A Framework for Real Time Collaborative Editing in a Mobile Replicated Architecture

A thesis submitted for the degree of

Doctor of Philosophy

Sandy Citro, M. Tech (IT),

School of Computer Science and Information technology,

Science, Engineering, and Technology Portfolio,

RMIT University,

Melbourne, Victoria, Australia

15th June, 2007

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Sandy Citro

School of Computer Science and Information Technology

Royal Melbourne Institute of Technology

15th June, 2007

Acknowledgements

This thesis is the result of three and half years of work whereby I have been accompanied and supported by many people. It is a pleasure that I have now the opportunity to express my gratitude for all of them.

I would like to thank my supervisors, Assoc. Prof. Jim McGovern and Dr. Caspar Ryan. I could not have imagined having better advisors and mentors for my PhD, and without their common-sense, knowledge, perceptiveness and clear direction, I would never have finished. Since I started my research, both of them have never stopped showing their enthusiasm and inspiration which have kept me going. Throughout my thesis-writing period, they have provided much encouragement, sound advice, and lots of good ideas.

I wish to thank my extended family for providing support and encouragement throughout my study: my dad, my mom, and all my brothers and sisters. Although they are overseas, they have not stopped giving me encouragement in whatever way they can.

I wish also to thank my own family for providing such a loving and caring environment for me. I would like to thank my beautiful wife, Maria, for giving me a full support for my study, for sacrificially taking care of our children during my research, and most importantly for constantly loving me. I would like to thank my gorgeous children, Jochebed and Jochanan, for continually giving me incomprehensible joy throughout my study.

Lastly, and most importantly, I wish to thank my personal Saviour and Lord, Jesus Christ. He created me in His own image, forgives me, saves me, supports me, gives me joy, and He loves me. To only my Maker, my Father, my Savior, my Redeemer, my Rewarder, to only a God like You do I give my praise. To Him I dedicate this thesis.

"The LORD is my shepherd, I shall not be in want."- Psalm 23:1

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- S. Citro, J. McGovern and C. Ryan, *An efficient consistency management algorithm for real-time mobile collaboration*, In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, pp. 287-294, 2005
- S. Citro, J. McGovern and C. Ryan, *Handling and Resolving Conflicts in Real Time Mobile Collaboration*, in R. Meersman, Z. Tari and P. Herrero, eds., *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, Springer-Verlag, Montpellier, France, 2006, pp. 21-22.
- S. Citro, J. McGovern and C. Ryan, *Extending Real Time Mobile Collaboration Algorithm to Handle Membership Events in an Ad-Hoc Mobile Network*, In *Proceedings of The 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Atlanta, Georgia, 2006.
- S. Citro, J. McGovern and C. Ryan, *Conflict Management For Real-Time Collaborative Editing in Mobile Replicated Architectures*, In *Thirtieth Australasian Computer Science Conference (ACSC2007)*, pp. 115-124, Ballarat Australia. ACS, 2007

This work was supported by the Australian Telecommunications Cooperative Research Centre and the Distributed Systems group at RMIT University.

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.

Contents

List of figures

List of tables

Abstract

Mobile collaborative work is a developing sub-area of Computer Supported Collaborative Work (CSCW). The future of this field will be marked by a significant increase in mobile device usage as a tool for co-workers to cooperate, collaborate and work on a shared workspace in real-time to produce artefacts such as diagrams, text and graphics regardless of their geographical locations.

A real-time collaboration editor can utilise a centralised or a replicated architecture. In a centralised architecture, a central server holds the shared document as well as manages the various aspects of the collaboration, such as the document consistency, ordering of updates, resolving conflicts and the session membership. Every user's action needs to be propagated to the central server, and the server will apply it to the document to ensure it results in the intended document state. Alternatively, a decentralised or replicated architecture can be used where there is no central server to store the shared document. Every participating site contains a copy of the shared document (replica) to work on separately. Using this architecture, every user's action needs to be broadcast to all participating sites so each site can update their replicas accordingly.

The replicated architecture is attractive for such applications, especially in wireless and ad-hoc networks, since it does not rely on a central server and a user can continue to work on his or her own local document replica even during disconnection period. However, in the absence of a dedicated server, the collaboration is managed by individual devices. This presents challenges to implement collaborative editors in a replicated architecture, especially in a mobile network which is characterised by limited resource reliability and availability.

This thesis addresses challenges and requirements to implement group editors in wireless ad-hoc network environments where resources are scarce and the network is significantly less stable and less robust than wired fixed networks. The major contribution of this thesis is a proposed framework that comprises the proposed algorithms and techniques to allow each device to manage the important aspects of collaboration such as document consistency, conflict handling and resolution, session membership and document partitioning. Firstly, the proposed document consistency algorithm ensures the document replicas held by each device are kept consistent despite the concurrent updates by the collaboration participants while taking into account the limited resource of mobile devices and mobile networks. Secondly, the proposed conflict management technique provides users with conflict status and information so that users can handle and resolve conflicts appropriately. Thirdly, the proposed membership management algorithm ensures all participants receive all necessary updates and allows users to join a currently active collaboration session. Fourthly, the proposed document partitioning algorithm provides

flexibility for users to work on selected parts of the document and reduces the resource consumption. Finally, a basic implementation of the framework is presented to show how it can support a real time collaboration scenario.

Chapter 1

Introduction

1.1. Background

Alice, Bob and Cameron are three scientists working together to produce a conference paper. When computers were not networked, they would have written the document on a disk, which is passed on from one scientist to another to be updated accordingly. Alternatively, they can sit together in front of one computer to write the paper together or they can handwrite the paper and then type it to the computer.

The development of the computer network technology allows users, using their own computers, to collaboratively edit a shared document over the network. The early technology of computer supported collaborative editing requires Alice, Bob and Cameron to be located at a common meeting room [123]. Each author, using an allocated workstation, updates the shared document that appears on a large digital whiteboard in front of them. The workstations and the digital whiteboard are on a computer network such that the updates can be sent to the digital whiteboard over the network.

The advancement of computer network opens up the opportunity for real-time collaborative editing where Alice, Bob and Cameron are no longer required to be in the same location. Each user has the real-time access to the shared document at his/her computer without having to see each other. In a real-time collaboration session, users work on a local device which communicates with the individual devices of other users via message passing. Each user interacts with the shared document (as it appears on his or her device) with changes propagated to other users as soon as possible so as to reduce the possibility of update conflicts.

Many collaborative editing systems have existed to support real-time collaborative editing. Real-time collaborative editing can adopt a centralised architecture [28, 97, 125] or a replicated architecture [46, 113, 119]. In a centralised architecture, only a single copy of the document exists on a central server, with participants updating it directly in a synchronous manner. In a replicated architecture, on the other hand, each device holds a replica of the shared document. Each update of one user is propagated directly to the other users without having to go through a central server.

1.2. Motivation

In the past few years, with the advancement of mobile computing technology, mobile devices have significantly added to the richness of distributed computing. Mobile devices are becoming increasingly powerful allowing implementation of the applications that were able to be seen only in PCs and laptops [93]. Mobile technology and the ability to connect to other devices afford the opportunity to extend existing applications into new realms, so that mobile device users are able to communicate, process and present information as much as do users of desktop devices. Given the ability to work on their portable devices anytime, anywhere, users are becoming more mobile, and the ability to connect to other devices opens up the possibility of extending collaboration to a wider range of users and circumstances.

Although many real-time collaborative editing systems exist [35, 54, 94, 118], they are not applicable to mobile environments. Although Alice, Bob and Cameron may work on their own mobile devices, the existing collaboration systems either assume the use of a dedicated central server where they have to be connected to be able to collaborate or assume the use of personal computers with large available resources such as memory, storage capacity and network bandwidth. The existing collaboration systems that utilise a replicated architecture may in principal be suitable for mobile networks. However, since they are designed not for mobile environments, they do not take the resource consumption into account in the design of their algorithms.

The most powerful forms of collaboration, but also the most challenging, will be those that require real-time synchronous collaboration in environments where there is peer to peer communication with no central servers, intermittent connection and low capacity devices. This research aims to explore such applications and to provide the basis for providing powerful new mobile applications but also improving collaboration support and usability of groupware in general.

1.3. Research Questions

The aim of this project is to devise a model or a framework to support and satisfy the requirements for real-time collaborative editing application in a mobile replicated architecture while addressing the limitations of mobile network environments. This project is divided into a number of smaller sub-projects, each of which addresses the above requirements.

This project will investigate the real time collaborative behaviour of object based editing applications in mobile environments. Key challenges in mobile environments are reduced processing power, memory capacity and lower bandwidth and connectivity. If new applications are to be developed for or migrated to mobile environments, then research is needed to extend current collaborative techniques to such environments. This project will propose new strategies, algorithms and models to address issues and challenges in such collaboration applications. These proposed techniques will then be combined into a comprehensive framework to allow real-time collaboration applications to run successfully in a range of ad-hoc mobile environments. This project will address the following research questions:

1. *How can document consistency among mobile devices be efficiently enforced in a mobile replicated architecture?*

This project aims to devise a consistency management algorithm that will work in a replicated architecture while taking into account the limitations of mobile devices and mobile network environments.

2. *How can the existing methods be adapted to handle and resolve conflicts in a real time mobile collaborative editing application?*

   As opposed to collaborations with a central server as the central collaboration manager, peer to peer collaborative applications have to able to handle conflicts without the presence of a central server. As an extension to the devised consistency management algorithm, this project also discusses the various potential conflicts and devises a strategy to manage and handle conflicts consistently in a replicated architecture.

3. *How can the collaboration session participants be managed in a mobile collaborative environment?*

   Taking into account the dynamic membership events in an ad-hoc mobile network environment, this project aims to devise a strategy that transparently let users continue to collaborate smoothly in the midst of sites joining, leaving, disconnecting and missing operations.

4. *How can document partitioning be used to enable collaboration on large documents and to reduce resource consumption in mobile devices?*

   As mobile devices have limited display and/or memory capacity, users may decide not to work on all parts of the document. This project aims to devise a strategy to allow the document to be divided into several partitions in order to allow users to flexibly work on selected parts of the document and to reduce resource consumption at the same time.

1.4. Methodology

1.4.1. Literature Review

An overall literature review focusing on real-time collaboration in a replicated architecture is provided in Chapter 2. Each of the above 4 questions are addressed in each chapter, which will include a further review of issues relevant to that research question.

1.4.2. Algorithm Construction

Firstly, a document consistency algorithm is devised to ensure consistency of the shared document. Then, for each sub-project, the devised algorithm is built on top of the algorithm devised in the previous sub-project. This ensures that the algorithm achieves the intended functionality while still achieving the goal of the previous sub-project. In other words, the conflict management algorithm is built on top of the devised consistency management algorithm to ensure that the document consistency is still maintained, the membership management algorithm is built on top of the consistency and conflict management algorithm to ensure that the document consistency and conflicts are properly handled while handling the various membership events, and finally, the document partitioning is built on top of all of them so as to maintain all achieved functionality while allowing users to divide the document into partitions.

