separate objects or components of the software based on its functionality into program, data, and user interface objects. Remember that components can be built up from other components.

Therefore, in a session there will be some clients, some program replica, and some data objects as illustrated in Figure 18.

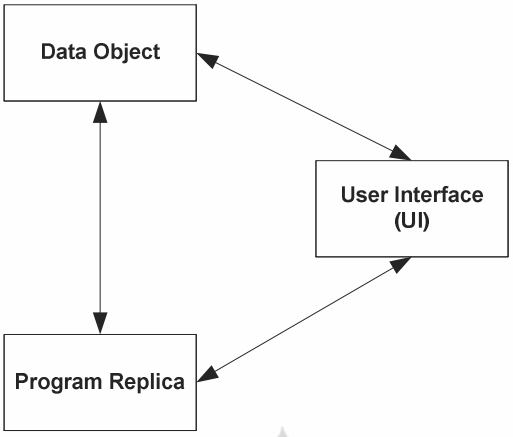

Figure 18. Separation of Components

The definitions of each object are described below:

10. **User Interface** (**UI**) is part of the program which provides interface to the main program. It is translated as the GUI of the program. UI is the medium for the user to interact with the program and the data object

11. **Program replica** is the copy of program, the brain, the processor with the computing and executing ability. Program replica has the ability to interact with the data object.

12. **Data Object** is part of the program where the information stored. It can be translated as a functional part of the program or as a part of the artifact.

4.4.2. Extensibility of Components

Components can be changed, extended or curtailed, during runtime. The changes however should not disrupt the application and it should be transparent to the user. The extensibility patterns of components proposed by Hummes and Merialdo [11] enable the extensibility to be encapsulated in one component. The extensibility pattern is intended to be used to provide a default behavior which can be changed at run time [11]. The application will only see the specific behavior of a Proxy class. The class diagram to illustrate it can be seen in Figure 19. The illustration of this pattern is slightly modified from the real one proposed by Hummes and Merialdo [11], to adopt the later design in this work.

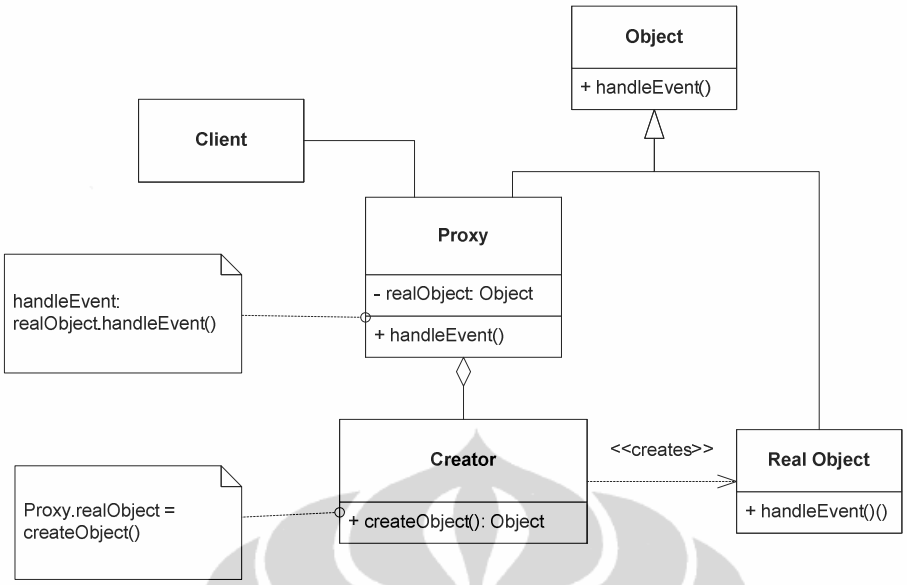

Figure 19. Class Diagram of Extensibility Patterns

According to this pattern, Proxy extends the interface of an Object, which may be inserted at run time. Inside this Proxy, exists a Creator which is responsible to create a new object of an arbitrary class Real Object corresponds with the interface Object. This pattern is combination of the Proxy and the Factory Method patterns.

