Gclipse

A Collaborative Editor Plug-In for
Eclipse

Marco Cicolini

Master's Thesis

August 3, 2004 –February 2, 2005

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistant: Keno Albrecht

Preface

Abstract

This thesis is about a collaborative text editor, Gclipse. The editor is implemented in Java as an Eclipse plug-in. Eclipse is an application development framework for Java that also provides a Java development environment. Gclipse extends this environment by adding the feature of collaborative editing additionally to the inherited abilities, such as syntax highlighting, code formatting and code completion.

Document consistency and high responsiveness are key characteristics in the field of collaborative editing. To achieve these requirements, first, Gclipse uses the approach of Operational Transformation to ensure document consistency even if different people modify them at the same time. Additionally, by executing local operations immediately, high responsiveness is guaranteed.

The editor is based on JGroups, a Java group communication framework. JGroups offers the notion of groups, which represent a set of connected hosts sharing the same document. In such group communication, reliable broadcast and group membership protocols is provided.

Gclipse shows that, based on Operational Transformation, a collaborative editor is achieving high responsiveness and document consistency. Gclipse is a prototype, but it is a first step towards a collaborative software development environment.

Acknowledgment

With this thesis I complete my degree in Computer Science. I would like to thank all the people who supported me during this long time of study at the ETH. Especially, I would like thank my family and my friends for supporting me. I spent a lot of time in learning and studying and particularly the last six months in writing this thesis. Therefore, some people missed me out, I would like to apologize for that. Since I have some spare time now I will try to catch this up.

I would like to thank the whole DCG group who supported me with good advices.

I would most notably thank Keno Albrecht, my supervisor, for supporting me the whole time. We had a lot of good discussions and ideas. Some of them were used, some not. He was always ready to answer my question helping me out when i had problem.

I would also like to thank Simon Schlachter he supported me with good ideas for the implementation and had always time for some debug sessions.

I also want to thank Roger Wattenhofer for giving me the possibility to write this interesting and demanding thesis.

Contents

1 Introduction

Extreme Programming is a new trend in the software development process. The benefit of this technique is higher quality software which is less error-prone and much more maintainable. One aspect of *Extreme Programming* is *Pair Programming*. Using this technique, two developers sit together in front of one computer, one keyboard, one mouse and one display. One of the developers is writing source code while the other one is checking the new code for errors and improvements. During this process the developers discuss current problems to find improved solutions. One disadvantage of *Pair Programming* is the fact that one developer is blocked.That's where Gclipse comes into play.

Gclipse is an editor which supports collaborative editing. With such a tool the two developers can sit apart but still do the same work. This opens up additional possibilities for the develpmoent process. One developer could write comment the other one program code. Alternative one developer observes the process and corrects mistakes on the fly. Oppositional to *Pair Programming* more than only two person can take part in the development process. This could also be done with *Pair Programming* but the place in front of one computer is limited. Finally a group editor also supports users working from different places all over the world. Both, space and place, are obviously no limitations to the group editor. It would have been nice to have written this documentation with a group editor so a proof reader could already have started correcting it.

Thinking of the open source community Gclipse could bring all these features to it. Integrating Gclipse into a software development environment would partially replace revision control systems. Additionally Gclipse would bring the *Pair Programming* paradigm to everybody who wants to try it out.

Why another editor? Gclipse provides the feature of collaborative editing. Collaborative writing demands document consistency and high responsiveness. To achieve these requirements it uses the approach of *Operation Transformation*. In to traditional consistency systems *Operational Transformation* achieves the requirements without locking.

Chaper 2 introduces the field of *Computer Supported Collaborative Work* (CSCW). Problems which occur in such system system and solutions to solve them are discussed. OT is introduced.

Chapter 3 brings the application development framework *Eclipse* in. Gclipse is integrated into *Eclipse*. This provides development features to Gclipse.

Chapter 4 describes the development process of Gclipse. Additionally *JGroups* is introduced. JGroups is group communication framework for Java. JGroups is used for the communication between the hosts running Gclipse. It offers membership and reliable broadcast options. The whole design of Gclipse, problems and their solutions, are explained.

Chapter 5 summarizes the whole work done during the master thesis. It again highlights important parts, solutions and encountered problems.

Chapter 6 gives an outlook on possible additions to Gclipse. Features to increase the usability and the efficiency while working with Gclipse.

2 Computer Supported Cooperative Work

This chapter introduces Computer Supported Cooperative Work (CSCW). First of all, an informal definition is given of what a groupware system is and what it is used for. After that, the idea of collaborative work is explained. Finally, difficulties and possible solutions to handle them will be discussed.

2.1 Introduction

On the one hand, there is computer supported work that shows up wherever a person works with a computer to accomplish a task. On the other hand, of collaborative work is spoken when a group of people work together to solve a task. Connecting these two parts opens the area of CSCW.

There are a lot of examples of computer supported collaborative work in the world, such as the Wikipedia project [3], a public encyclopaedia, which is written by everybody who wants to participate. Another example are group communication tools like email or instant messaging. In a more technical environment, there are so called Source Management Systems (SCM) which help to organize different kinds of resources. Version or Revision control systems like CVS or Subversion represent SCMs to control source code in a software development process. The meaning of CSCW is when a group of people need to achieve a goal with the help of computers.

2.2 Concurrency Control

Figure 2.1 shows a possible scenario what happens when two users work on the same document. The result in this case are two different words which is unintentional. The words should be the same for every user. This is a consistency problem and comes from the operation execution order. User 1 first inserts the character *N* and then *E*. User 2 does the other way round. This obviously leads to two different words *KENO* and *KNEO*. The desired result would be either *KENO* or *KNEO* in both documents. This problem introduces the area of consistency control. The task of consistency control is to guarantee the same states on both sides. There are a lot of different approaches to solve this problems.

Optimistic This type of concurrency control allows inconsistent state and therefore provides mechanisms for human users to resolve them. This is a good approach if inconsistencies are rare. CVS, offers this kind of concurrency control. If the system cannot resolve a conflict by itself it ask the user how to solve the inconsistency. One big advantage of this system is the high responsiveness on the user side.

Pessimistic This type of concurrency control does not allow inconsistencies as the optimistic approach. It ensures that the state of the system never converges to an inconsistent state. This means whenever the system is about to be changed, the system has to guarantee that after the change it is still consistent. To do so pessimistic concurrency control can be divided into two different categories, centralized and decentralized.

Centralized In this approach of pessimistic concurrency control, a centralized instance takes care of the concurrency control. Usually, there is some kind of central locking mechanism. For example, a server granting read and/or write access to a site, preventing other sites from reading/writing as well.

Decentralized In contrast of the centralized approach, decentralized pessimistic concurrency control uses decentralized mechanisms to guarantee consistent states. Examples are distributed locking mechanisms or voting techniques.

As mentioned before, the different approaches have different advantages and disadvantages. However, the more is known about a system the more specific a concurrency control system can be chosen. Therefore, some conditions have to hold for implementing a distributed group editor.

Responsiveness An editor should be as responsive as possible. If the user needs to wait too long after every input, the editor is not usable. While typing, the editor should behave the same as a single user editor. Therefore, responsiveness is very important.

Consistency The documents need to be consistent. Otherwise, the idea of a group editor is senseless. However, it is not needed, that every time a change is done the document has to look the same. So no strict consistency is needed. The local changes should be done as fast as possible but remote changes might be done later in time, to achieve good responsiveness. So if all changes are executed and the system is in a quiet state the documents became eventually consistent.

Intention Preservation This means to ensure that the local operation effect observed by user is the same as its remote effect observed by other users in the face of concurrency.

Causality Preservation Causality preservation means to ensure that user actions are always performed in their natural cause-effect order during a session.

According to these constraints the concurrency control can be chosen. Due to the requirement of high responsiveness, every system using locks is dropped because locking blocks sites for an unknown long time. Due to consistency the optimistic approach is also dropped, because of the user intervention. This decreases the responsiveness for other users. Also the change done by the user need to be propagated to the other sites.

Figure 2.1 shows a simple example what happens when two users insert a character at the same time. The results are two different words which violates the consistency constraint. Now two possible ideas for concurrency control are discussed regarding an implementation of an editor and the constraints.

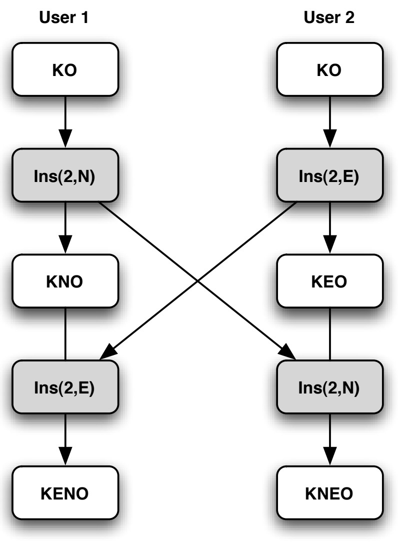


Figure 2.1: *This figure shows what happens, when two user concurrent insert a character at the same place in a document.*

2.2.1 Operational Transformation

This approach for consistency control was introduce by Ellis and Gibbs in 1989 [6] and was refined by several other scientists [7], [10], [11]. The idea of this approach is to transform all incoming operations so they have the same effect on every document even the documents on which the operations are executed don't look the same. To achieve this so called operational transformation functions are used. These functions transform operations if the operations are concurrent. To check if two operations are concurrent a vector time stamp is used. The proof that this approach is correct was done in 2004 by Imine et. al [7]. They also proved the correctness of their proposed functions. A lot of functions proposed early were proven to be incorrect.

Introduction

The Operational Transformation (OT) approach defines a request as a tuple (u, v, o) where u is the user issuing the operation, v is the vector time stamp and o is the operation to be executed on the document. The execution of an operation is the insertion of the operation content into the document. In the OT approach the documents on all host have a different document state denoted by the vector time stamp v. When an operation op_1 is issued the local time stamp is attached to it before its execution. Then this operation is distributed to other users. Other users have different document states. The new operation is transformed against already executed operations resulting in a new operation. The adOPTed algorithm proposed in [10] describes how this transformation works.