1.4.3. Testing and Performance Evaluation

The testing and the performance evaluation will mainly be done using a software simulation written in the JAVA programming language. Each algorithm has been tested for its correctness in a simulation environment. Simulation methodology is chosen over implementation because this thesis focuses more on the performance and correctness of the algorithm than the actual implementation of the collaboration. Furthermore, the simulation methodology is more flexible and can easily be configured to represent various collaboration scenarios.

Firstly, for the consistency management algorithm, collaborating sites run through various scenarios to ensure the consistent views among them. The algorithm is then compared against existing algorithms to measure its performance. Storage space, bandwidth, and processing power usage are the parameters to be used to determine how well the algorithm performs. Secondly, for the conflict management algorithm, the algorithm is tested using a prototype application to ensure all types of conflicts can be properly handled consistently at all sites. Thirdly, for the membership management, collaboration sites run through scenarios consisting of combinations of disconnections, reconnections and late-joins. The devised strategy is deemed to be correct if all sites can resume collaboration despite those limitations of mobile network and all sites eventually end up at a consistent state. Finally, the document partitioning algorithm is tested for its correctness in a simulation with different number of partitions and its performance is measured against the non-partitioned document. Storage space, bandwidth, and processing power usage are used as the parameters to be used to determine how well the algorithm performs.

1.5. Thesis Organisation

This thesis is organised such that each sub project (each research question) is discussed in each separate chapter. This thesis is organised as follows.

1. Chapter 2 discusses the general development of computer supported collaborative work both in non-mobile and mobile networks.
2. A document consistency management algorithm is proposed in Chapter 3 to ensure the consistency of the collaboration document. The algorithm takes into account the limited resource of mobile environments. The performance of the algorithm is also analysed to determine the most efficient implementation of the algorithm.
3. In addition to ensuring document consistency in the midst of concurrent updates, in Chapter 4, a conflict management technique is proposed to handle and help users resolve conflicts. The proposed conflict management technique can be used with any conflict resolution strategy.
4. In order to support the membership events in mobile networks, a membership management algorithm is proposed in Chapter 5 to ensure the collaboration continues

smoothly regardless of the membership events. The performance of the proposed algorithm is analysed and it shows that the algorithm is able to handle the various membership events without consuming significant additional resource.

5. In Chapter 6, a document partitioning algorithm is proposed to provide users with flexibility to work on different parts of the document and to avoid unnecessary resource consumption. The performance of the proposed algorithm is analysed to determine under what conditions the proposed algorithm can be used to reduce the resource consumption.
6. Chapter 7 presents the application architecture of the collaborative editor and a sample scenario of how the proposed framework can support real time collaboration.
7. Finally Chapter 8 provides the summary and the conclusion of the thesis, and outlines the future work.

Chapter 2

Mobile CSCW

Collaborative editing is a part of Computer Supported Collaborative Work (CSCW), which has been a major research area in computer science for over two decades. This chapter briefly reviews the background of CSCW, extensively reviews various existing work in the area of real-time collaborative editing, and discusses the challenges in implementing practical real-time collaborative editing in mobile ad-hoc network environments.

CSCW is defined as the study of how people work together or collaborate using computer technology [20]. In other words, it is a generic term which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques [145]. It includes emails, hypertext that includes awareness of the activities of other users, videoconferencing, chat systems, and real-time shared applications, such as collaborative writing or drawing.

Groupware is a common term for systems that support CSCW. Ellis et al [46] defines groupware as computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment. This technology may be used to communicate, cooperate, coordinate, solve problems, compete, or negotiate. While traditional technologies like the telephone can also be used to communicate, cooperate, and solve problems, the term groupware refers to technologies relying on modern computer networks, such as email, newsgroups, videophones, or chat.

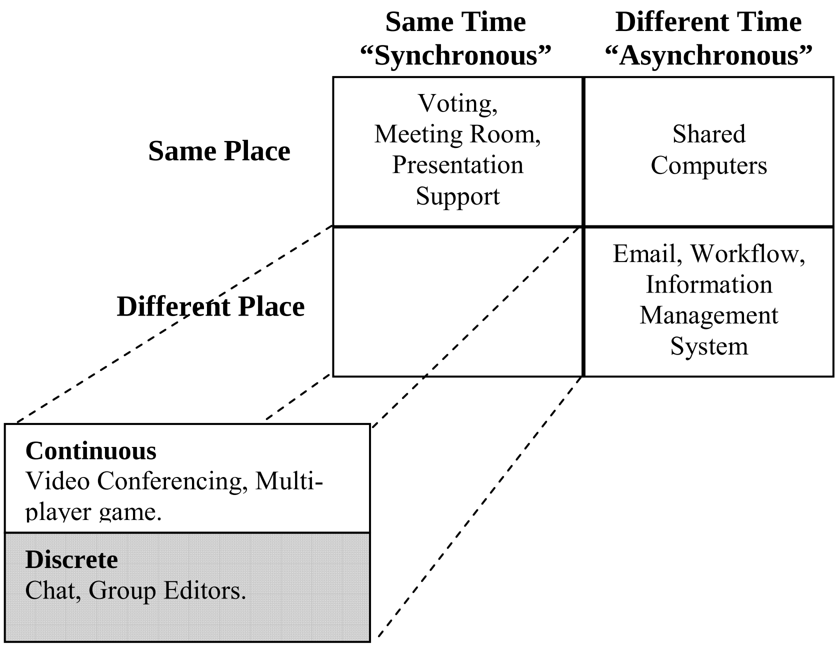


Figure 2-1 Groupware Dimensions, adopted from [21]

Groupware technologies can typically be categorized into two dimensions [58]:

1. Time
    a. Synchronously/real-time - communication occurs at the same time
    b. Asynchronously/different time - communication occurs at different times.

2. Place
    a. Same place - the participants can see each other physically.
    b. Different place - the participants are widely dispersed and can not see each other physically.

In synchronous (real-time) collaboration, participants are connected at the same time, and can communicate with each other while working on the document. The document update of one user is propagated immediately so that the other users can view the update as soon as possible. Some examples of synchronous groupware are chat applications, video conferencing applications and group editors, such as GRACE [130], GroupKit [119], Groupgraphics [109] and GroupDesign [74]. The collaboration can happen with users at the same place or at different places. A meeting room is an example of collaborating at the same place, while email is an example of collaborating from different places.

Synchronous collaboration can be further categorised into: continuous collaboration and discrete collaboration. In a discrete collaboration, such as a collaborative document editing, the updates are sent from time to time as soon as the user makes changes to the shared document. The updates are not continuous hence the updates are not continuously streamed. In a continuous collaboration such as video conferencing and multi-player first-person shooter games, however, the updates are continuous since all users need to know what is happening with other users at all time. The updates are sent, or rather streamed, continuously to capture all changes, movements, and events.

This thesis focuses on discrete synchronous groupware that allows users to collaboratively edit the same document from different places. For continuous applications, such as video conferencing and multi-player games, readers can refer to relevant work, such as [28], [27], [87], [88], [91], [92], [107] and [143]. Discrete synchronous groupware for users in different locations is referred to in this thesis as Real Time Collaborative Editors or simply Group Editors. Unlike most existing work, however, this thesis focuses on addressing challenges and requirement to implement group editors in mobile ad-hoc network environments where resources are scarce and the network is significantly less stable and less robust than wired fixed networks.

The remainder of this chapter is organised as follows. Firstly, section 2.1 discusses the real-time collaborative editing (group editors) in general and the requirements that need to be met by group editors. Secondly, section 2.2 discusses the proliferation of mobile ad-hoc networks, their characteristics and consequently the requirements for group editors to be implemented in mobile ad-hoc networks. Thirdly, section 2.3 reviews the existing work in group editors both in non-mobile and mobile networks, discusses their limitations and drawbacks. Finally, section 2.4 concludes the literature review and briefly points to how the challenges and limitations of the existing group editors are addressed in this research project.

2.1. Real-Time Collaborative Editing (Group Editor)

A real-time collaborative editing application (group editor) falls into the synchronous groupware application category and can be defined as an application that allows two or more people, using their own devices, working together on a shared document at the same time regardless on their geographical location, in order to produce a group-intended final document. Basically, a group editor is a system that allows several users to simultaneously edit a document without the need for physical proximity and allows them to synchronously observe each others' changes [111]. It enhances collaboration by providing a shared workspace in which users can carry out tasks such as organising ideas, jointly preparing papers, and brainstorming. The document may range from a simple text document, with characters and simple operation primitives, to a complex multimedia document, with objects and more sophisticated operation primitives. At some level of abstraction, any collaborative application can be considered a generalized editor [39]: a text editor edits a text file, a debugger edits a debugging history, a graphic editor edits a graphic file, a spreadsheet edits a worksheet file, an online whiteboard edits composite objects, a computer based training tool manipulates training document and a CASE tool edits a document with predefined objects.

2.1.1. Groupware Application Transparency

There are two general approaches to providing computer support for synchronous collaboration [16]: *collaboration awareness* and *collaboration transparency*. The *collaboration awareness* approach designs an application specifically to support cooperative work. On the other hand, *collaboration transparency* approach considers that single-user applications are pervasive and therefore provides a mechanism for sharing and leveraging legacy single-user applications for multi-user collaboration [147, 134]. The mechanism is unknown, or transparent, to the original application and its developers.

In the collaboration awareness approach, a collaborative editing application is specifically designed and purposely built from scratch to satisfy the specific collaboration requirements. All existing groupware systems described in section 2.3 were built using a collaboration awareness approach. The main reason is that by using the collaboration awareness approach, the system can be built specifically and purposefully to satisfy the requirements of a given research project. This approach has a few drawbacks: relatively higher cost to develop as compared to the latter approach as it has to be built from scratch and the fact that users might not be familiar with the newly built system unlike the commonly known pervasive single-user applications.

In the collaboration transparency approach, there is typically a collaboration enabling application that invokes a single-user document editing application such that all modifications done to the document are sent to other participants (they should also use the same collaboration enabling application, open the same document editing application and join in the same collaborative session). Some examples of collaboration-transparent systems include XTV [9], HP SharedX [54], SharedApp [5] for the X Window System, and NetMeeting, a collaboration-transparent system for the Windows platform [2].

Although the collaboration-transparency approach imposes an inflexible, tightly-coupled style of collaboration, it does not adequately support some key groupware principles [15]. Firstly, conventional collaboration-transparent systems do not allow input from more than one person at a time; therefore it does not promote concurrency. Secondly, in conventional collaboration-transparent systems, all participants see exactly the same view at the same time - What You See Is What I See (WYSIWIS) [122]. Although this enforces consistency, it does not give users flexibility to navigate to different parts of the document. Thirdly, since the collaboration transparency approach builds on the current single-user application, it does not provide adequate group awareness. Fourthly, due to the use of centralized display-broadcasting architectures, conventional collaboration-transparent systems generally require higher network bandwidth than collaboration-aware applications, which are typically replicated.