The design is adapted from this pattern. Client will always communicate to the object through the Proxy. At start up, Creator will automatically instantiate an Object, a Default Object, and pass the reference of the Default Real Object to the Proxy. Any event that Proxy receives will be delegated to the Default Object. When the creator receive an event to create a new real subject, it instantiates the respective class and sets the reference in the Proxy to the newly created New Real Object. The Proxy from now, forward all subsequent events to this object, unless the Creator changes the reference to the Default Real Object again. The interaction between objects is illustrated in the sequence diagram in Figure 20.

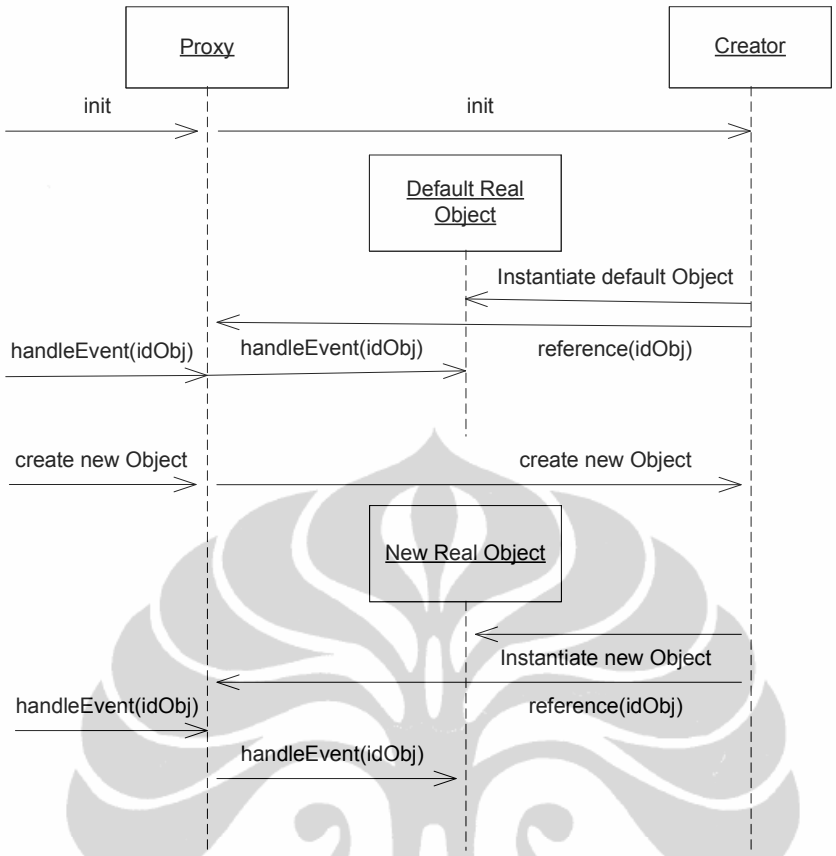

Figure 20. Sequence Diagram of Extensibility Pattern

Following the creation of a new Object, Proxy can delete the existing Real Object or simply create the new Real Object while storing the old ones. The incoming events are then forwarded to corresponding Real Objects. This variation will be best used if the newly added Real Object has a different functionality than the old ones.

4.5. Concurrency Control

Concurrency control deals with the issues involved with allowing multiple people simultaneously access to shared entities, be they objects, data records, or some other representation [www5]. The goal of concurrency control is to allow several transactions to be executed simultaneously, but in such a way that the collection of data items (e.g. files or database records) being manipulated, is left to consistent state. This consistency is achieved by giving transactions access to data

items in a specific order whereby the final result is as the same as if all transactions had run sequentially.

Parallel operations on a shared data are possible to bring interference to each other. Possible problems that might occur are [www3] :

- Lost update. This problem occurs when overwrites uncommitted data.
- Inconsistency retrieval. This problem occurs when reading uncommitted data.
- Unrepeatable read. This problem occurs when reading one object twice with two different results.
- Incorrect summary problem. This problem occurs if one transaction is calculating an aggregate summary function on a number of records that are being updated at the same time by a second transaction.