The adOPTed Algorithm

To use the transformation functions an algorithm which transforms and executes them is needed. As mentioned, the adOPTed algorithm is used. A detailed description and explanation of the *adOPTed* algorithm, which is employed in Gclipse, can be found in [10]. The algorithm ensures the causal execution order of the operations. So remote operations might be delayed for local operations this cannot happen. While typing the local causal order is defined immediately, because every change in the document is executed immediately. The algorithm keeps a history of all received operations before execution, which is needed to execute the transformation functions.

Transformation Functions

This approach uses so called *operational transformation functions*. These functions transform operations against the state of a document, that when the operation is executed, has the same effect as the execution on the host which initiated the operation. A transformation of two operations is written as $T(op_1, op_2)$, whereas T is one of the transformation functions defined later. The meaning is, op_1 is transformed against op_2 that implies that op_2 was already executed. The functions must meet the following two conditions also know as the *Convergence Properties.*

- The first condition, C_1, defines a state identity. A document state generated by first executing op_1 and then the transformed operation $T(op_2, op_1)$ must be the same as if the state was generated by first executing op_2 and then the transformed operation $T(op_1, op_2)$. C_1 is defined as follows:

$$C_1 : \quad [op_1; T(op_2, op_1)] \equiv [op_2; T(op_1, op_2)]$$

- As there are usually more then only two hosts condition C_1 is necessary but not sufficient. Therefore condition C_2 ensures that an operation transformed against a sequence of concurrent operations does not depend on the order in which the operations are transformed.

$$C_2 : \quad T^*(op_3, [op_1; T(op_2, op_1)]) = T^*(op_3, [op_2; T(op_2, op_2)])$$

where T^* is the definition of transformation one operation according a sequence of operations.

$$\begin{aligned} T^*(op, []) &= op \\ T^*(op, [op_1; op_2; \ldots; op_n]) &= T^*(T(op_1, op_2), [op_2; \ldots; op_n]) \end{aligned}$$

The prove that the conditions C_1 and C_2 are sufficient to ensure the convergence property for any number of concurrent operations is done in [10] and [9]. To use the properties the operations need to be defined. The operations differ depending on the type of the object which has to be consistent. In the case of the group editor it is a simple text document. Therefore the operations *Insert* and *Delete* are defined.

Insert $Ins(p, c, w)$ Defines the operation for inserting a character c at the position p. The parameter w is used for the transformation functions to keep track of the insertion positions and what is described in more detail in [7].

Delete $Del(p)$ Defines the operation for deleting a character at position p. The deletion does not need the additional parameters c and w.

Nop $Nop()$ Defines the empty operation, which has no effect.

The transformation functions are defined for every combination of all defined operations. In this case, four functions are needed: (insert → insert), (insert → delete), (delete → insert) and (delete → delete). The functions are defined as follows.

```
T(Ins(p1,c1,w1), Ins(p2,c2,w2)) =
    α1 = PW(Ins(p1,c1,w1))
    α2 = PW(Ins(p2,c2,w2))
    if (α1 < α2 or (α1 = α2 and C(c1) < C(c2)))
        return Ins(p1,c1,w1)
    else if (α1 > α2 or (α1 = α2 and C(c1) > C(c2)))
        return Ins(p1 + 1,c1,p1w1)
    else
        return Nop(Ins(p1,c1,w1))
```

```
T(Ins(p1,c1,w1), Del(p2)) =
    if (p1 < p2)
        return Ins(p1,c1,w1)
    if (p1 > p2)
        return Ins(p1 -1,c1,w1)
    else
        return Ins(p1,c1,p1w1)
```

```
T(Del(p1), Ins(p2,c2,w2)) =
    if (p1 < p2)
        return Del(p)
    if (p1 > p2)
        return Del(p1 + 1)
```

```
T(Del(p1), Del(p2) =
    if (p1 < p2)
        return Del(p)
    if (p1 > p2)
        return Del(p1 - 1)
    else
        return Nop(Del(p1))
```

Figure 2.2: *These are the four functions to transform the operations. For every combination of operations a function is needed. There are 4 functions missing, the ones containing the* Nop *operation. This operation has no effect on the document. When an* insert *or* delete *operation is transformed against a* Nop *operation the result is the unchanged* insert *or* delete *operation. The other way round the result is a* Nop *operation.*

In the first transformation, the PW function is used which is defined as follows:

$$\mathrm{PW}(\mathrm{Ins}(p, c, w)) = \begin{cases} p & \text{if } w = \epsilon \\ pw & \text{if } w \neq \epsilon \text{ and } (p = \mathrm{Current}(w) \text{ or} \\ & p = \mathrm{Current}(w) \pm 1) \\ \epsilon & \text{otherwise} \end{cases}$$

To understand the *PW* and the *transformation functions* the *PWord* is introduced. The *PWord* is a vector of numbers which is denoted by w. This vector keeps track of the insert position of an operation before its transformation, because otherwise this information would be lost, for a more detailed description see [7]. The *PW* function, defined only on insert operations, returns the PWord of the given insert operation. pw denotes the concatenation of the position p in front of the PWord w. *PW* uses the function *Current(w)* which is the first element of w.

Examples

To see how these functions work two examples are given, the correction of Figure 2.1 and a more complex insertion.

The first example is shown in Figure 2.3. Two users insert one character into the initial string *KO*. They both want the same string as result. After User 1 inserted the character N at position 2 it gets the string KNO. User 2 inserts the character E at position 2, resulting in the string KEO. Both user then send their operation to the other user. User 1 inserts op_2 from user two by first transforming it against operation op_1 which results in the operation op_2' which in this case is the same as the original op_2. User 1 gets to string KENO which is the desired one. User 2 receives op_1 which before inserting needs to be transformed against op_2 resulting in op_1'. According to the transformation functions the position of op_1 was shifted by one. So User 2 inserts the character N at position 3 resulting in the same string as the one from User 1 KENO. To mention is that also the string *KNEO* would have been correct as long as the two users get the same string.

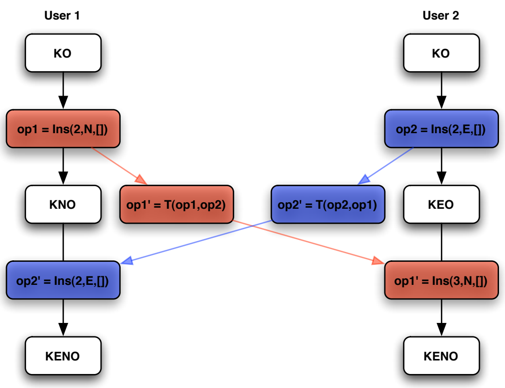


Figure 2.3: *This figure shows an example of an insertion using OT. It is the correction of the concurrency problem example in Figure 2.1*

The example in Figure 2.4 shows a more complex example of insertions. This example is called the C_2 *puzzle*, because lots of proposed OT functions don't solve this problem due to a violation of the C_2 condition. The interested reader can follow the example and try to do the transformations by himself. It's just applying the transformation functions to the received operations.

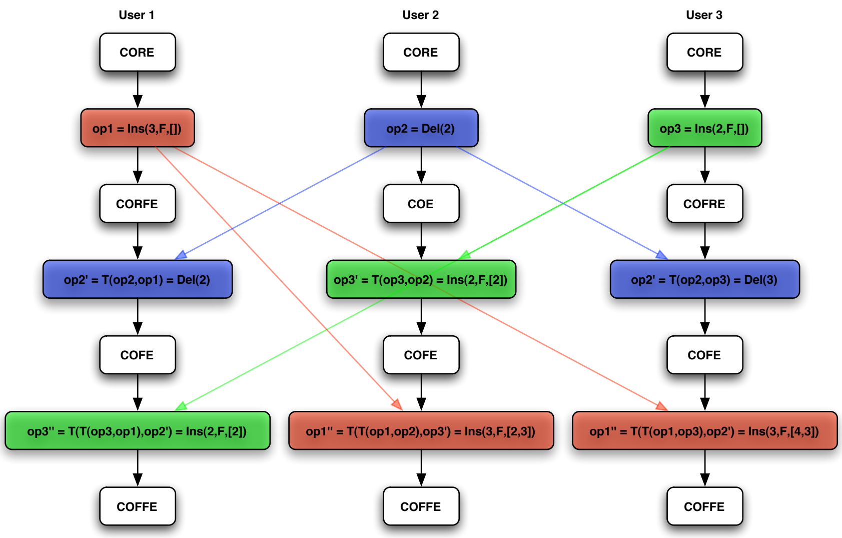

User 1
User 2
User 3
CORE
CORE
CORE
op1 = Ins(3,F,[])
op2 = Del(2)
op3 = Ins(2,F,[])
CORFE
COE
COFRE
op2' = T(op2,op1) = Del(2)
op3' = T(op3,op2) = Ins(2,F,[2])
op2' = T(op2,op3) = Del(3)
COFE
COFE
COFE
op3'' = T(T(op3,op1),op2') = Ins(2,F,[2])
op1'' = T(T(op1,op2),op3') = Ins(3,F,[2,3])
op1'' = T(T(op1,op3),op2') = Ins(3,F,[4,3])
COFFE
COFFE
COFFE

Figure 2.4: *This figure shows the C_2 puzzle. It is a scenario to check if the C_2 condition is met.*

2.2.2 Time Stamp Ordering

To achieve concurrency control a deterministic order on operations is needed. Whenever two operations are concurrent, meaning they were issued at the same time, it is not clear which one is executed first. For this case, a technique to guarantee the same order on every host is needed. In this approach, this order is guaranteed by time stamps. This time stamps are, in contrast to those from OT, not logical but are taken from the local clock. Therefore, the clock on all hosts need to be synchronized. This could be done by some kind of clock synchronization algorithm like the *Network Time Protocol* could be used.

The storage of the document also differs from that one in OT. In this case, a linked list is used and not an array. Every node in this list contains the character, a unique ID, a flag to declare if the node is deleted or not and a time stamp. The operations executed on the list are *Insert*, a new node is inserted, and *Delete*, a node is removed from the list.