Another way to build collaboration transparent systems is to replace some components of the single-user application at run-time, such as Flexible JAMM (Java Applet Made Multiuser) [15]. Flexible JAMM replaces the single-user objects in the otherwise single-user Swing-based application with their multi-user object counterparts. Using *dynamic binding*, the run-time resolution of a function invocation or data retrieval allows an instance of one class to function in place of another, thus a single-user interface object may be replaced at run time by a multi-user version. Since the application source code is not modified, the substitution is transparent to the shared application. Flexible JAMM, however, can not be applied to all application platforms. In many object-oriented environments, such as Smalltalk and Java, dynamic binding is the norm, whereas in others, such as C++, dynamic binding must be explicitly programmed [15]. Furthermore, Flexible JAMM can only be applied to serialisable Swing-based Java applications. AWT-based Java applications have some constraints that do not allow Flexible JAMM to be implemented.

A recent work on collaboration transparency, CoWord [147], uses a different approach namely *Transparent Adaptation* approach. Although this approach does not require any changes to the single-user application's source code, it requires the application's API to be adaptable to the data and operational models of the underlying concurrency control technique. Combined with a replicated architecture, the transparent adaptation approach is able to achieve high responsiveness, concurrent work, relaxed WYSIWIS, and group awareness.

This thesis uses the collaboration awareness approach since its focus is on devising algorithms that provide essential functionality in real-time mobile group editors, not developing the collaboration application itself. The algorithms devised in this thesis, however, can be used by developers that wish to leverage existing single-user applications for real-time multi-user collaboration.

2.1.2. Groupware Architecture

In terms of the application architecture, real-time collaborative editing applications, like other distributed systems ranges from *centralised*, where a central server or process maintains the shared data and processes any updates or modifications to the shared data, to *fully replicated*, where each participant maintains a copy of the shared data, processes the updates locally and notifies the other participants about the updates.

In a centralised architecture, clients connect to a server to join in a collaboration session, and all updates in the client machines are sent to the server so all other clients can get the updated document from the server. The central server is responsible for managing the concurrent updates by the participants and maintaining the consistency of the shared document. The server holds the main document and each participant (client) device either holds a synchronised copy (thick client) or a view of the main document (thin client).

In a thin-client centralised architecture, each client machine does not hold a document copy. It simply receives and displays the document view or representation from the server. It does not need to know the underlying data structure of the document [110]. The client machine needs to be able to capture user inputs as the user modifies the document at his/her local device display. The user input is sent to the server to be processed by the server. Once the document in the server is updated, the participant will retrieve the updated view or representation of the main document in the server so as to display the latest document state. Most of the existing centralised architecture collaborative editing applications fall into this category, such as Rendezvous [108], WebEx [7], GoToMeeting [1], and WebArrow [6].

In a thick-client centralised architecture, each client machine holds a copy or a cache of the document for better responsiveness. It also holds local state or local context that allows the participant to locally change the view or presentations of the document and to do any client-side processing. Every update generated by each collaboration session participant that affects the document state is sent to a central server. This architecture is suitable for high-performance or complex group collaboration software such as multi-player online games (e.g. World of Warcraft [8], Half-life [17] and Halo 2 [23]), and some group editors such as Dolphin [124] and Tivoli [96].

Regardless of the 'thickness' of the client in a centralised architecture, the communication happens only between each participant and the server, and participants do not communicate directly with each other. The centralised architecture provides simpler maintenance of the document since the server is the one machine which is responsible for managing the document updates and each device does not need to handle concurrent updates to the document. Moreover, the server can easily ensure the consistency of the document as the server holds the main document and each device synchronises with the server regularly. This approach does however have a number of drawbacks. Firstly, the application is less responsive since the user does not view his/her changes immediately due to the round-trip latency as the user interaction must travel to and from the central server. Secondly, a central server must be present and running at all times, thus introducing a single point of failure whereby the entire collaboration session ceases when the server is down. Thirdly, if an individual device is unable to connect to the server, whether due to total network failure or low bandwidth or sporadic disconnection, that user cannot participate in the session. Fourthly, in a wireless mobile network environment, the presence of a server is not guaranteed, especially in an ad-hoc wireless environment where two or more devices can start a session wherever they are as long as they are within a wireless transmission range of each other (either directly or indirectly through packet forwarders). Finally, depending on the implementation, the network usage may be high since all operations must be directed through a central server even when a thick client is used and a local copy of the document exists. Nevertheless, this architecture can be simple and effective in a local area network where these factors can be more readily controlled. In contrast, the centralised approach is not readily suited for mobile networking environments which are characterised by high network delays, frequent disconnection and reduced bandwidth and relatively high communication cost.

In contrast to centralised architectures, in a replicated architecture, each participant holds a *replica* of the shared data and each participant is responsible for processing all the changes to the replica that it holds. Each user makes changes to his/her local replica, and then notifies other users about the changes by broadcasting messages or updates without going through a server. In the replicated architecture, each site acts as both a client that interfaces with the user, and a server that manages the actual collaboration. Replicated architecture, however, can further be categorised into: *semi-replicated architectures* and *fully replicated architectures*.

As the name implies, the participants in a fully replicated architecture hold replicas of the document and they communicate solely with each other without the presence of a dedicated server. In a distributed environment with nondeterministic communication latency, a fully replicated architecture is usually adopted for the storage of a shared document in order to meet the requirement for high responsiveness [39]. Some examples of group editors that employ a fully replicated architecture are Colab [123] GROVE (*GR*oup *O*utline *V*iewing *E*ditor) [47], REDUCE (REal-time Distributed Unconstrained Cooperating Editing) [131, 132], GroupDesign [74] GRACE (GRAphics Collaborative Editing) [130], and Draw-together [65]

Similar to the fully replicated architecture, each participant in a semi-replicated architecture also hold a replica of the document. The difference is that the presence of a server is required in the semi-replicated architecture. The server, however, does not manage the document nor does it manage the document updates. Instead, the server simplifies the collaboration by providing some particular centralised service, such as participant registration, session management, or centralised sequencing/ordering. DistView [113] and GroupKit [118] are examples of group editors in this category.

Compared to the centralised architecture, the replicated architecture has several advantages. Firstly, when compared to the thin-client centralised architecture, replicated architecture potentially requires less bandwidth since each participant processes his/her own replica, thus each user needs to notify only the changes to the shared data. It also provides faster response to the user input as the local replica is updated immediately before the changes are sent and applied at the remote replicas. Secondly, when compared to the thick-client centralised architecture, replicated architecture requires more or less the same bandwidth and comparable responsiveness since each participant device holds the document replica and only updates of the document are sent out either to the server or to the other participants. However, with the exception of the semi-replicated architecture, a central server is not required in a replicated architecture, and therefore, there is no single point of failure. If a site is disconnected, that user can continue working on his or her local replica while other sites can still collaborate with all of the sites to which they have the connectivity. Upon reconnection, the previously disconnected site can re-synchronise with other sites to bring its document up to date. Finally, replicated architecture promotes concurrency as each participant can modify their document replica anytime they want to, without the presence of or regardless of the status of the server. This ability to work concurrently and independently of a central server, and to operate while disconnected from other sites, makes the replicated architecture attractive for real-time collaborative editing, especially in mobile ad-hoc networks. However, compared to the centralised architecture, the replicated architecture has a few disadvantages. It increases the storage requirement since each device holds a document replica. It also requires each device to manage the various aspects of the collaboration such as consistency management and membership management in the absence of a dedicated server. Therefore, this thesis aims to devise the framework to manage the collaboration in a mobile replicated architecture while taking into account the limited resource of mobile devices and mobile networks.

2.1.3. Group Editor Requirements

The process of developing a collaborative application is considered to consist of three main steps [39]: 1) design the functionality, 2) decompose the application into components, and 3) use tools for implementing the components. This thesis focuses on the functionality definition and proposes algorithms to achieve the desired functionalities while taking into account mobile network constraints. This thesis also presents the collaboration framework components showing the role of each component in a real-time collaboration and the basic implementation of the framework. This thesis however does not discuss the collaboration awareness and the social aspect of the CSCW. A number of researchers describe application decomposition and the groupware implementation tools [39, 43, 56, 118, 119], while the collaboration awareness and the social aspect of groupware systems are covered in other work [13, 14, 60, 61, 63, 114, 139, 140].

Ellis et al. [46] and Kanawati [72] outline some features required from a real-time collaborative editing application including the following:

1. *Document Consistency.*

   The document that appears at one participant's device must be consistent with all other documents that appear at other participants' devices.

   In conventional collaboration systems, all participants see exactly the same view at the same time in a manner referred to as strict What You See Is What I See (WYSIWIS) [122]. There are several advantages of WYSIWIS as follows.

   - It is ideal for meetings that require very close collaboration.

- Each user knows exactly what other users are seeing.
- It can be defined in an application-independent fashion, whereby the users do not have to be aware that they are interacting with other users.

Strict WYSIWIS however imposes some disadvantages: it can lead to dispute whenever users are forced to share changes that they do not want to share or view the document portions that they do not want to view.

An alternative to WYSIWIS is WYSINWIS (What You See Is Not What I See) [30] or relaxed WYSIWIS [122], where each user can define his/her own *logical view* of the overall document space based on his/her own interest and responsibility. Relaxed WYSIWIS is attractive as it gives users flexibility to view the document in the way (format) they want, and it is easier to implement across multiple platforms. In relaxed WYSIWIS, the consistency is not imposed on the actual document appearance, but it is imposed on the document data structure and, consequently, the semantic meaning of the document. The document may appear differently on different devices, but as long as the actual data structures are consistent, each user will be able to understand the other users' intention or the meaning of the other users' updates.

2. *Interactivity* or *Responsiveness*.

The collaboration must be as responsive or as interactive as possible to improve the user's collaboration experience. The changes made by a user must be reflected as soon as possible on the document as it appears at his/her local device, and the changes made by a user must also be reflected as soon as possible on the documents that appear at the other users' devices. Furthermore, each user should be allowed to modify any part of the document at any given time.

3. *Dynamic Membership*.

   A single-user application starts as soon as a user starts it, and it ends as soon as the user quits. In contrast, multi-user applications must support a richer regime [108]. A session may be started by someone, e.g., a conference administrator, and in a typical collaboration session, due to various reasons, participants could come and go during a collaboration session. Users will arbitrarily join and will leave as necessary without halting the session. Additionally, sessions may or may not be terminated by the departure of the last user. Real-time collaborative editing systems must support dynamic membership events, such as users joining the collaboration session or users leaving the session temporarily or permanently. On the occurrence of such events, the session must be able to resume as smoothly as possible without intervention from users. These aspects of session management are necessary if multi-user applications are to be readily accessible to their users.