To prevent those problems, in this system we will use locking and updating mechanism. Locking will be implemented for the role **Floor Holder**. While updating mechanism is implemented to all users.

4.5.1. Logging Mechanism

In this system we will need a dynamic replication mechanism, since this system use a hybrid-replicated architecture. The approach proposed in this thesis is by using logging mechanism. Logger and logable mechanism is based on mechanism proposed by [1]. But in our scenario, to implement logging mechanism, there will be two dedicated component which functioning as logger; History Logger (HL), and Temporary Logger (TL). The histories of all committed events are stored in the HL. While the events before a user who has the lock decided to commit or abort are stored in TL. Logger is a component that belongs to the program. Therefore it stays with the client who acts as replicated master. Synchronization and updating protocol should be implemented in the logger. Therefore, there is no need to synchronize all shared data objects, but only the logger. If all consequent

states in the logger are synchronized, then the data objects will also be synchronized.

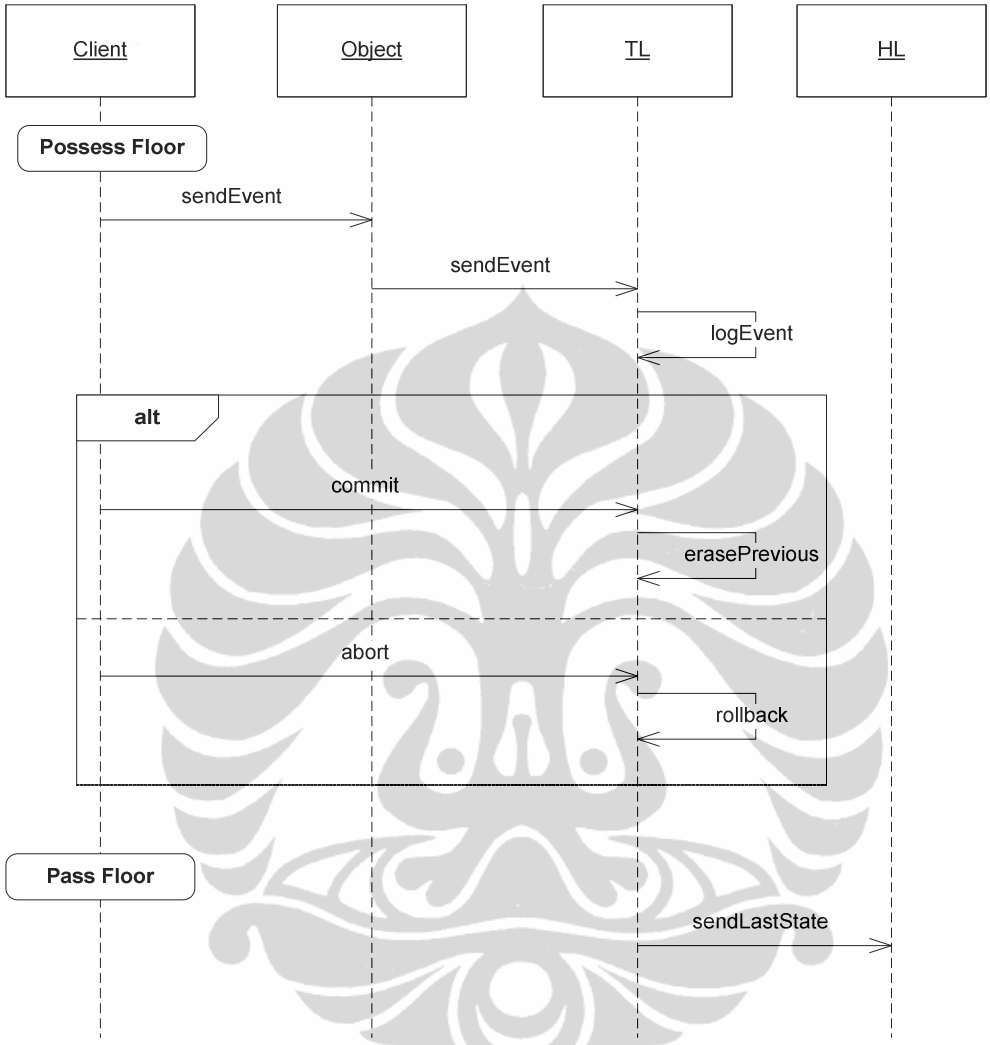

Figure 21. Sequence Diagram for Logging Mechanism

Every event will be stored temporarily in the TL until the user who holds the lock commit or abort those events. If the user decides to abort all subsequent events, then a rollback mechanism should be performed. The rollback mechanism is simply done by erasing all events in TL, and use the last state in HL. When the user decides to commit the subsequent events, the logger simply copy the last state in TL to HL and erases all events in TL.