Delete

The delete operation is quite simple. When a node is declared as deleted it is not shown anymore in the user interface, but it is still in the list. It might not be that obvious why the deleted nodes cannot be removed from the list. Assuming User 1 receives the delete operation from User 3, but User 2 does not. User 1 removes the deleted node, in the meantime User 2 is inserting another node after the deleted Node and sends this operation to User 1. User 1 already deleted the node after which the new node should be inserted. Thats way the deleted nodes cannot be removed. This results in an always growing list. But when the system is in a steady state and all sites agree on the same state they could agree on removing deleted nodes, so eventually the list size, can be decreased. Agreement can be achieved by some consensus algorithm, which is a well known problem.

Insert

The insert operation inserts a new node after an existing node in the list. Every node has a unique id. This id consists of the site user or host name and a sequence number that is increased whenever a site adds a new node. This sequence number is different for all sites, it can be considered as a local logical time stamp. If two nodes need to be inserted after the same id, the time stamp is used to decide which operation happened before the other and is inserted first. Using these time stamps, a total order on all sites is achieved. For the case the time stamps are identical the lower id decides which node is inserted first.

Two operations are declared as concurrent if they are inserted after the same node. Therefore every node keeps a list of all nodes which initially were inserted after that node. If this list is empty a new node is inserted directly after that node. The history list sorted according the time stamps of the nodes. The new node is then inserted into this list according its time stamp. The history list is needed because messages carrying operations might be delayed so they arrive later even if they were invoked earlier than another one. So every node has 3 pointers showing to other nodes. The first one, the *list pointer*, is the one for the linked list itself, the second one, the *history start pointed*, denotes the starting point of that nodes history list and the last one, the *history forwarder pointer*, is for keeping track of a history list of another node. Figure 2.6 gives an example of such a linked list.

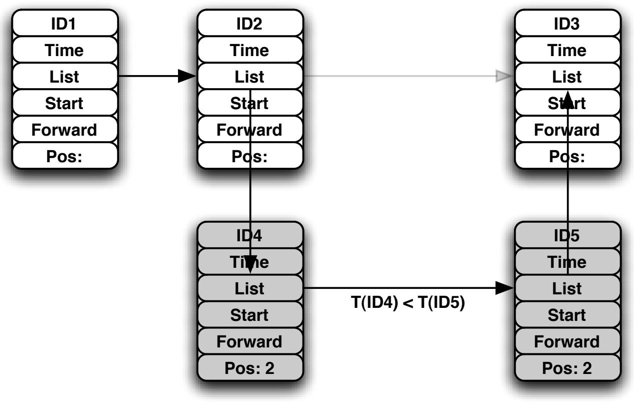


Figure 2.5: *This figure shows a simple insertion of two nodes with different time stamps. The upper three white nodes were the initial list and the two grey lower nodes were inserted after node 2. Node 4 is inserted before node 5 because of the lower time stamp of node 4.*

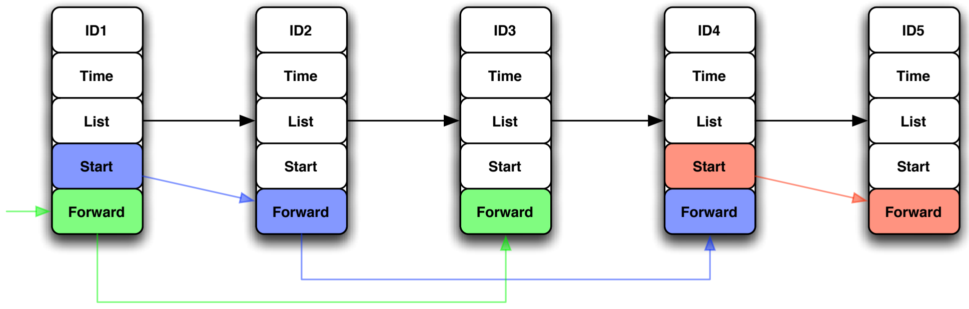


Figure 2.6: *This figure shows how the linked list looks like. Each color defines a history list, the green one is from a node not in the list, the red one from node 4 and the blue list is from node 1. The start pointers of the nodes 2, 3 and 5 point nowhere meaning that no other node was inserted after them.*

Examples

A simple insertion is illustrated and explained in Figure 2.5. A more complex insertion is shown in Figures 2.7 and 2.8 to explain how the history list is used:

- At the beginning, the list consists of three elements with the ids 1, 2 and 3. The nodes with id 4, 5, 6 and 7 need to be inserted, Node 4, 5, 7 after node 2 and Node 6 after 5 (Figure 2.7a). The insertions are done in the order, 4, 5, 6, 7. This is the reception order of the messages containing these insert operations on the node executing the insertions.
- Node 4 is inserted after Node 2, the start and the list pointer of Node 2 are set to Node 4. The list pointer of Node 4 is set to Node 3 (Figure 2.7b).
- Node 5 is the next node to insert. Since the start pointer of Node 2 is not empty, the history list of Node 2 need to be traversed. And Node 5 is inserted before Node 4 because of the lower time stamp of Node 5 compared to the one from Node 4. The start and the list pointer from Node 2 are set to Node 5. The forward and the list pointer of Node 5 are set to Node 4 (Figure 2.7c).
- Next, Node 6 is inserted after node 5 since the start pointer of Node 5 is empty. Node 6 is directly inserted after Node 5. The start and the list pointer of Node 5 are set to Node 6. The list pointer of Node 6 is set to Node 4 (Figure 2.8a).
- Finally, Node 7 is inserted after Node 2 whose start point references Node 5. For Node 7, the history list of Node 2 needs to be traversed. Assuming that Node 7 has the highest time stamp of all nodes in that list it is inserted after Node 4. So the forward and the list pointer of Node 4 is set to Node 7 and the list pointer of Node 7 is set to Node 3 (Figure 2.8b).
- The resulting state of the linked list is illustrated in Figure 2.8c.

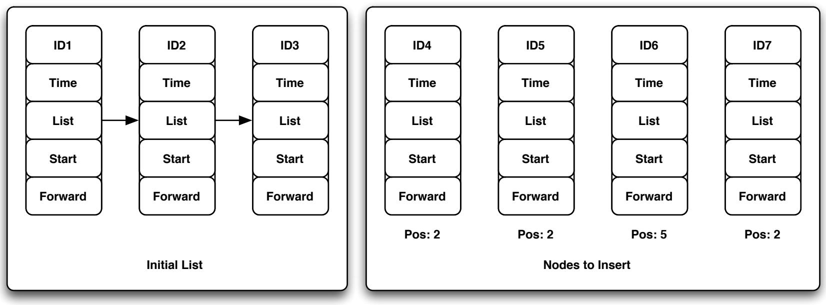
Without a history list the linked list would look different, 1, 2, 7, 4, 5, 6, 3. This order depends on the order of how the messages, containing the operations, were received. This order might differ from host to host. Since some messages sent over network might be delayed and others not.

2.2.3 Discussion

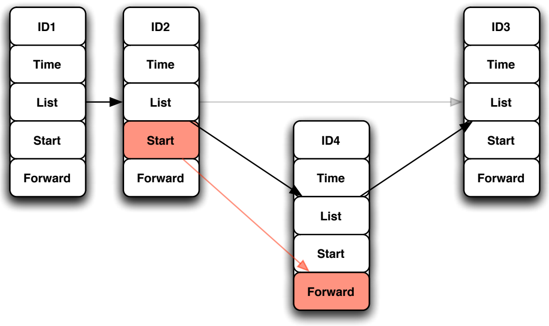
1. **Real-time time stamps**
   This approach relies on the quality of the time stamps. If the clock of a host loses, then on concurrent insertions the nodes from this host might always have a smaller time stamp. As a consequence the user's intention is not always hold. Also, the problem of deleted nodes has to be resolved. One idea is to use some kind of checkpoints on which all hosts agree to remove unnecessary nodes.

2. **Operational Transformation**
   The OT approach achieves a high user responsiveness due to the reason of executing local operations as fast as possible. The problem is the need of a history of all executed operations which is very space expensive and slows down the algorithm that transforms all operations. To reduce this problem checkpoints could be used to agree on a set of executed operations and remove them from the history.

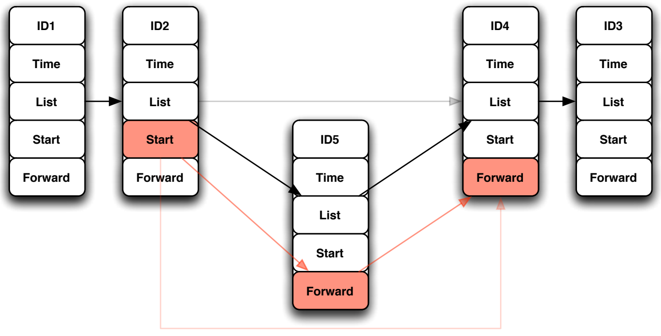
The decision for the concurrency control mechanism falls to OT. Because of the better possibility to integrate this approach into Eclipse, the internal document can be used. With the real time approach the linked list need to be implemented and integrated into Eclipse. The linked list is put off as future work.