4. *Document Availability*.

   The shared document must be available at all times so that users can get access to necessary document when they need to, regardless of the users' location.

While those requirements may be satisfied by some collaborative applications developed for fixed PC networks, they present a set of new challenges when implemented in mobile networks, especially mobile ad-hoc networks. The characteristics of mobile ad-hoc networks, and the challenges and requirements of real time collaborative editing in mobile ad-hoc networks are discussed in the next section.

2.2. Real Time Collaborative Editing in Mobile Ad-Hoc Networks

Mobile devices have significantly added to the richness of distributed computing in the past few years. Mobile devices range from laptop computers with the equivalent memory and processing power of a desktop machine, through to Personal Digital Assistants PDAs, and programmable smart-phones with considerably less memory, processing power and display capability [93].

Mobile devices were initially designed for personal use with a few personal applications such as calendar, to-do list and other various personal organisers to serve as an electronic personal assistant to the owner. With the advance of technology, however, mobile devices are becoming increasingly powerful. Mobile technology gradually allows applications, which were able to be seen only in PCs and laptops, to be ported into smaller size mobile devices such as PDAs. Applications such as word processors, spreadsheets, and even games are quite commonly found in mobile devices [93]. Furthermore, the ability to synchronise the mobile data with the data in a workstation or a server allows users to bring the data out of the office, work on it anywhere and synchronise it back to the main repository once they are back at the office. Hence, users are becoming more mobile given the ability to work on their mobile devices anytime, anywhere.

The proliferation of wireless networks, such as infra-red, Bluetooth, GPRS and 802.11 standards, has resulted in mobile devices becoming increasingly connected. Networking applications such as email clients and web browsers can be found in mobile devices and can be used to asynchronously collaborate with the other colleague by exchanging email messages and/or downloading files from servers. Users will be able to access company information remotely and to collaborate with colleagues from any location, whenever necessary [52]. Furthermore, different mobile devices, being within a particular proximity, can establish a connection and start communicating with each other. Users of these connected mobile devices can choose to exchange messages, data or files in real time (synchronously) over the wireless network. This opens up the opportunity to develop applications that enable users to collaboratively work on the same document at the same time without the presence of a fixed network infrastructure.

Implementation of the synchronous collaborative editing applications in mobile network environments, especially ad-hoc networks, will open up even greater flexibility and greater potential of anytime-anywhere collaboration. The following are just some of the possible scenarios for ad-hoc collaboration in mobile networks [22]:

- Emergency search and rescue in areas where a wired infrastructure is not available.
- Groups attending a conference can share ideas and data anywhere by conducting "virtual" meetings.
- Field survey operations in remote areas.

- Cooperative group carrying out activities where there is no visual contact, such as in hunting.
- A team of construction workers on a site without a network infrastructure can share blueprints and schematics.
- Staff and security of large events such as concerts, or sporting events can more easily coordinate crowd control and security.
- Military intelligence and strike teams can be more easily coordinated to provide quicker response time.
- Collaborative software engineering.

As reflected in the above scenarios, the collaboration documents do not necessarily have to be a text or worded document. They can be graphic documents, architectural drawing, diagram or real-time maps. Regardless of the type of the documents, the implementation of such real-time collaborative editing application will give greater flexibility in various application domains.

Implementing real time collaborative editing in mobile ad-hoc networks, however, is more difficult than in wired PC networks. The characteristics of mobile ad-hoc network and the limitations of mobile devices present some challenges to be addressed before successfully implementing collaboration in mobile environments.

2.2.1. Mobile ad-hoc network characteristics

A mobile ad-hoc network is defined as an autonomous and self-organising system that consists of wireless nodes/devices that dynamically establish connection [78]. One of the biggest advantages of ad-hoc networks is that they can be quickly created without the need for a fixed network infrastructure. However, due to the hostile characteristics of such a network environment, collaborations over a mobile ad-hoc network have additional requirements as compared to collaborations in a fixed-network environment. Mobile ad-hoc networks have the following characteristics:

- *Dynamically Formed*. Mobile devices move freely in a random and unpredictable manner. When two or more nodes are within a wireless transmission range, they can start establishing connections and communicate with each other.
- *Dynamic Network Topology*. Mobile devices can easily connect and reconnect with each other depending on their proximity to each other. This will result in the frequent change in the number of mobile nodes currently participating in the network, thus the network topology would dynamically and rapidly change without prior notification.
- *Wired/fixed communication infrastructure is not necessary*. Mobile nodes come and go from one place to another, unattached to any static cabling. They are able to establish communication with each other anytime and anywhere, even without the presence of fixed communication infrastructure.

- *Low and fluctuating bandwidth and high latency*. The network connectivity of mobile devices depends on radio frequency technologies to transmit and receive data. As a result, the available bandwidth will vary from location to location depending on factors such as the physical structures of the locations and radio frequency interference. This will cause high latency in places where the bandwidth is low.
- *Frequent disconnection*. The mobility of mobile devices will mean that they keep moving from within the wireless transmission range to outside of range. Each participating node will have to continuously update which nodes are currently participating in the collaboration session. This creates a challenge in the membership management of collaborative work.

The dynamic nature of a mobile ad-hoc network plays a major role in designing the software used for collaboration. Furthermore, compared to fixed PC workstations, mobile devices have the following limitations:

- *Limited display capabilities*. Not only do mobile devices, such as PDAs or smart-phones, have an obviously smaller display screen size, they also have a lower screen resolution and, in some cases, reduced colour depth. They often have to display a large amount of information, such as a large document or a large diagram using paging or scrolling. Displaying a complex graphical image is also a problem due to low screen resolution and lower colour depth.

- *Reduced processing power*. Mobile devices processing power has reached the stage of PCs processing power a few years ago. However, it is still much lower than current PCs processor technology (e.g. 400MHz of a PDA as opposed to approximately 3 GHz of a PC).
- *Reduced memory size*. This will not only affect the performance (responsiveness) of running applications on mobile devices, but will also make it difficult for mobile devices to display and manipulate sophisticated documents and large graphic images.
- *Battery dependency*. Unlike stationary PCs, mobile devices depend on battery life. Processing power and wireless transmission reduce battery life. Increasing the battery life can mean a bigger battery size that will decrease the portability of the device itself.

While many existing software systems fulfil the synchronous collaboration editing requirements, most of them are designed to run on stable and permanently fixed networks where the quality of service is much higher and the state of the network is more predictable [103]. The key characteristics that make the existing collaboration software or algorithms ill suited for ad-hoc networks are their single point of failure, and their assumption of reliable network and large available resources. The current generation of software is very resource intensive in terms of both memory and processor requirements [22].

2.2.2. Real-Time Mobile Collaborative Editing Requirements

While there have been some groupware systems developed for mobile network environments, due to mobile network limitations, there has not been a groupware system that allows seamless synchronous collaborative editing in mobile replicated (ad-hoc) network. Several algorithms have been developed to meet various collaborative editing application requirements. However, implementation of such applications (and algorithms) in mobile networks is not easy, as there are other requirements needed to be satisfied in order to successfully support mobile collaborative editing and there are limitations in mobile network that have not been addressed or taken into account by existing algorithms.

As mentioned in section 2.1, the replicated architecture is well suited to mobile collaborative editing where mobile devices can still continue working on their local devices while disconnected and a central server is not required for collaboration. In a replicated architecture, collaboration starts when a user begins a collaboration session, joined by the other participants. The participants will start the collaboration with each holding the same shared document, either a blank document or a previously saved document. Each user makes changes or modifications to the document as it appears at his/her device, and the updates will have to be reflected on the other participants' document. During the session, due to various reasons, a participant could leave the session, either temporarily or permanently, and another participant may join the session while the session is still running. Looking at this typical collaboration session, a collaborative editing application must satisfy some minimum requirements.

Due to the nature of the replicated architecture, every participant can update their own local replica whenever they want. While it provides flexibility and a greater degree of concurrency, the implementation of mobile real-time collaborative editing applications in a replicated architecture presents some complexities and challenges to the requirements that are mentioned in section 2.1.3. Furthermore, the characteristics of ad-hoc mobile networks mentioned in section 2.2.1 have to be taken into account in the requirements for mobile real-time collaborative editing applications.

Therefore, a successful implementation of a real-time collaborative editing application in mobile replicated architecture requires the general group editor requirements mentioned in section 2.1.3 to be extended as follows:

1. *Document Consistency*.

   In the group editor requirements previously mentioned, the document that appears at one participant's device must be consistent with all other documents that appear at other participants' devices. For implementation in mobile ad-hoc networks, the resource consumption must be taken into account, i.e. due to limitations in mobile ad-hoc network environments, the memory, processing power and network bandwidth consumed to ensure consistency among document replicas must be significantly reduced.

2. *Interactivity* or *Responsiveness*.

   As mentioned previously, the collaboration must be as responsive or as interactive as possible to improve the user's collaboration experience. The changes made by a user must be reflected as soon as possible on his/her local replica as well as on the other users' replicas. For implementation in mobile ad-hoc networks, not only does the resource consumption have to be taken into account, the collaboration must still be interactive and responsive despite the frequent disconnections in mobile networks. Users must be able to work on the document even though they are disconnected, and once they are re-connected, the document must be brought up to date as soon as possible.

3. *Dynamic Membership*.

   As mentioned previously, real-time collaborative editing systems must support dynamic membership events, such as users joining the collaboration session or users leaving the session temporarily or permanently. On the occurrence of such events, the session must be able to resume as smoothly as possible without intervention from users. In a fixed PC network environment, a dedicated server is readily available to handle these events. Users joining and leaving the session can easily notify the server and the server can handle these events and notify other participants accordingly. In contrast, mobile ad-hoc networks do not guarantee a presence of a dedicated server for managing membership. Therefore, in mobile ad-hoc networks, each participant is responsible for handling membership events such as arbitrary joining and leaving. Furthermore, disconnection occurs frequently and unpredictably in mobile ad-hoc networks, and it has to be handled individually and consistently by each participant.

4. *Document Availability*.

The shared document must be available at all times so that users can get access to necessary data when they need to, regardless of the users' location. In mobile ad-hoc networks where a dedicated server is not necessarily present, each device must store a document replica so users can get access to the document even though they are disconnected. Depending on the type of the document, its size can grow too large for mobile devices capacity (either its display screen or its storage space). In a fixed PC network environment, given that the available bandwidth is large and the PC's display capability is significantly higher than mobile devices, the user can more easily view the whole document and receive updates on all parts of the document. In a mobile network environment, however, bandwidth is relatively scarce and mobile devices might not be able to store and/or view the whole document at one time. To reduce resource consumption, users need to be able to select and work only on desired parts of the document.

2.3. Existing Work

This section presents a review of various existing group editor implementations. The systems that were designed for fixed PC networks, organised by their application architecture, are discussed first followed by existing systems designed for mobile, though not necessarily ad-hoc, networks. For each system, a description of its contributions and shortcomings is outlined. A description of its limitations with regards to the implementation in mobile ad-hoc networks is also discussed in order to underscore the significance of the work presented in this thesis.