4.5.2. Updating Mechanism

In this proposal, a logger is not only store the data, but also implements synchronization and updating protocol. Therefore, loggers will actively communicate with each other.

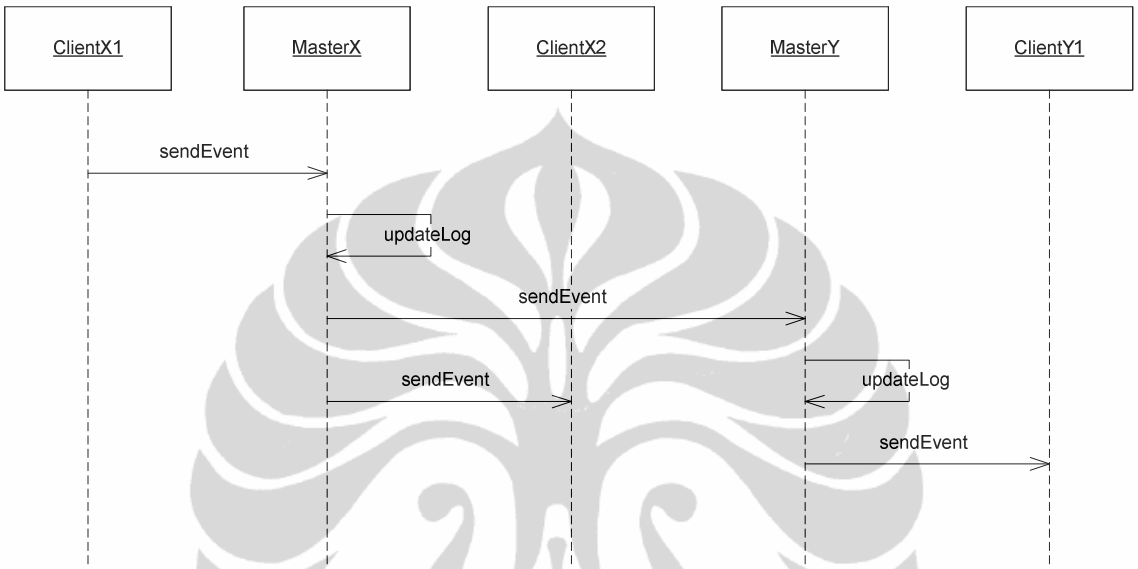

Figure 22. Sequence diagram for Updating Mechanism

This mechanism is also applied when generating components. Through a proxy, client command the proxy to generate objects, as explained in Figure 20. Local proxy will communicate to the other proxy with the same mechanism as Figure 22.

4.5.3. Locking Mechanism

There are 4 terms of locking mechanism used in distributed systems; lock, commit, abort, and unlock as defined in [25].

In a pure distributed system locking mechanism should be decided carefully. In a pure distributed system scenario, every task can have the same privilege to work on some shared objects at the same time. In groupware, it can be translated to the condition where every member in a session can work on the same artifact at the exactly same time. For example, in a design session by using CAD tools, every

member may have his own cursor and modify the drawing at the same time as illustrated in Figure 23. Another example, in electronic Brainstorming System (EBS), every member might have possibility to write his ideas simultaneously, without having to wait the floor or token.