(a) *Initial List*

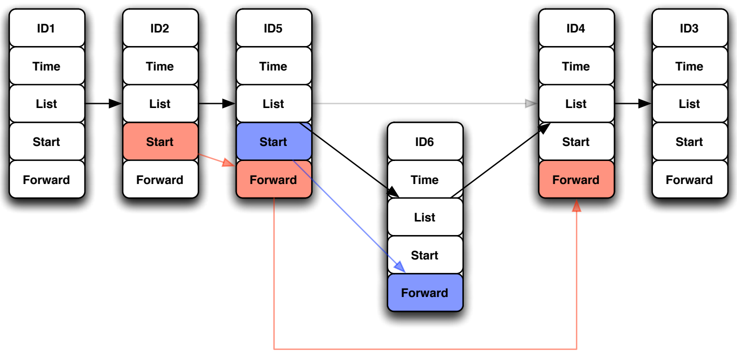


(b) *Insertion of Node 4*

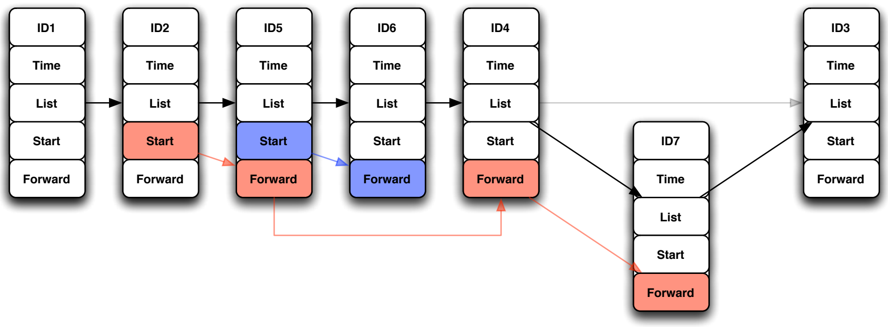


(c) *Insertion of Node 5*

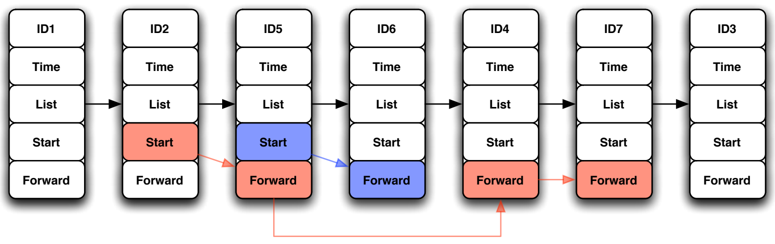
Figure 2.7: *This figure shows the insertion of node 4 and 5 into the initial list containing node 1, 2 and 3.*



(a) *Insertion of Node 6*



(b) *Insertion of Node 7*



(c) *List after all insertsions*

Figure 2.8: *This figure shows the insertion of node 6 and 7 and the final list structure after all insertions.*

3 Eclipse

This chapter introduces Eclipse [2]. Most people think that Eclipse is a Java development environment, but it is much more than that. This understanding comes from its early releases. At the beginning, Eclipse was meant as a Java development environment. It will be shown what else can be done with Eclipse beside development of Java, even though Java development is an important part of Eclipse.

3.1 Introduction

The Eclipse company was founded in November 2001. It was formed by a board of industrial companies. Until February 2004 other companies joined the board. From then on Eclipse was reorganized into a non-profit corporation, which is now an independent body which will drive the platform's evolution to provide an open-source framework to the end user.

The Eclipse project is divided into different subprojects, which group common tasks.

- Eclipse Project
  In this project, the basic Eclipse resources are developed, like the core platform, the Java development tools (JDT) and the Plug-in development environment (PDE).

- Eclipse Tools Project
  In this project, tools for the Eclipse platform are developed which introduces completely new features to the Eclipse platform. The C/C++ development tools (CDT) or the visual editor (VE) for creating graphical user interfaces are two examples, which give new abilities to the Eclipse platform.

- Eclipse Technology Project
  In this project, new technologies are developed. These projects are used by others to implement new features or tools. This project is meant to be something like a library or framework.

- Eclipse Web Tools Platform Project
  In this project, tools for development of web-centric and J2EE appliacations are developed.

- Eclipse Test and Performance Tools Platform Project
  In this project, a standard base platform test and performance tools are developed.

- Business Intelligence and Reporting Tools (BIRD) Project
  In this project, a base framework for reporting in Java is developed.

3.2 Eclipse - An Overview

As mentioned before, Eclipse is not just a Java development environment. With the release of version 3.0 it is more than that. Before this version it was intended to be a Java development tool. There was already the possibility to extend and/or improve the platform but not in such a generic way like it is now.

In Figure 3.1 the different parts of the Eclipse platform are shown. The runtime platform is responsible for dynamically discovering and loading plug-ins. It maintains information about each plug-in and the extension points offered by the plug-in, which are explained in Section 3.2.1. The Runtime Platform itself is not a plug-in. The Workspace is responsible for resource handling, creating and managing all kind of resources (projects, files and folders). The Help plug-in offers extension points to provide help facilities. The Debug plug-in defines a debug model and user interface to for building debuggers. The Workbench plug-in offers user interface facilities, for navigating the platform. This plug-in also offers additional toolkits, SWT and JFace, to build user interfaces. The Team plug-in offers support for team developing facilities. The platform runtime and the plug-ins together build the Eclipse platform.

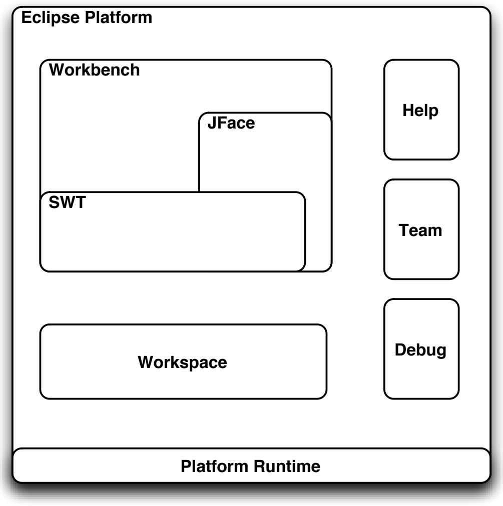


Figure 3.1: *This figure shows the Eclipse components. At the bottom is the Platform Runtime as the core element of Eclipse. It handles the loading mechanism of plug-ins. The other parts are plug-ins. The Team and Help plug-ins offer team an help functionality. The Workspace handles the resources which all plug-ins might access. The Debug part offers debugging facilities for other tools. SWT, JFace and Workbench are responsible for the graphical user interface. They offer basic widgets but also more sophisticated GUI elements.*

```
1 <extension
2      id="org.eclipse.jdt.debug.ui.SnippetDocumentFactory"
3      name="%snippetDocumentFactory.name">
4      point="org.eclipse.core.filebuffers.documentCreation">
5      <factory
6           extensions="jpage"
7           class="org.eclipse.jdt.internal.debug.ui.snippeteditor.SnippetDocumentFactory
                 ">
8      </factory>
9 </extension>
```

Listing 3.1: *DocumentCration extension point*

3.2.1 Extension Points

In Eclipse, the concept of so called *Extension Points* is very important. Whenever a plug-in is defined it might use and/or offer extension points. These points describe well defined interfaces to facilities a plug-in offers. First of all a plug-in must define an extension point before another plug-in can use it. The extension point is defined according to an XML schema, it defines attributes and expected values. A very basic extension point would define a unique ID and its name. With the help of an XML schema the points are well defined and other plug-ins might access these points. To access such an extension point it is defined in the *plugin.xml* file. This file is part of every plug-in and describes the accessed and defined exteions points.

Example

Listing 3.1 shows how a plug-in has to plug itself to an extension point. Lines 2-4 define the id, the name and the used extension point. This has to be done for every extension point. In line 7 the class is defined which is called by the extension point. And in line 6 the file extension for which this point is used is defined. Whenever the platform creates an internal document, the extension point registry is searched for plug-ins which plug into this extension point. If the document which will be created has the extension jpage, as in listing, the SnippetDocumentFactory is called. The factory class in this case implements the IFactory interface which defines the callable methods.

3.2.2 Rich Client Platform

Eclipse is not just a development environment (see 3.2). The extension points were introduced so that implemented features can be used and/or modified. Eclipse offers also the possibility to create *Rich Client Platforms* (RCP). This platform is a set of minimal plug-ins to develop a rich client application. For example, if an application does not need any debug facilities then the debug plug-in is not necessary and therefore should not be integrated into the application. So every plug-in not needed can be omitted. The minimal platform application with a user interface would contain the following plug-ins:

- org.Eclipse.core.runtime
- org.Eclipse.ui

Every desired application can be build on top of these two plug-ins.

Now, what could that be used for? Today, a lot of applications have to be small, fast, platform independent and should also have a nice look and feel. The main thing is the platform independency. Therefore, a lot of applications are developed inside browsers. But browsers offer not that much possibility to implement an application. For example, the user interface is very basic, as long as the application should run and behave in the same way in different browsers and on different platforms. But wherever Java runs, an application can be implemented using Eclipse and the Rich Client Platform. So the developer has the same possibilities like developing an application for just one platform. Actually, there is not much difference between an RCP developed with Eclipse compared to a Java application. First when using Eclipse RCP to develop a new application all the mentioned plug-ins and much more from the community could be used. Another reason to use RCP, is the huge community behind Eclipse which does a lot of testing. Then, the plug-in system used in Eclipse can be used to improve and extend the application.

Basically, there is no difference, but when using Eclispe a lot of work is already done. For example the plug-in mechanisms can be used, also all existing plug-ins can be used. Otherwise all this components need to be implemented additionally.

3.2.3 Java Development Tools

The Java development tools can be compared to professional tools like JBuilder from Borland or NetBeans from Sun. Of course it is open source, like the most Eclipse plug-ins. Figure 3.2shows where and how the JDT plugs into the Eclipse platform. It is only a shallow figure. But JDT and also the PDE use many plug-ins inside the Eclipse Platform. The Plug-in Development Environment is important for the development for plug-ins. It can be used to develop and to debug plug-ins.

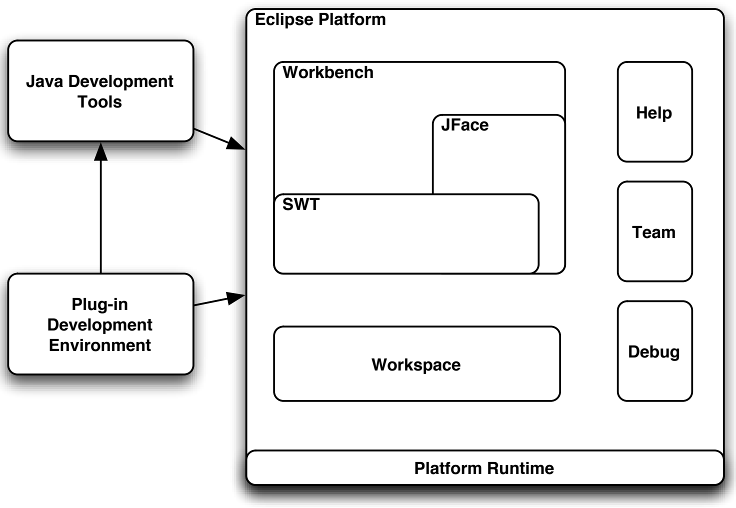


Figure 3.2: *This figure shows how the Java Development Tools and the plug-in development environment are connected. It is a very abstract view. JTD plugs into several plug-ins of the Eclipse Platform and the PDE plugs into the Eclipse Platform and the JDT.*

4 Group Editor in Eclipse

In this chapter, the process of implementing and integrating a group editor into Eclipse is described. The information from the previous chapters like the plug-in mechanism of Eclipse and the theory of concurrency control is used to implement the group editor. The different parts of the development process are discussed separately. The group editor design is layered and an Eclipse plug-in integrates the editor into Eclipse. Some basic information are needed to understand what and how it was done. First of all, the group editor is implemented in Java and represents a plug-in, which can be installed into Eclipse. The idea was to integrate the shared editor feature into the Java Development Tools, so that the features from JDT can be used. To do so, some steps are needed. First, when a document is not shared it shouldn't make a difference to the user. Therefore, a mechanism has to be implemented which checks if a document is shared or not. Second, the user input has to be taken and redirected to the concurrency control and then sent to the other group members. Also the possibility for insertion of text changes from other hosts has to be provided. That aims to interrupt the existing cycle of user inputs.