2.3.1. Centralised Architecture

In Rendezvous [108], the virtual terminals of the connected users all communicate with one centralized process that controls the application. The central process contains the underlying objects for the application, provides support for session management, and generally facilitates the coordination among users.

Dolphin [124] is another example of a graphical group editor for supporting joint work and brainstorming with users not having to reside at the same place or meeting room. DOLPHIN supports the creation and manipulation of unstructured graphics (e.g., freehand drawings, handwritten scribbles), structured graphics (e.g., hypermedia documents with typed nodes and links), their coexistence, and their transformation. DOLPHIN utilises a centralised architecture to allow multiple distributed clients to share common hypermedia objects stored in the cooperative hypermedia engine server.

Tivoli [96], similar to DOLPHIN, provides whiteboard–like functionality, with an added flip-chart capability to handle multiple sheets that can be printed or saved for later use. However, being group editors that utilise centralised architecture and pessimistic concurrency control, DOLPHIN and Tivoli suffer from the same set of limitations as Rendezvous.

Jupiter [101] is another multi-user, multimedia virtual world intended to support long-term remote collaboration on shared documents, shared tools, and, optionally, live audio/video communication. Jupiter also utilises a centralised architecture: Jupiter's users run the client on their local workstation. It makes a TCP connection to the server, running on a central machine. Jupiter's central server stores the state of virtual objects and executes all of their associated program code, while the clients simply manage their local input/output hardware on behalf of the server and the user.

The use of the centralised architecture makes document consistency easy to manage since the document only resides on the central server. However, as mentioned in section 2.1.2, this architecture has several drawbacks: it has a single point of failure, it imposes relatively high bandwidth consumption, each client has to always be connected to participate in collaboration, and it is not applicable in mobile ad-hoc networks since the presence of a dedicated server is not guaranteed in such environments. Furthermore, with the exception of Jupiter, the above systems utilise a pessimistic concurrency control (locking) such that only one user may provide input and that all others are blocked. This not only diminishes concurrency, the use of locks in mobile, especially ad-hoc, networks imposes additional overhead to manage the locks. Jupiter adopts an optimistic concurrency control as introduced by GROVE [46] where users can edit the document concurrently. The fact that Jupiter utilises a centralised architecture makes concurrency control substantially simpler than its replicated architecture counterpart. However, having utilised centralised architecture, Jupiter still suffers from limitations and drawbacks of centralised architecture as described in section 2.1.2.

Centralised architecture offers a simple collaboration management, and imposes very little resource consumption in the client side. However, due to the limitations of the centralised architecture and the increasing power of personal computers, the replicated architecture is becoming more attractive.

2.3.2. Replicated Architecture

As mentioned in section 2.1.2, replicated architecture can be categorised into: *fully replicated* and *semi-replicated*. A fully replicated architecture replicates the collaboration process and the shared document, and does not require a central server. The semi-replicated architecture also replicates the process and the shared document, but it requires a server for membership, registration, and/or shared objects registry purposes.

Colab [123], an experimental meeting room developed by Xerox PARC, is one of the earliest use of computers to support collaborative work. Colab is designed for small working groups of two to six persons using personal computers connected over a local area network with the aim of making meetings among computer scientists more effective. Colab is a replicated architecture group editor that employs a distributed database: each machine has a copy of the database and changes are installed by broadcasting each modification without any synchronization. Since Colab is designed for a meeting room setting, the participants use verbal negotiation with other group members before altering shared data.

Colab has a few meeting tools: (1) Boardnoter, which closely imitates the functionality of a chalkboard; (2) Cognoter, a tool for organizing ideas to plan a presentation; and (3) Argnoter, a tool for considering and evaluating alternate proposals. As an early work in CSCW, Colab uses a rather naïve concurrency control strategy: if two participants make changes to the same data simultaneously, there is a race to see which change will take effect first, and the result can be different on different machines. Colab relies on the fact that the end results are independent on the order of the concurrent updates and the use of verbal cues of the users to coordinate their behaviour. Furthermore, Colab also uses locking to prevent two users working at the same document part at the same time, hence reducing the ability to work concurrently.

One example of group editors that employ a semi-replicated architecture is DistView [113]. Intended for supporting synchronous collaboration over wide-area networks, DistView supports the building of collaborative multi-window applications allowing some of the user's application windows to be shared with other users at a fine-level of granularity while still keeping other application windows private. DistView uses an object-level replication scheme in which the application and interface objects that need to be shared among users are replicated to keep bandwidth requirements low and to maintain the responsiveness of the groupware system.

A user may *export* a window to the group when s/he observes something interesting. DistView, however, requires a server as the registrar of the shared windows. Exported windows will be registered in the shared window server and other users can inspect the list of remote windows available to be imported. The other users may then individually *import* the shared window to observe and modify its content. When a window is imported, DistView replicates all the objects of the window and the window object itself from an export window manager of the originator site (the site that exports the window) to an import window manager of the destination site (the site that imports the window). For correct replication, the state transfer includes the window type (class), its type-specific internal state, its references to other objects, and its location within its parent window. DistView uses locking mechanisms so that simultaneous interactions by users can be supported, without leading to undesirable or inconsistent results: all user operations must acquire appropriate locks to ensure that interface and application objects, when updated concurrently, lead to correct results and consistent replicas. Since it uses locks and requires a central server for shared window registry, DistView is not well suited to mobile ad-hoc networks.

GroupKit [118], another example of a synchronous group editor that utilises a semi-replicated architecture, is a groupware toolkit that lets developers build applications for synchronous and distributed computer-based conferencing. Its runtime infrastructure consists of three types of processes: registrar, session managers, and conference applications. The registrar is the first and centralised process created in a GroupKit session. There is usually one *registrar* for a community of conference users, and its address is “well known” in that other processes know how to reach it. The session manager is replicated, and once it is created, it connects to the registrar to locate existing conference processes.

The conference application is invoked by the user, managed by the session manager, and it consists of groupware tools such as a shared editor, whiteboard, and group chat. The conference applications utilise remote procedure calls (RPC) to communicate, share information, and trigger program execution between replicated application processes in a session. Based on the belief that no one concurrency control works in all groupware systems [55], unlike DistView, GroupKit does not implement a specific concurrency control to allow the developer to implement the appropriate concurrency control depending on the conference application.

Group editors that employ semi-replicated architecture are not readily suited to mobile ad-hoc networks since they require a server to be present for handling some collaboration functionalities.

One of the earliest groupware editors that utilises fully replicated architecture and supports collaboration for geographically dispersed group members is GROVE (*GR*oup *O*utline *V*iewing *E*ditor) [47]. GROVE is an outline editor intended for use by a group of people simultaneously working on a textual outline. Participants can modify the underlying outline by performing editing operations, such as insert, delete, cut, and paste in the window, they may also open and close parts of the outline (using the small buttons on the left side) or change the read and write permissions of outline items. In addition to displaying views, group windows also indicate who is using the window.

Without the presence of a central server, GROVE could not use traditional concurrency control such as locking and/or transactional processing as those methods require a server to manage the locks and/or the transactions. GROVE, therefore, is the first groupware system that uses the notion of Operational Transformation (OT) [46] to ensure document consistency in the presence of concurrent updates. Since GROVE, operational transformation has been used and developed by various researchers to ensure document consistency in a replicated architecture. The technical aspect and the development of operation transformation will be discussed in more detail in section 3.3.3.

REDUCE (REal-time Distributed Unconstrained Cooperating Editing) [131] is another collaborative editing application that uses fully replicated architecture. Similar to GROVE, it uses operation transformation for its concurrency control. However, the algorithm implemented by REDUCE (either GOT [137] or GOTO [131]) fixes the shortcomings of GROVE's dOPT algorithm. The shortcomings and the detail of the algorithms are discussed in greater detail in section 3.3.3.

An example from another application domain is GroupDesign [74], a multi-user drawing tool for structured graphics that runs in a heterogeneous environment comprising a network of Apple Macintosh computers and Unix workstations. Similar to Cognoter and Argnoter [123], the members of a group can simultaneously edit a diagram. GroupDesign uses a relaxed WYSIWIS (What-You-See-Is-What-I-See) paradigm [122], since a strict WYSIWIS approach would not have allowed users to work independently on different diagram areas. The document is the same for all replicas but each user has his or her individual view of the diagram. For example, users have an independent control over the scroll bars and window placement. Similar to GROVE, GroupDesign uses a replicated architecture: an instance of the application (a replica) runs on the computer of each user. GroupDesign does not use any central process for the coordination of the replicas, nor does it give a special role to the user who first launches a session. However, unlike GROVE, GroupDesign uses simpler concurrency control than the operation transformation approach used by GROVE. As a drawing tool, the events always *commute* or *mask*. For example, if a user moves an object and another user changes its colour, the order of execution of these actions is irrelevant. In other words, events carrying these actions *commute*. On the other hand, if a user changes the colour of an object to red and another user then changes it to green, the corresponding events do not commute. However, if 'change to green' has been received and executed by a replica and 'change to red' arrives later, the latter can simply be discarded. In other words, the second event (in the total order) has masked the first one. Therefore, since events always commute or mask, they are always handled immediately providing the best response time possible for the interface. However, the decision to mask one of the conflicting operations does not preserve all users' intentions. The masked operation is simply discarded, and consequently, the intention is never noticed by other users.

Aiming to preserve both conflicting operations, GRACE (GRAphics Collaborative Editing) [130], an internet-based prototype system developed using the Java programming language, uses the multi-versioning technique to keep both intentions in separate object versions. GRACE has a system architecture resembling the architecture of the (text-oriented) REDUCE system [131], where multiple collaborating sites are directly connected via TCP connections over the Internet. Each collaborating site runs a replicated GRACE process which takes care of operation generation, processing and propagation, and document management. GRACE has a graphics editing interface and uses the multi-versioning technique for consistency maintenance, whereas REDUCE has a text editing interface and uses operational transformation for consistency maintenance. Both multi-versioning technique and operational transformation are useful to ensure document consistency. Multi-versioning technique is discussed in more detail in section 4.3.3.

Recently, a collaborative graphic editor called Draw-together [65] was proposed. Like GRACE, it uses a replicated architecture where each user works on a copy of the document. Local operations are executed on the local copy of the document immediately after their generation and then broadcast to the other sites. When a remote operation arrives at a site, some of the operations that have been performed at that site might be undone and re-executed together with the remote operation in order to satisfy a combined effect of the concurrent operations. The differences between GRACE and Draw-together are the document structures and the way it resolves conflicts. The document structure of Draw-together is not only object-based, but it also shows the hierarchical object grouping. A document consists of pages, each page consists of layers, each layer consists of several composite objects (groups of objects), and each group may consist of either several objects or several other groups. Furthermore, unlike GRACE, it attempts to resolve conflict by using an operation serialisation algorithm based on the reordering of nodes in a graph. Two types of conflicting operations are defined: *real* and *resolvable* conflicting operations; and two directed graphs are introduced to help resolve the conflicting operations. However, the conflict resolution strategy proposed in Draw-together is only applicable to graphic editors. Consequently, Ignat et al [68] has proposed a flexible conflict definition and resolution to handle conflict in generic document editors with hierarchical document structure, which is discussed in section 4.3.2.