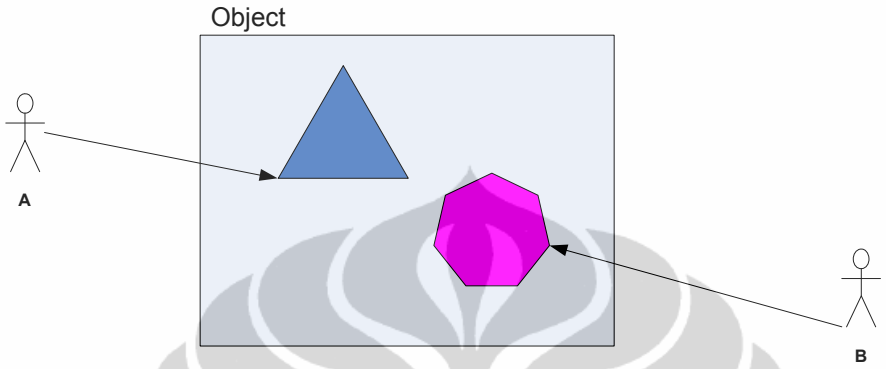

Figure 23. Pure distributed system

Thus, it leads to a situation where more than one task change an object and therefore possible problems mentioned above might be happened. To provide a pure distributed system, an expensive resource should be given.

In our scenario, when a user changes his role from Member to Floor Holder, automatically he will get a lock. Locking mechanism begins by the time the floor hands to him. There will be no further problem of locking in sequential and free floor control model, since there is one floor holder at a time. There is a single locking which the floor holder automatically receives. The groupware provide a commit and abort button that can be chosen by the floor holder either during his possession to the floor or at the end of his possession. System will prompt floor holder if he choose neither commit or abort at the end.

But when shared floor control model is used, locking should be applied carefully. Inside an application, there are some objects. Each user will only lock one object at a time. By the time he click an object, locking is applied, thus no one else can modified his possessed object. After he finished working on an object, he should choose either to commit or to abort. If for some period of time, he does not make any decision, the system will prompt him to make a decision.

This mechanism is only applied when there is many separated objects inside the application. Thus it can be applied to EBS, CAD tool, but not an application like Text Editor, or Presentation Display.

4.6. Implementation tools

The development of this groupware is designed to be implemented by using JavaBeans [www2], which based on Java [www1] language. As well as Java, JavaBeans implement the basic rules "Write Once, Run Anywhere™".

In our groupware, JavaBean components will be used as building blocks in composing applications. Such builder tool will be used to connect together and customize a set of JavaBean components to act as an application. But it does not necessarily that all programming are written in the shape of Beans (component). Classes, which are best written in modules, will be written in Java classes.

The communication of distributed components will be using the advantages of CORBA. Java is now supporting CORBA IDL. Thus, it is solving the problem that a basic JavaBeans could not support remote interaction in distributed applications. The other reason of choosing JavaBeans is because it supports communications to JDBC, Java RMI, as well as CORBA server.

Typical functionalities of JavaBeans that we will take advantages are its support for:

- Introspection
- Customization
- Events
- Properties
- Persistence

The three most important features of a Java Bean are:

- The set of *properties* it exposes
- The set of *methods* it allows other components to call
- The set of *events* it fires.

JavaBeans provide class and interface discovery as a mechanism to locate a component at runtime and to determine its supported interfaces so that these interfaces can be used by others. The component model also provides a registration process for components to make itself and its interfaces known. Dynamic (or late) binding allows components and applications to be developed independently. An application does not have to include a component in the development process in order to use it at runtime; it only needs to know what a component is capable of doing. Dynamic discovery also allows developers to update components without having to rebuild the application that use them. This discovery process can also be used in design-time environment. The development tool may be able to locate a component and make it available for use by the designer.