4.1 Software Design

Figure 4.1 illustrates the general software design approach. Five different parts are shown. The reason for this layered and component oriented approach is that all the parts can be developed independently. To guarantee this independence interfaces are designed which handle the cooperation between these parts. This way the implementation can differ as long as the interfaces remain. The different parts are discussed now.

Network and P2P Layer First it was planned to implement a network layer based on the new Java non-blocking input/output library (Java NIO). But after some reviews of already existing frameworks the choice falls to JGroups [1]. JGroups is mainly a group communication framework for multicast group communication. It also offers some more high level protocols like group membership handling, which represents the P2P layer. How JGroups is used in detail is explained in Section 4.2.1. This layer is responsible for sending and receiving messages and also for group membership handling, when new users join or leave the group.

The topology of the P2P network created by the group editor is document oriented. That means for every shared document a new P2P network is created (see Figure 4.2). Inside a group, which represents a P2P network, every host is connected to every other host, that is mainly for latency reasons and the

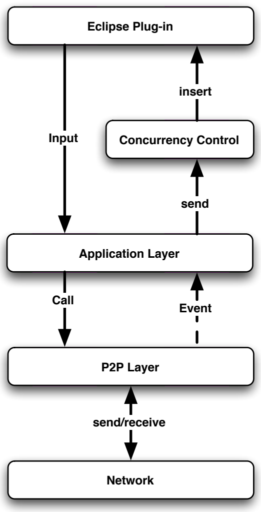


Figure 4.1: *This figure shows in general how the software is built, and how the different parts are connected together.*

fact that usually not that many users join a document. From the document point of view it looks like all the networks are disjoint, because they don't know anything of other documents. When a message is created on one host it is then sent to all members of that group. Therefor no messages are sent and received which don't concern that group.

In Figure 4.3, the same network is shown from the host point of view. The difference between the two views is that a host can have more than one connections to another host, because they might share two documents. Although this is not very efficient, reusing connections has been put off as future task.

Application Layer This layer is responsible for the main application logic. It connects the lower layers to the Eclipse plug-in and the concurrency control logic. This layer receives the user input from Eclipse, handles it locally and sends it through the p2p layer to another group editor on another computer. Upon receiving an event from the p2p layer it dispatches the event to the concurrency control.

Concurrency Control This component is responsible for the concurrency control of the user inputs from the different users. Whenever two users on two different machines make some input, then this component has to guarantee that all users see the same result. It uses the approach of operational transformation discussed in Chapter 2. After the user input has been transformed it inserts it into the text document inside Eclipse.

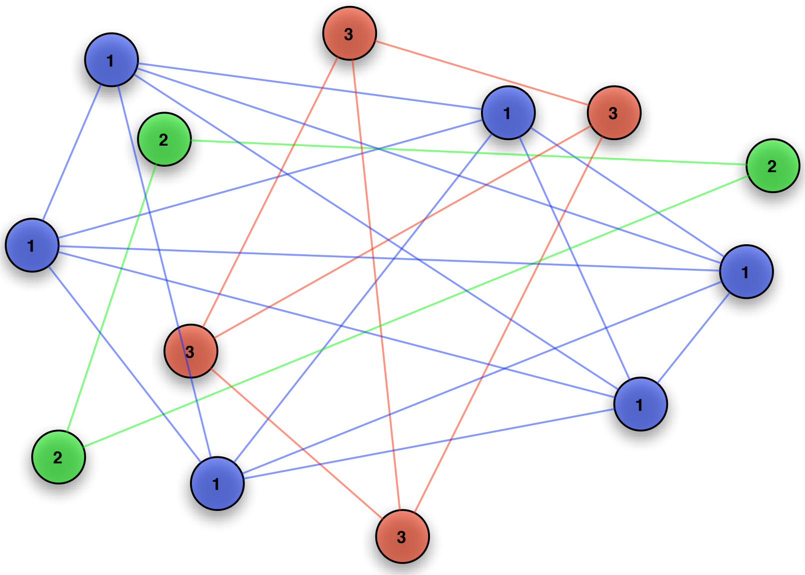


Figure 4.2: *This figure shows the P2P topology used in the group editor. Three documents are shared, the circles represent a shared document on a host, they might also be on the same physical host. This figure shows the network from the document point of view.*

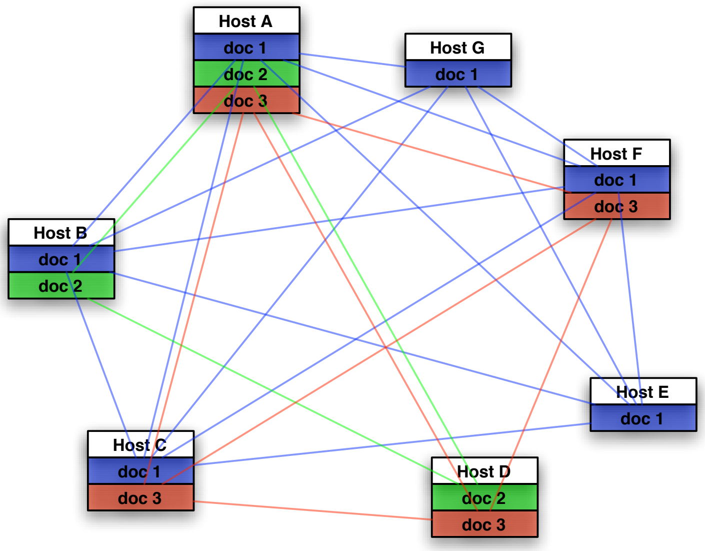


Figure 4.3: *This figure shows the p2p topology from the host view. It is the same topology as in Figure 4.2.*

Eclipse Plug-in This component integrates the group editor into Eclipse. It takes the user inputs and passes them to the application layer. This component is also responsible for the generation of the user interface of the group editor.

4.2 P2P Layer

In this section, the tasks of the p2p layer are explained in detail and JGroups is introduced.

4.2.1 JGroups

JGroups is a toolkit for reliable multicast communication. This framework offers network features for the group editor. Sending messages to other users are needed and also joining and leaving a group. Some features of JGroups are the ones listed below.

- Group creation and deletion. Group members can be spread across LANs or WANs
- Joining and leaving of groups
- Membership detection and notification about joined/left/crashed members
- Detection and removal of crashed members
- Sending and receiving of member-to-group messages (point-to-multipoint)
- Sending and receiving of member-to-member messages (point-to-point)

JGroups consists mainly of three different parts, a protocol, a network abstraction and building blocks. All of them are now introduced.

Protocol In JGroups a protocol, can be defined by an XML file. Listing 4.1 shows such a possible xml file. The *config* is the outermost tag. Inside this tag, the different protocol layers are defined. The first one, in this case *TCP1_4* defines the bottom most protocol in the protocol stack on top of it comes the next one and so on until to the topmost one, the *pbcast.GMS* protocol. The protocol tags correspond to Java classes in JGroups. For every tag there has to exist an instanciable Java class. A short description of the different protocol layers is given below.

- TCP1_4
  This class defines the network protocol used to send and receive messages. It starts the local listening port and then waits for incoming requests or for sending request from the upper protocol layer.
- TCPPING
  This layer is responsible for finding other hosts running JGroups and for connecting to them. A list of possible remote hosts can be given to which this protocol tries to connect to.

```xml
<config>
    <TCP1_4 start_port="7800" loopback="false" bind_addr="testserver" />
    <TCPPING timeout="3000" initial_hosts="localhost[12000]" port_range="1"
        num_initial_members="1" />
    <FD timeout="2000" max_tries="4"/>
    <VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
    <pbcast.NAKACK gc_lag="100" retransmit_timeout="600,1200,2400,4800"/>
    <pbcast.STABLE stability_delay="1000" desired_avg_gossip="20000" down_thread="
        false" max_bytes="0" up_thread="false"/>
    <pbcast.GMS print_local_addr="true" join_timeout="5000" join_retry_timeout="2000"
        shun="true"/>
</config>
```

Listing 4.1: *This listing shows the protocol definition in xml used in the group editor.*

- FD
  This layers is responsible for failure detection by sending ping messages. If a response is not received within a timeout interval the pinged host is suspected as unreachable and FD sends a suspect event up the protocol stack.

- VERIFY_SUSPECT
  This layer counter checks the suspected members by the FD layer. If they are still considered as suspected it forwards the suspect information up the stack. Otherwise, the suspect event is discarded. This layer tries to minimize false suspicious.

- pbcast.NAKACK
  This layer is responsible for lossless FIFO delivery of multicast messages, using negative acknowledgements. Whenever messages are missing the receiver ask for retransmission of the messages.

- pbcast.STABLE
  This layers drops messages seen by all members of the group. Because of possible retransmission of messages, each member stores all messages. If it is sure, that all members have received a certain message the message is dropped by this protocol layer. It sends the lowest message id to all members. According to this received message id the minimum of all this ids is calculated and all lower messages are dropped.

- pbcast.GMS
  This layer is responsible for the group membership service. It handles joins and leaves of members. It also handles suspect events from the FD layer by excluding the suspected member. Whenever the topology changes this layer sends a new group view to all members of the group.

JGroups offers a lot more protocols to build a protocol stack and also the possibility to implement own protocols. But for the group editor the presented protocol stack suffices to accomplish the task of communication.

Network Abstraction As in Java there is also in JGroups a network abstraction layer.Java sockets are commonly used to implement network applications.