Although fully replicated architecture group editors do not introduce a single point of failure, the existing systems assume a known number of participants and do not generally address how dynamic membership events are handled. Although DistView and GroupKit are able to manage session membership, they require a central server to do so; hence it is not readily applicable to mobile ad-hoc networks.

2.3.3. Groupware Systems in Mobile Networks

Sync [98], a Java framework for mobile collaborative application, is based on object-oriented replication and offers high-level synchronization-aware classes that developers may easily tailor to the synchronization needs of their application. The Sync framework employs a centralised architecture that requires a server to do the synchronisation of replicas. Each change in the client side must be synchronised to the server, and the central synchronization server accepts synchronization requests from remote replicas, collects all changes received by the server since the remote replica's last synchronization, and merges them with the replica's changes included in the replica's synchronisation request. Of the merged set of changes, those originating from the replica are applied to the server's replica, and those originating from the server are sent to the remote replica.

Another example of a groupware system in mobile network that employs centralised architecture is QuickStep [120]. The major difference is that Sync allows asynchronous collaboration between mobile users, while QuickStep allows synchronous collaboration between mobile users using handheld devices. QuickStep uses the database paradigm and stores application data as records. Each handheld device has its own local database which contains the application's records. Only the owner can add, change or remove local records. The QuickStep server has a copy of each local database, the mirror database. The mirror database is incrementally updated each time a handheld device is connected to the server.

As opposed to Sync and Quickstep that employ centralised architecture, YCab [22] is a framework proposed for use by collaborative services in a wireless ad-hoc network. YCab offers services such as text chat, shared whiteboard, shared images, and global positioning system navigation. Specifically, Ycab is a collaborative environment and a framework API suited towards ad-hoc networks of small mobile devices. It was designed having realised that conventional collaborative tools are not suited to the demands of portable computers and mobile networks, especially in situations in which no fixed-network infrastructure is present.

YCab has support for decentralised session control and decentralised communication managers. The framework consists of: (1) Communication and service managers, (2) Session and election services, (3) State recovery manager, and (4) GUI components. The framework provides an environment for services on one client to communicate and share information with services on other clients in an ad hoc network. The framework also implements various decentralised protocols, such as session registration protocols, leader election protocols, and state recovery protocols.

While YCab discusses the various implication of mobile ad-hoc network to the requirements of mobile real-time collaborative editing application, such as mobile devices joining and leaving the session, it does not discuss how it maintains the consistency of the shared document in the midst of concurrent updates and out-of-order operations arrival. Furthermore, it relies on a session coordinator for state recovery process and the joining process requires all existing members to be connected. The shortcoming of YCab in handling dynamic membership events is discussed in greater detail in Chapter 5.

DSC [89] is another peer-to-peer groupware system that uses decentralised topology. The DSC system is implemented using the JXTA platform providing a pure P2P architecture and a dynamically created ad hoc network without central control or server. DSC offers a set of protocols and a series of services that let peers find each other, form groups and directly exchange messages across firewalls and NATs [3]. DSC provides three shared objects in terms of a web browser, file viewer and drawing pad as well as a text chat tool. The DSC system software consists of five parts: group agent, message handler, data recorder, shared spaced controller and shared objects.

DSC faces some challenges and limitations. Due to the distributed advertisement publishing and searching mechanism in JXTA, it takes about 1 to 10 minutes to know the existing peers and groups using the search function provided by the JXTA discovery service [89]. DSC employs WYSIWIS mode which restrict users' ability to navigate through different parts of the shared document. Furthermore, it lacks coordination control policies and mechanisms to keep the collaboration harmonious and the document consistent. Chapter 5 of this thesis presents algorithms to handle dynamic memberships in mobile ad-hoc network, and the concurrency control policies to handle different types of conflicts are presented in Chapter 3.

Speakeasy [45] is also a peer-to-peer system built for ad-hoc collaboration. Speakeasy allows users to create *converspace*, a shared space where users can drag and drop any resource to be shared and once a resource is in the converspace, the other peers will be able to access the shared resource from their local converspace view. Architecturally, it is comparable to JXTA as it provides a service to let peers share resource, discover shared resource and access resources from the converspace. Although it supports P2P collaboration and resource sharing, it does not support synchronous collaborative editing.

It is also worth mentioning the use of publish/subscribe systems for achieving collaborative work. Systems such as YACO [25] and MOTION [77] provide services such as file sharing, distributed artefacts, resource sharing and searching, messaging, and system event subscribing. YACO (Yet Another Framework for Collaborative Work) exploits capabilities of the SIENA publish/subscribe system [26] with support for mobile users using MOBIKIT, a support service for mobile publish/subscribe applications [24], for providing services to support collaborative work.

A publish/subscribe system, although providing capabilities to share file and artefacts, is not built to support synchronous collaborative work. It is suitable for asynchronous collaborative work where documents are viewed and edited by one person at one time. Furthermore, it relies on an established network of message routers to distribute messages and artefacts from the publisher to the client or vice versa.

The following Table 2-1 provides the summary of the collaboration systems described above.

Group Editors	**Architecture**	**Features and/or advantages**	**Limitations and/or disadvantages**
Randezvous	Centralised	One centralized process controls the application; hence it is simple to manage.	Single point of failure
Dolphin and Tivoli	Centralised	Supports joint work and brainstorming with users not having to reside at the same place or meeting room.	Single point of failure
Jupiter	Centralised	Document consistency is easy to manage since the document only resides on the central server.	Single point of failure. Relatively high bandwidth consumption. Each client has to always be connected to participate in collaboration. It is not applicable in mobile ad-hoc networks since the presence of a dedicated server is not guaranteed in such environments.
Colab	Replicated	One of the earliest uses of computers to support collaborative work.	Naïve concurrency control strategy: concurrent updates are not handled appropriately;

			hence the document state can be different on different machines.
DistView	Semi-replicated	Supports collaborative multi-window applications. Low bandwidth requirements. Maintains the responsiveness of the groupware system.	It uses locks and requires a central server for shared window registry; hence it is not well suited to mobile ad-hoc networks.
GroupKit	Semi-replicated	A groupware toolkit that lets developers build applications for synchronous and distributed computer-based conferencing	It does not implement a specific concurrency control. It requires a server to be present for handling some collaboration functionalities.
GROVE	Replicated	GROVE is the first groupware system that uses the notion of Operational Transformation to ensure document consistency in the presence of concurrent updates.	The dOPT algorithm used by GROVE is unable to maintain consistency in all scenarios. Further discussion is presented at Chapter 3.
REDUCE	Replicated	The algorithm implemented by REDUCE fixes the shortcomings of GROVE's dOPT algorithm.	The shortcomings and the detail of the algorithms are discussed in greater detail in section 3.3.3.
GroupDesign	Replicated	A multi-user drawing tool for structured graphics. Runs in a heterogeneous environment	The decision to mask one of the conflicting operations does not preserve all users' intentions.
GRACE	Replicated	GRACE has a graphics editing interface and uses the multi-versioning technique for consistency maintenance.	Multi-versioning technique is discussed in more detail in section 4.3.3.
Draw-together	Replicated	Hierarchical document structure.	The conflict resolution strategy proposed in Draw-together is only applicable to graphic editors.
Sync	Centralised	A Java framework for mobile collaborative application	It requires a server to do the synchronisation of replicas. It allows only asynchronous collaboration.
QuickStep	Centralised	QuickStep allows synchronous collaboration between mobile users using handheld devices.	It requires a server to do the synchronisation of replicas.
YCab	Replicated	Supports collaborations in wireless ad-hoc networks.	It does not discuss how it maintains the consistency of the shared document in the midst of

			concurrent updates and out-of-order operations arrival.
DSC	Replicated	The DSC system is implemented using the JXTA platform providing a pure P2P architecture and a dynamically created ad hoc network without central control or server.	DSC employs WYSIWIS mode which restrict users' ability to navigate through different parts of the shared document.
Speakeasy	Replicated	It supports P2P collaboration and resource sharing	It does not support synchronous collaborative editing

Table 2-1 Comparison of various existing collaboration systems

2.4. Summary

Synchronous groupware systems allow two or more users to work on a shared document at the same time regardless of their physical location. There have been many groupware systems developed to support real-time collaborative editing in various environments, ranging from the ones designed for fixed PC networks to the ones designed for mobile networks. In this chapter, the requirements for real-time mobile collaborative editing have been discussed. Although the existing groupware systems can support real-time collaborative editing in fixed PC networks, they have shortcomings with regards to satisfying the requirements for mobile networks.

The shortcomings of the existing groupware systems and how they are addressed in this thesis can be summarised as follows.

1. The existing concurrency controls or the document consistency algorithms either rely on a central server, assume a reliable network, or assume unlimited resources; hence

they are not well suited to mobile ad-hoc architecture. Chapter 3 of this thesis presents a consistency algorithm that is fully replicated, consumes considerably less resources (memory, storage space and processing power) and still works in unreliable mobile networks. Chapter 4 discusses conflicts in real-time mobile collaborative editing and presents a conflict management algorithm to handle the conflict and facilitate conflict resolution.

2. The existing groupware systems either assume a fixed number of participants or utilise a central server to manage session membership, neither of which is suited to mobile ad-hoc network characteristics. Consequently, Chapter 5 presents a mechanism to handle dynamic membership events while still ensuring the consistency of the shared document.
3. The existing replicated architecture group editors assume the whole document is replicated. As mentioned previously in section 2.2.2, this consumes large bandwidth since all updates have to be sent to all participants and mobile devices may not necessarily be able to accommodate a large document. Chapter 6 discusses the idea of dividing the shared document into document partitions and presents a mechanism that allows users to work on desired partitions.

As mentioned in section 2.1.1, this thesis uses the collaboration awareness approach since its focus is on devising algorithms that provide essential functionality in real-time mobile group editors, not developing the collaboration application itself. The algorithms devised in this thesis, however, can be used by developers that wish to use a collaboration transparency approach to leverage single-user application for real-time multi-user collaboration.

Chapter 3

Consistency Management

3.1. Introduction

As discussed in the previous chapter, the replicated architecture is well suited to real-time collaborative editing in mobile ad-hoc networks. In a centralised architecture, the main document is held by the server, and the document is always convergent since the server applies all the updates to the document and all participants retrieve the updated document state from the server. In a replicated architecture, however, since there is no dedicated server, each mobile device maintains a replica (local replica) of the shared document. Consequently, each user has full access to his/her local replica, thus promoting concurrency and interactivity. Each update of a user has to be broadcast to all other participants such that all participants can view the update reflected in their local replicas. Each update is broadcast as an operation that realise the intention of the user who initiates the update, such that when the operation is executed in another document replica, the update is reflected correctly.

Due to the concurrency of the operations generated at each site and the fluctuating bandwidth in mobile networks, operations may arrive at one site in a different order to that in which they arrive at another site. Furthermore, due to concurrency, the operation that arrives at one site may have not been generated from the same document context as the context of the current document replica at the recipient site. The *document consistency* requirement needs to ensure that the end results of all replicas are consistent regardless of the concurrent updates and regardless of the arrival order of those updates. Without proper consistency management, the convergence of the document copies cannot be assured, which means after a certain period of time one site may have a different document state from the others. The *consistency management algorithm* is also commonly known as the *concurrency control algorithm*, and therefore those two terms are used interchangeably throughout this thesis.

This chapter presents the document consistency problem and presents a document consistency algorithm that supports real-time group editors in mobile ad-hoc networks. The proposed consistency algorithm not only supports real time mobile groupware, it is also applicable to non-replicated architecture and it incorporates some novel techniques to improve its efficiency. Furthermore, it also incorporates some corrections to the existing technique so that it is able to ensure document consistency in scenarios where existing algorithms fail to do so. This chapter focuses on presenting a concurrency control mechanism that allows non-conflicting concurrent updates to be applied to the document replicas so as to maintain their consistency. The mechanism and algorithm for handling *conflicting* concurrent updates will be addressed in Chapter 4. The remainder of this chapter is organised as follows: section 3.2 presents a general model of a real-time collaboration system as used by many existing consistency management algorithms [46, 126, 142, 135, 136]; section 3.3 discusses existing consistency algorithms, their contributions and their shortcomings in regards to implementation in real-time mobile collaborative editing; section 3.4 presents and provides comments on the two most recent work in document consistency; section 3.5 presents the proposed consistency algorithm including the proposed operation transformation rules; section 3.6 shows the results of the performance evaluation of the proposed algorithm; and finally, section 3.7 concludes the chapter and outlines some future work.

3.2. Group Editors Model

In replicated architecture, the shared document that users are working on is replicated so that each user works on the local replica that exists on each site. A user at each site works on the shared document by applying an operation to the local replica, each of which will change the document state. To allow all sites to get the latest state of the document, any operation generated at one site has to be broadcast to all other sites. A *local operation* is an operation generated by the local site, whereas a *remote operation* is an operation generated by another site and received as a result of the operation broadcast. Due to concurrency of the generated operations and the unpredictable fluctuating network delay, remote operations may arrive out of order. Each participant has to process the operations (local and remote) in such a way that the document replicas are consistent and the intentions of the users are respected.

This section presents the generic model of real-time collaborative editing systems. Section 3.2.1 describes the model of the document to be used throughout this thesis. The document model aims to be as generic as possible such that the algorithms devised in this thesis can be used in most application domains. Section 3.2.2 describes the operation model to represent the updates made by users in a collaboration session.

3.2.1. Document Model

All documents can be considered to be composed of objects. A simple text document consists of letters, an XML document consists of nodes, and a complicated multimedia document may consist of hundreds or even thousands of graphical and composite objects. Users work on a document by adding objects to the document, modifying the existing objects in the document, and deleting objects from the document. An addition, a modification, or a deletion of an object may or may not affect the existing document objects. For example, in a text document (Figure 3-1), when a user inserts a letter on the text at a certain position, all letters after that position will be shifted to the right; and if a user deletes a letter, all other letters after it will be shifted to the left. This is called a *dependent-object document*, where an operation done to an object affects the positions or attributes of existing objects.

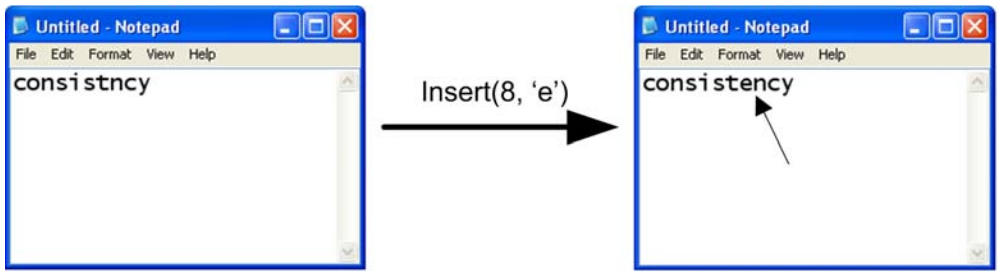


Figure 3-1 Dependent-object document

An *independent-object document*, on the other hand, is a document where an operation done to an object (insertion, deletion, or modification of an object) in the document will not affect the existing objects. An example for this type of document is a simple drawing document (Figure 3-2), such as Microsoft® Paint document. If a user adds an object to the document, the other objects will stay where they are, and if there is an overlap, then one object will be on top of the other object.

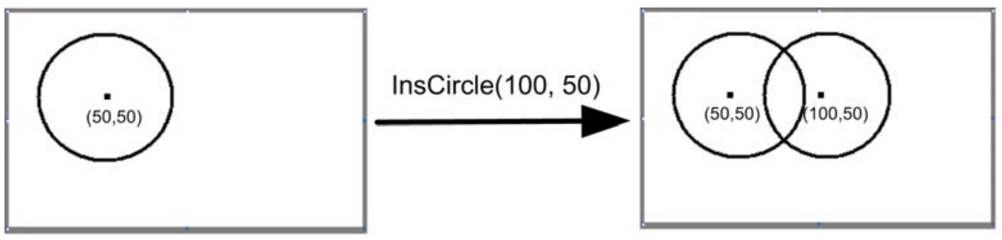


Figure 3-2 Independent-object document

In a dependent-object document, applying an operation to an object might affect some other objects. On the other hand, applying an operation to an object in an independent-object document will not affect any other object. Thus, if an algorithm works for dependent-object documents by considering the operation effect on one object on another, the same algorithm will also work for independent-object documents by excluding the operation effect. Consequently, independent object documents can be considered as a subset of dependent-object documents, and an algorithm that works for dependent-object documents can be easily adjusted to cater for independent-object documents. Therefore, in this thesis, the focus is on developing algorithms that work for the more challenging case of dependent-object documents.

Much of the existing work on consistency management algorithms [46, 126, 132, 142] uses a plain text document model to develop the algorithm. While it is simple and serves its purpose for developing the algorithm, it does not reflect the data/document structure of the current pervasive applications. Furthermore, it is not adequate to reflect the two main types of conflicts: *exclusive* and *non-exclusive* conflicts (readers can refer to section 4.1 for conflicts definitions). On the other hand, object based documents are common at the present time, and therefore this thesis addresses algorithms that work both on simple character-based text documents and on complex object based documents.

Therefore, an object-based text document is assumed throughout this thesis due to the following properties.

- It is general enough to represent most application domains.

- It is simple enough to provide an understanding of collaborating on object based documents.
- It is broad enough in scope to show the two main types of conflicts that can occur in real-time collaborative editing (refer to section 4.1 for conflicts definitions), and consequently, how they are handled by the proposed algorithm.

The text document consists of character objects that are identified by an object identifier (*objId*) and have a number attributes, such as its position in the document (*pos*), its size (*fontSize*) and its colour (*fontColor*). The *pos* attribute, with 1 being the first position, is dependent on the position of other objects and thus the insertion or deletion of one object might affect the position of another, meaning that this is a *dependent object document*. Changing the other attributes, however, does not affect other objects and thus the document also shows *independent object* characteristics, thereby allowing the algorithm presented in this thesis to be tested for both dependent and independent object cases. It could be argued that an object-based text document is too simple and is not as useful as more pervasive applications. However, as the focus of this thesis is to offer essential functionalities to support group editors, not to develop a pervasive group editing application, object-based text documents are adequate to demonstrate the proposed algorithms. Although the proposed algorithms are designed with object based text documents as the example, they can be adjusted to support collaboration on more complex documents.

The notion of a multi-level document structure in a general text editor has also been used in [67, 37, 134, 147], where a document consists of several chapters, each chapter consists of several paragraphs, each paragraph consists of several sentences, which consists of several words and, at the bottommost of the level, characters. Unlike their work, this thesis assumes a simple single-level document (a text document consists of many character objects). Although, multi-level document structure may represent the document better, it can be argued that the way conflicts and concurrency are handled are similar to the single-level document structure. Section 3.3.3 discusses the proposed treeOPT algorithm by Ignat et al. [67] and furthermore, section 6.3 (as part of the Chapter 6 – document partitioning) discusses and compares the document structuring of treeOPT with the document structuring of the proposed document partitioning algorithm.

3.2.2. Operation Model

A user works on a document by adding, modifying, and deleting objects of the document. Every update intended by the user is realised by an operation. The term 'user intention' and 'operation intention' are used interchangeably throughout this thesis to represent the document update intended by the user who generates the operation. In an object based text document, the three generic operation primitives used in the document are:

- *insert(objId, pos, char, attrSet)*: inserts character *char* with object id *objId* at position *pos* with an initial attribute set of *attrSet*,
- *delete(objId, pos)*: deletes object with object id *objId* at position *pos,* and

- *changeAttr(objId, attr, value)*: change an attribute *attr* of object *objId* to a new value of *value*.

Since *changeAttr* operation commutes with both *insert* and *delete* operations (the effect of a *changeAttr* operation does not change regardless of whether it is executed after or before an *insert* or *delete* operation), it is not included in the discussion in this chapter. Furthermore, the position of the character (*pos*) and the character itself (*char*) are the two parameters that should be taken into account in the presence of concurrent operations. On the other hand, the object id and the attribute set are not of interest in the concurrency control discussion of this chapter since they are not affected by the concurrency of operations except in the case of conflicting *changeAttr* operations when two users changing the attribute set of an object to two different values. Therefore, without losing generality, the *insert* operation can simply be represented as *insert*(*pos, char*) and the *delete* operation can simply be represented as *delete*(*pos*), while the object id and attribute set parameters, and the *changeAttr* operation, will be included in Chapter 4 where a conflict management mechanism for handling conflicting *changeAttr* operations is discussed.

Figure 3-3 illustrates a simple diagram of operation execution and operation propagation from one site to another. First, the user at site 1 makes a change on the document by inserting a character ’X’ after ’A’. This is realised by an operation *insert*(2, ’X’). This operation is then sent to site 2 so that site 2 can make the same update to its document. Second, the user at site 2 deletes character ’B’ from the document, by executing an operation *delete*(3). This operation is then sent to site 1 so that site 1 can apply the same change to its local replica.