In JGroups on top of these sockets the protocols are implemented and form a protocol stack. This protocol stack is used by the *JChannel* to provide a high level network abstraction. In Figure 4.4 the life cycle of such a channel is presented containing four different states in which the channel may be.

In the open state the channel is initialized and the corresponding protocol stack is set up. Whenever a connect is executed the channel changes into the connected state, within that state the channel may perform sending and receiving operations. It then can return to the open state. When the channel is disconnected it can be reconnected and again switches to the connected state. Whenever the channel goes to the closed state it is not usable anymore the channel has to be dropped. And the fourth state is the shunned state, in which the channel is excluded from the group, but if reconnect is set to true the channel can reconnect. Otherwise, it changes to the closed state and the channel gets unusable.

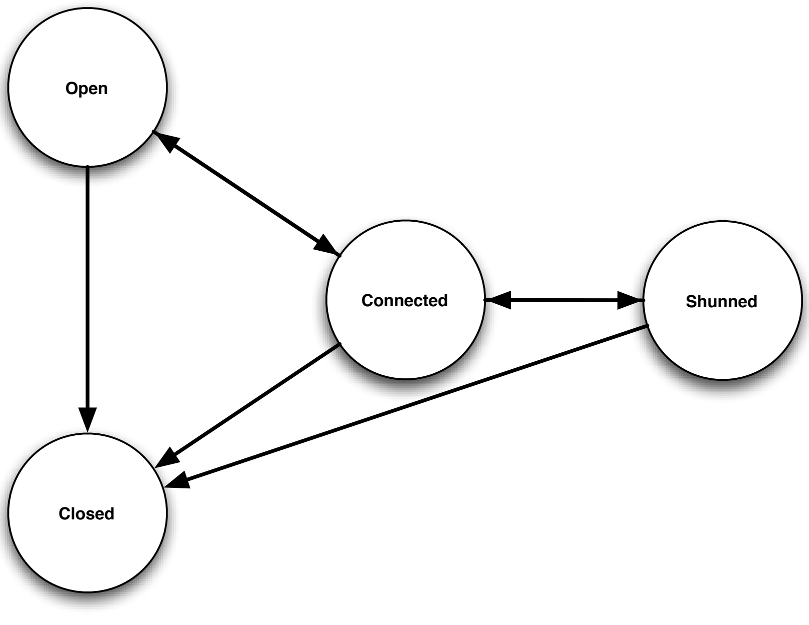


Figure 4.4: *This figure shows the states a JChannel in JGroups can have.*

Building blocks The building blocks are high level components which offer additional features on top of JChannel and the protocol stack. In Figure 4.5 the *PullPushAdapter* is used for non-blocking receive on top of a JChannel. Usually, this has to be implemented by the software developer by using multiple threads. This adapter offers this feature to the application. Most of the blocks provide high level communication features, for example distributed data structures like queues, lists, hash maps and trees. There are a lot of other building blocks like: Distributed Hashtable, Distributed Tree, Distributed Queue, Notification Bus and Transactional Hashtable.

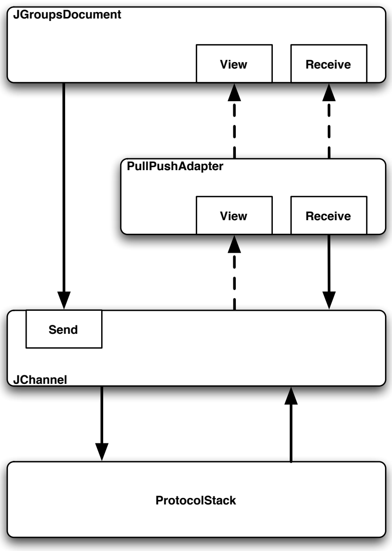


Figure 4.5: *This figure shows how JGroups is used for the group editor. There are 4 different kinds of components in this figure. JGroupsDocument is part of the group editor implementation and the others, PullPushAdapter, JChannel and the ProtocolStack, are part of JGroups.*

4.2.2 JGroups in the group editor

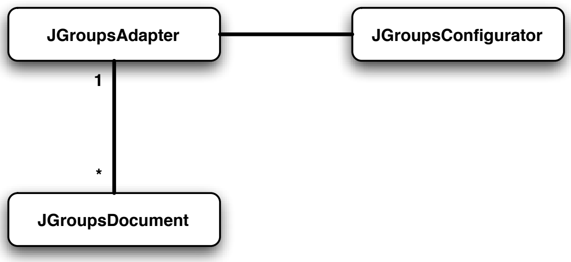


Figure 4.6: *This figure shows the main p2p layer design.*

Figure 4.6 shows the integration of JGroups and the P2P layer. The P2P layer is responsible for creating P2P networks or groups. As mentioned before the idea was to create a group for every shared document, represented by the *JGroupsDocument* class. Inside this class all needed information for group communication is stored and maintained. A user is removed if it leaves the group or crashes, and when a new user connects to the group it is added to the user list. Mainly, there are three different actions the P2P layer has to offer to the application layer, share a document, join a document and leave a document. In Listing 4.2 the service provided by the P2P layer to the application layer is

```
public interface IP2PService {
    public boolean start();
    public void join(String filename, String hostname, int localport, int remoteport,
         MessageContainer joinMsgCont, MembershipListener listener);
    public void leave();
    public void leave(String filename, MessageContainer leaveMsgCont);
    public void publish(String filename, int localport, MembershipListener listener);
    public void send(MessageContainer msgcont, String docname);
    public void send(MessageContainer msgCont, String docname, String username);
}
```

Listing 4.2: *The listing of the IP2PService interface*

listed. All these methods are implemented by the *JGroupsAdapter* class, which represents the P2P layer.

Start By calling this method the JGroupsAdapter is being initialized and the message handling is started. But no network connections are opened.

Send There are two different send methods, one is for sending a message to a whole group and the other is for sending messages to another specific host. But it is only possible to send a message to a single host if it is also in the same group.

Publish The first user who wants to publish a document calls this method. The adapter creates a JGroupsDocument which represents a group. It opens the ports, registers all listeners for messages and membership changes. After that it waits for incoming messages or user input.

Join This method is called when a user wants to join a document group. How the join exactly works is explained later. This action does also all the initializing stuff as the publish operation. But after initiating the document group the join protocol is started. When the protocol has finished the user has joined the document group and the p2p layer waits for incoming messages and user input.

Leave There are also two different leave methods. One is for leaving a document group by sending a leave message and removing all information corresponding to this document group. The other leave method is for leaving all document groups by calling the single leave for all documents.

4.3 Application Layer

This component is the glue of the whole application. It is responsible for handling incoming messages and events from the p2p layer, dispatching input information from the user to the different documents and also handling user input from the GUI. The application layer is mainly represented by the *CoreApp* class which contains several data structures to store user and document information. Figure 4.7 illustrates the design of the application layer. As part of the application layer there are two components, the *CoreApp* class and the *ConcurrenyControl* class. The latter one will be explained in a separate section, but it is needed here for better understanding. At the beginning, when the application is started, the *CoreApp* is initialized by setting local information, like the user name, the local network information, the port and the ip address. After that *CoreApp* is ready for actions triggered by a user. The important actions are share, join and leave. All of them correspond to documents which can be published, joined or left.

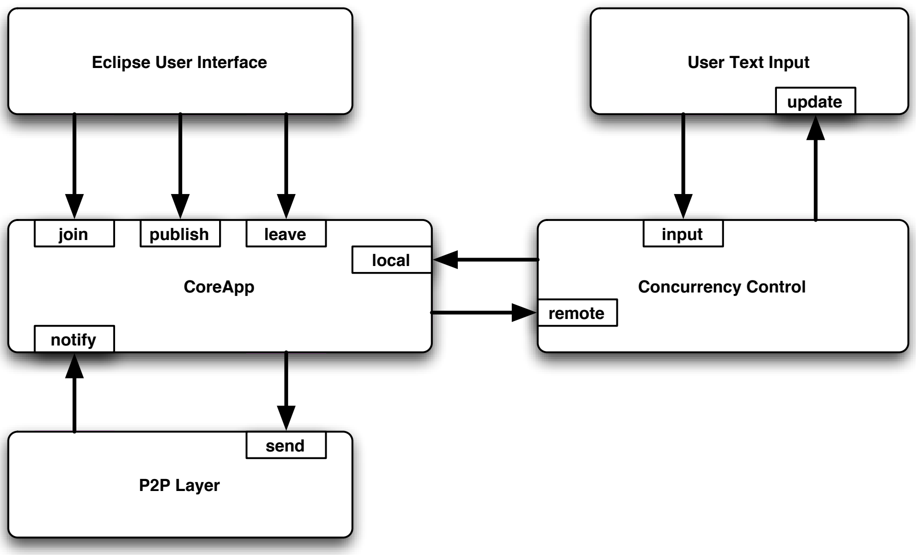


Figure 4.7: *This figure shows how the different components of the application layer are connected.*

Publish When the publish operation is called, JGroups is started and a local server is set up so that other clients can join the published document. The other important modification is the change of the user input insertion flow. Usually when a user inputs text into a document, the change is immediately executed. Figure 4.8 shows the original input from a user. The GUI receives events from the system triggered by a user. These events are translated into user input, handled, inserted into the intern document representation. This change is then reported to the GUI. There is no interruption in the input flow. To distribute the changes this flow has to be broken up, as shown in Figure 4.9. The input has to be redirected to the network layer and also to the transformation component, after the transformation the input is inserted into the intern document representation. These figures present how the thread model looks like. Because of the usage of more than one thread to handle user input and network input, it was not easy to ensure, that the input is still correct. This will be discussed in detail later.

Join The join operation is similar to the publish operation in that the user input cycle has to be broken up and replaced by the new input cycle. Additionally the join protocol is started which connects to a given host and requests the initial document. How this protocol works is explained later.

Leave The leave operation reverts the modifications necessary for the join and publish operations. It reinstalls the original user input cycle and initiates the leave operation on the p2p layer.

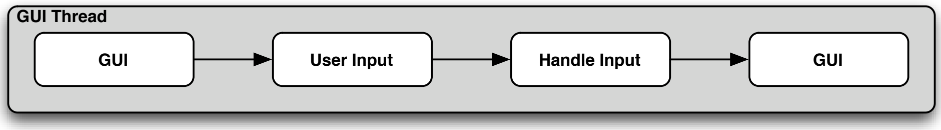


Figure 4.8: *This figure shows the original user input process.*

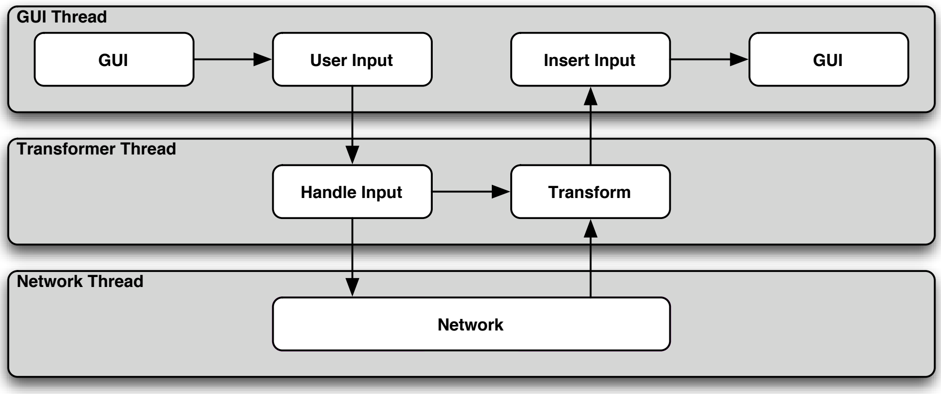


Figure 4.9: *This figure shows the user input process after the publish operation is done. Here the input is redirected to the different components.*

4.4 Concurrency Control

This section describes the implementation issues of the OT approach discussed in Chapter 2. It handles concurrent user input on different peers. To achieve document consistency a vector time stamp is needed which is provided by the *VectorTime* class from JGroups, where it is used for the causal communication protocol. Because of a dynamic P2P system the vector time stamps need to support changing of hosts. When a host joins or leaves, the number of elements in the time stamp change. As the *VectorTime* class supports this it is also used in the group editor. Whenever a peer leaves or joins the network the vector time stamps have to be updated and since the *VectorTime* class supports this it is used. In the Transformer component the algorithm, which transforms local and remote operations against the state of the document, is implemented.

Operations

In Section 2.2.1 operations on a text document are defined. These operations, *Insert* and *Delete*, are represented by the *Operation* Java class. This class carries all information about an operation: the position, the character, the pword and the state in which the operation was created.

4.4.1 adOPTed

The implementation has to guarantee the causal order of all the operations. Additionally, it is also responsible for the editor responsiveness. Local operations are given a higher priority then remote operations. The Java class *Adopted* implements this algorithm. This class also handles joining or leaving hosts by updating the *VectorTime* and deleting the history of all executed operations. The history is dropped because of two reasons. First, the time stamps change and cannot be compared anymore. A new time stamp might have a different number of elements. Second, all hosts have to agree on the set of hosts within the group and on the state of the document.

4.5 JGroups Protocols

There are mainly two protocols, one for joining and one for leaving.

4.5.1 Join Protocol

The join protocol is for joining an existing P2P network which represents a document group. When a host wants to connect to a P2P network, it first executes a *connect* on JGroups which results in a new P2P network that includes the joiner. From that point on the joiner is able to send messages to the whole group or to a specific host in the joined group. Then the join protocol starts. Figure 4.10 shows the protocol flow.

Join-Request This is the initial message sent in the join protocol. The joiner sends information about itself to all members of the P2P group, to make itself known to all other peers.

Join-Reply This message is the response to the *Join-Request* message. The sender of the reply message makes itself public to the joiner.

After these two steps, all members of the P2P group know each other and have all information needed to communicate. After that the document distribution starts. Depending on the joiners information the joiner requests a whole project or only the document. These requests are sent to only one peer, not to all peers. The joiner sends the request to the peer from which the joiner got the first *Join-Reply*, which is now called the replier.

Project-Request The joiner sends the document and project name to the replier. The replier then stores the document and serializes the whole project and sends it to the joiner.

Project-Reply The joiner deserializes the project and initializes it. Then the document is opened and the joiner is now part of the document group.

Document-Request The joiner sends the document name to the replier which serializes the document and sends it back to the joiner.

Document-Reply The joiner deserializes the document and integrates it into the project and the document will be opened and the joiner is part of the document group.

This join protocol does not support concurrent joins and writing. While a new host is joining the group and other hosts are still writing some of these operations might be lost. A smarter protocol which supports joins at any time is put off as future work.

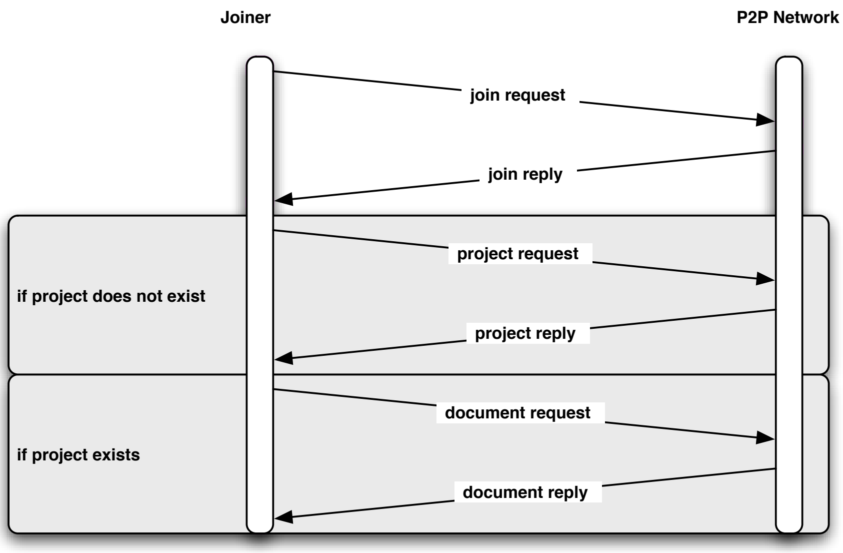


Figure 4.10: *This figure shows the join protocol. After the join messages the joiner only communicates with one other peer, the replier.*

4.5.2 Leave Protocol

The leave protocol is even simpler than the join protocol. The leaving host sends a messages to all peers notifying that he is leaving the document group. Then it closes all the network and P2P components on the leaver side and restores the initial state in the application as if no shared writing took place. All peers receiving a leave message remove the leaver from the local information store, and the leave protocol is finished.

4.6 Integration into Eclipse

This section describes how the group editor was integrated into the Eclipse framework. Specifically, the editor was integrated in JDT.

4.6.1 GUI

First of all, a user interface was developed and integrated into Eclipse. This was done by using extension points. The following extension points were used.

org.Eclipse.ui.views This point is used to register a view. The view maintains the shared documents.

org.Eclipse.ui.viewActions This point is used to register actions. Actions are menus which the user can select and execute. The registered actions correspond to the actions which the application layer offer namely publish, join and leave.

org.Eclipse.ui.preferencePages This point is for registering a preference page within the Eclipse preference window. In the preference page the user name, the local network address and also the remote network address are defined.

The user interface was kept simple, because the focus of this work lies in correctness of the group editor.

4.6.2 Eclipse

In Eclipse two other extension points were used.

org.Eclipse.core.runtime.preferences This extension point is for storing the preferences. When Eclipse is closed the settings are stored permanently and when started the next time to old settings are reloaded.

org.Eclipse.core.filebuffers.documentCreation This extension point is used for defining a new document factory. The reason for that is to interrupt the already described user input cycle and then to get the user input for further processing. By defining this extension point also a type of file has to be added. This type is a Java file, defined with the file name extension. But also JDT uses this extension point to introduce his own document factory and there is no deterministic way which factory is chosen. Therefore, a small hack to ensure that the group editors document factory is chosen needs to be done. More details about it and how to do it is discussed in the Appendix.

4.6.3 Redirect User Input

This was the hardest part, because a single threaded user input has to be transformed into a multi threaded user input scenario. Some odd consistency problems had to be solved and the operational transformation approach had to be integrated. The scenario can be divided into two parts operational transformation and the synchronization of the multi threaded local user input. In Figure 4.11, the whole design of the user input redirection is shown which is discussed now.

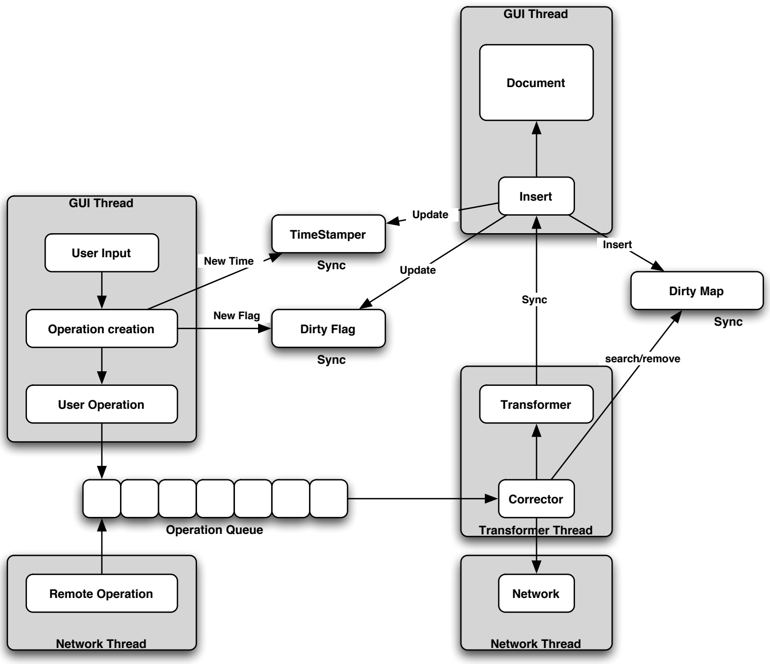


Figure 4.11: *This figure shows the user input redirection and the correction facilities to solve the multi threaded input synchronization.*

User Input Correction

Since the user input cycle is split up, some problems appear. Figure 4.12 shows what happens when the user input thread is divided into more threads. The user types a new string into the document. First, the input is received by the GUI thread where the operations are created. This operations are put into the Operations Queue. Let's assume that the Queue Consumer Thread blocks. The user interface is still responsive which means the user might go on with modifying the text. The operations created before are still in the queue and the new operations don't take account of them. This results in a different document state than desired by the user.

Therefore some correction mechanisms needs to take care of this situation. There would be an easy one: block the user interface. But this decreases the editor responsiveness which is not desired.

The idea of how to solve this problem is shown in Figure 4.13. A Dirty Flag and a Dirty Hashtable are introduced. The Dirty Flag represents the state of the document. When a new operation is created the dirty flag from the document is attached to it and whenever a operation is inserted into the document the dirty flag is incremented. Then the operation is put into the operation queue. The corrector component consumes the operations from the queue. If the state of the document is newer, the dirty flag has a higher value, the document was changed while the operation was in the queue. The information to insert the operation are from the past. Therefore, the operation is checked against all operations happened in the meantime. The operations can be read from the hashtable and the original operation has to be adjusted. If an operation in the Hashtable has a smaller flag than the one from a new operation this flag can be removed from the hashtable. After the correction the operation is sent to all members of the group and is locally processed.

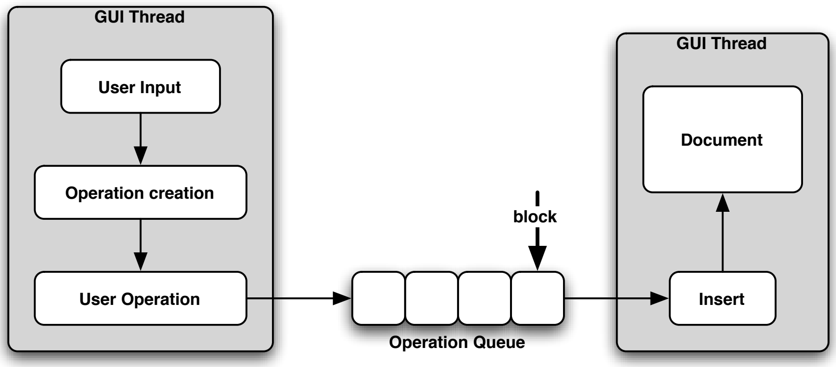


Figure 4.12: *This figure shows what happens if the user input cycle is broken up and inconsistencies might appear.*

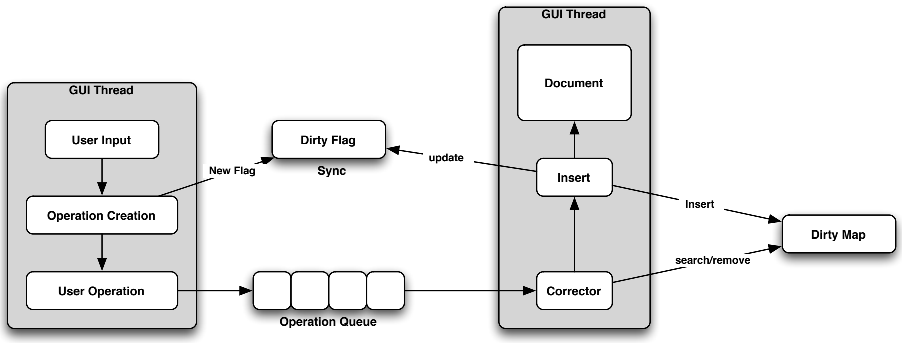


Figure 4.13: *This figure shows the approach to correct the user input inconsistency.*

5 Summary

This thesis implements a group editor in Java. The editor is designed as a Eclipse plug-in which is integrated into the Java Development Tools. The thesis addresses the issues of concurrency, responsiveness and implementation.

To solve the concurrency problems which appear in distributed systems two approaches are discussed. To guarantee consistency the operations executed by the users must be ordered in some way. The sorting should be the same on every host. It should also be meaningful, sorting according to the IP address would be completely senseless. The first and chosen technique uses the approach of Operational Transformation (OT). This approach is commonly used for group editors because it achieves consistent documents and a high responsiveness. The OT approach guarantees document consistency and high responsiveness.Transformation functions transform operations received on a host according to the document state it has at the moment. The operations are transformed so that the effect on all documents is the same, regardless of what operations have already been processed. Thats how consistency is achieved. Local operations are executed immediately so a high responsiveness is achieved. Remote operations are delayed and executed when there is time. The second technique uses real-time timestamps to guarantee that the order of the operations is the same on every host. This approach uses a special linked list to keep track of the insertion history. Both approaches have in common that they keep a history of past insertions. This comes from the fact that in a distributed system messages might be delayed and lost. The OT approach was favored over the time stamps approach because of lower implementation complexity.

to handle the consistency issues. The other one was dropped because of higher implementation complexity.

The implementation was done in Java. The editor is plugged into the JDT of Eclipse. This way all the features provided by JDT like code completion, syntax highlighting and code formatter can be used. The editor plug-in plugs itself into the input cycle of the user. First of all, the user input is received and operations are created. Next these operations are sent over the network to other users. Then the operations are transformed using OT and, finally, executed. Executing an operation means inserting it into the internal document representation. Remote operations are also received and after the transformation executed. They are only executed when there is time to do so, local operation are given a higher execution priority.

For communication the JGroups framework is used. JGroups is a group communication toolkit for Java. It offers membership handling like joining, leaving and creating groups. In terms of the editor, these groups define a document group. These groups define a P2P network. The choice of employing JGroups was because it is a widely used and well tested communication framework.

5.1 Discussion

Gclipse is a prototype. Of course a lot of GUI and usability improvements need to be done. But the implementation shows that the technique of OT is usable for handling document consistency and high responsiveness. The subjective impression of the responsiveness is very good, as if working in a single user editor. Although more testing and benchmarking need to be done, to find out how Gclipse behaves with high user load. The approach of integrating Gclipse into JDT is probably not the best one. Another idea is to design an editor from scratch with collaborative editing in mind. The integration into JDT was the most difficult part. The reason is to get all the user input. The input comes through different ways in the internal document representation. To find all these ways and handle them correctly and synchronized is difficult and error-prone. Also some of the JDT features do not work as expected. For example in some cases to many strings are inserted which of course is done on all hosts.

Because of the early state of Gclipse a lot of improvements need to done. Some are described in Chapter 6.

6 Future Work

In the process of designing and developing the group editor described in this thesis, some problems and improvements emerged. Some of them could be solved or were integrated into the group editor, but due to lack of time not all issues could be integrated. Here is a list of possible future works to extend and improve the group editor.

6.1 Concurrency Control

In Chapter 2 two approaches to gain consistent documents were discussed. One with time stamps and the other with transformation functions.

6.1.1 Time Stamps

This approach was discussed but not yet implemented. This approach could be integrated into the editor so the two concurrency control mechanisms could be compared.

6.1.2 Operational Transformation

The transformation functions work on operations that always handle one character. It would be interesting to find out if there is a possibility for blocks and not only single characters. That might decrease the number of sent messages and might also improve the responsiveness. It would also be interesting if the algorithm which transforms the operations could be improved.

6.1.3 Checkpoints

In the discussion of the two approaches the term of checkpoints was mentioned several times. Checkpoints are an idea to improve the concurrency control. For the time stamps to keep the linked list small and for the OT approach to reduce the stored data. A consensus algorithm could be used to implement and realize checkpoints.

An approach for integrating checkpoints would be to use a consensus algorithm.

6.2 Group Editor

In Chapter 4 the implementation of the editor is described. Here are several improvements or new features which might be added.

6.2.1 Implement a new Editor

The actual implementation of the group editor is an integration into JDT. Eclipse now offers an easy way to implement a completely new editor. Into this new editor the collaborative working features could be integrated better than into JDT. For example, the changes done by other users could be colored and the cursors of other users could be shown. This way the input cycle has not to be broken up. Another improvement could be that all text documents could be shared not only Java documents. There is only one disadvantage the syntax coloring, and the features provided by JDT cannot be used directly.

6.2.2 Improve the Group Editor

The editor is integrated into JDT therefore most of the features like command completion, syntax highlighting and code formatter are supported. But the support is not for all that good. Sometimes to much characters are inserted like the closing braces. This would be an important part to improve.

6.2.3 GUI

The GUI should be improved, especially the view showing the shared documents. It could be improved with additional buttons for the actions. Also a list of users which are connected to a specific document would be a nice feature. The context menu when clicking on a java document should contain the publish, join and leave actions.

6.3 P2P and Network

In Section 4.2 the P2P layer and the Network layer are discussed.

6.3.1 Reuse of Network Connections

A host can have more than one connection, when sharing more than one document, to the same host. This should be improved and the connections should be reused. JGroups need to be checked if there is a way to achieve this. Otherwise a new Network layer should be implemented or other network frameworks should be analyzed.

6.3.2 P2P Protocols

The join protocol was introduced. It was kept very simple. Therefore the lack of concurrent joining and writing. This should really be improved, otherwise at least a mechanism to block the group when a user is joining need to be provided which is not the desired solution.

1 Bibliography

[1] Bela Ban. Jgroups - a toolkit for reliable multicast communication. URL: `http://www.jgroups.org/`.

[2] Eclipse Community. Eclipse - a application development framework. URL: `http://www.eclipse.org/`.

[3] Wikipadia Community. Wikipadia, the free encyclopedia. URL: `http://www.wikipedia.org/`.

[4] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developers Guide to Eclipse.* Addison-Wesley, 2004.

[5] Berthold Daum. *Java-Entwicklung mit Eclipse 3.* dpunkt.verlag, 2004.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407. ACM Press, 1989.

[7] Abdessamad Imine, Pascal Molli, Gérald Oser, and Michaël Rusinowitch. Achieving convergence with operational transformation in distributed groupware systems. Technical report, INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 2004.

[8] Guido Krüger. *Handbuch der Java-Programmierung.* Addison-Wesley, 2002.

[9] Brad Lushman and Gordon V. Cormack. Proof of correctness of ressel's adopted algorithm. *Inf. Process. Lett.*, 86(6):303–310, 2003.

[10] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297. ACM Press, 1996.

[11] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, pages 59–68, 1998.

[12] Paul Wilson. *Computer Supported Cooperative Work.* Cromland, Incorporated, 1991.