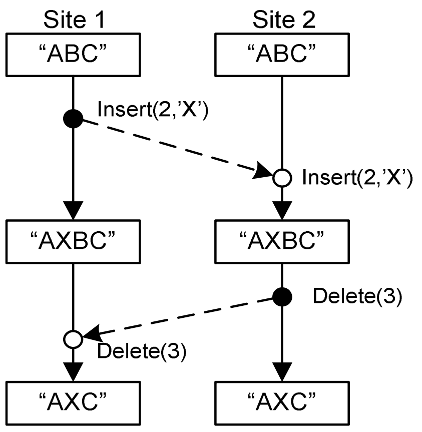


Figure 3-3 Document operation

Suppose op_i is an operation generated at site S_{op_i} and op_j is generated at site S_{op_j}, there are two possible relationships between op_i and op_j - one operation *causally precedes* another, or they are *concurrent* to each other [80]. One operation causally precedes another if the former is executed at a site before the generation of the latter at the same site. They are concurrent if neither of them precedes the other. Formal definitions of both relations are as follows.

Definition 3-1. *Causal precedence operations relation "→"*

op_i causally precedes op_j ($op_i \rightarrow op_j$) iff:

- $S_{op_i} = S_{op_j}$ and *op_j* is generated after *op_i* (Figure 3-4a), or
- $S_{op_i} \neq S_{op_j}$ and S_{op_j} has received and executed op_i before generating op_j (Figure 3-4b), or
- There exists an operation op_k, such that $op_i \rightarrow op_k$ and $op_k \rightarrow op_j$ (Figure 3-4c and Figure 3-4d).

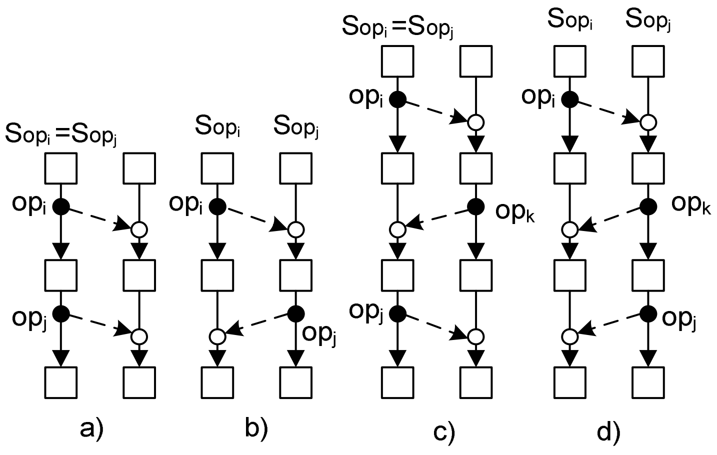


Figure 3-4 Causal precedence

Definition 3-2. *Concurrent operations relation "||"*

As illustrated in Figure 3-5, op_i is concurrent to op_j ($op_i \parallel op_j$) iff:

- $S_{op_i} \neq S_{op_j}$, and
- S_{op_i} has already generated op_i before receiving remote operation op_j, and
- S_{op_j} has already generated op_j before receiving remote operation op_i.

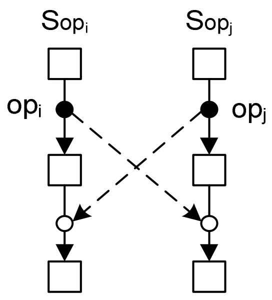


Figure 3-5 Concurrent operations

Additionally, another type of relation – the ‘happened before’ relation – has also been defined for events in distributed systems. This relation can be deduced by examining the operations’ generation time, knowing that each operation is generated by a particular site at a specific time. By looking at the physical time (real clock) they are generated, one can infer which operation ’happened before’ the other. However, such clocks are not guaranteed to be accurate and there are problems on how to synchronize the real clocks of the sites in a distributed system. Alternatively, the ’happened before’ relation can be determined by using *Lamport’s logical clock* [80]. The use of Lamport’s logical clock to determine the ’happened before’ relation and the use of *state vector* to determine the ‘causal precedence’ relation of two operations are described in the following subsections.

Lamport’s Logical Clock

Lamport [80] describes a distributed algorithm for synchronizing a system of logical clocks which can be used to totally order events. Each event (i.e. operation) is timestamped with a logical clock C, and for any events a, b: if $a \rightarrow b$ then $C(a) < C(b)$. This is defined by Lamport as the *Clock Condition*. Each site S_i maintains a logical clock C_{S_i} that will be assigned to any operation generated by that site. If site S_i generates an operation op_i, op_i will be timestamped with $C(op_i)$ where $C(op_i) = C_{S_i}$. Let us consider the illustration in Figure 3-4b. If S_{op_i} is S_1 and S_{op_j} is S_2, with C_{S_1} and C_{S_2} are their logical clocks respectively, op_i and op_j will be timestamped $C(op_i)$ and $C(op_j)$ respectively. Following the Clock Condition, since $op_i \rightarrow op_j$, therefore $C(op_i) < C(op_j)$. To ensure the Clock condition is satisfied, the following two rules must be followed:

1. Site S_i increments C_{S_i} after generating each operation. If op_i and opj are generated by S_i, and op_i is generated before op_j, op_i will bear a timestamp less than op_j ($C(op_i) < C(op_j)$) because C_{S_i} is incremented after op_i is generated.
2. Suppose op_j is an operation generated by S_j bearing a timestamp $C(op_j)$. When site S_i receives op_j, site S_i sets C_{S_i} greater than or equal to its present value and greater than $C(op_j)$. This is to make sure that any other future operation will bear a greater timestamp.

A total ordering scheme can be achieved simply by ordering the events based on their logical clock. If two operations bear the same logical clock values, the total ordering of the sites are used to break ties. The total ordering of sites can be based on the site IDs, site IP addresses or any other unique attribute of sites. For example, if site ids are used, then $S_i < S_j$ if $id(S_i) < id(S_j)$. The following definition (Definition 3-3) outlines the total ordering relationship of two operations.

Definition 3-3. *Total ordering relation "⇒"*

Let op_i be generated by S_i and op_j by S_j , then $op_i \Rightarrow op_j$ if and only if: (1) $C(op_i) < C(op_j)$, or (2) $C(op_i) = C(op_j)$ and $S_i < S_j$.

Although the logical clock can be used to totally order operations, it can not be used to determine causality. The clock condition is not bidirectional: while $a \rightarrow b$ means that $C(a) < C(b)$, $C(a) < C(b)$ does not necessarily mean that $a \rightarrow b$.

Figure 3-6 depicts how Lamport's logical clock cannot be used to determine causality and concurrency. It is obvious that $op_1 \rightarrow op_3$ and $op_2 \rightarrow op_4$, thus $C(op_1) < C(op_3)$ and $C(op_2) < C(op_4)$. Even though $C(op_1) < C(op_4)$, op_1 does not causally precede op_4 ($op_1 \nrightarrow op_4$). The same case also applies for op_2 and op_3. Furthermore, while op_1 and op_3 are concurrent to op_2 and op_4, there is no way to infer concurrency using the logical clock. In other words, Lamport's logical clock is useful to determine the total ordering relationship of operations but not their causality relationship. To detect causality and concurrency, the *state vector* technique should be used instead.

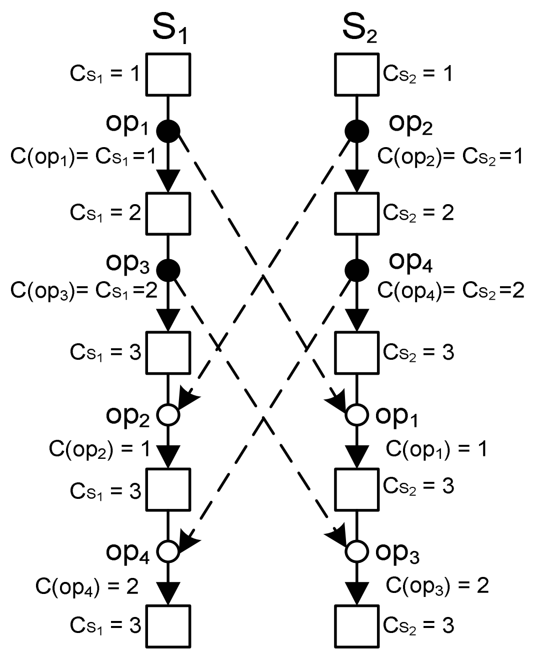


Figure 3-6 Logical clock unable to detect causality and concurrency

State Vector

A state vector, a modification to the clock vector introduced by Mattern [90], is an N-sized vector where N is the number of the participating sites. Each site S_i maintains a state vector $V_{S_i} = (V_{S_i}[1], V_{S_i}[2], ..., V_{S_i}[N])$, where $V_{S_i}[j]$ holds the number of operations generated by site S_j that have been executed by site S_i. For example, if site S_i has already received and executed 3 operations generated by site 2, then $V_{S_i}[2] = 3$. The size of a state vector is the same as the number of participating sites, therefore the more sites participate, the bigger the size of the state vector.

Each operation is piggybacked with the state vector of the generator site to signify the state when the operation is generated. If op_i is generated by site S_i, then op_i will bear a state vector V_{op_i}, which is equivalent to V_{S_i} right before V_{S_i} generates op_i.

Figure 3-7 illustrates the use of the state vector. Initially, site S_1 and S_2 have not executed any operations and their state vectors V_{S_1} and V_{S_2} are equal (0, 0). Operations op_1 and op_2 are the first operations generated by each site, thus they bear the same state vector $V_{op_1} = V_{op_2} = (0, 0)$. After generating and executing op_1, site S_1 updates its state vector with $V_{S_1}[1] = V_{S_1}[1] + 1$, which indicates that S_1 has just executed an operation generated by S_1 ($V_{S_1} = (1, 0)$). Then S_1 generates another operation op_3 and op_3 bears a state vector $V_{op_3} = V_{S_1} = (1, 0)$. Since op_1 is generated by S_1, the fact that $V_{op_3}[1] > V_{op_1}[1]$ indicates that op_3 is generated after the execution of op_1, or in other words, op_1 causally precedes op_3. After receiving op_2, S_1 generates op_4 with $V_{op_4} = (2, 1)$, which means op_4 is generated when S_1 has already executed two operations from S_1 (op_1 and op_3) and one operation from S_2 (op_2). Thus, op_4 happens after op_1 and op_3, deduced from $V_{op_4}[1] > V_{op_1}[1]$ and $V_{op_4}[1] > V_{op_3}[1]$ respectively. Operation op$_4$ also happens after op_2 since $V_{op_4}[2] > V_{op_2}[2]$. Therefore, operation op_i causally precedes op_j ($op_i \rightarrow op_j$) iff $V_{op_i}[S_{op_i}] < V_{op_j}[S_{op_i}]$, where S_{op_i} and S_{op_j} are the sites that generate op_i and op_j respectively. Consequently, op_i and op_j are concurrent if $op_i \not\rightarrow op_j$ and $op_j \not\rightarrow op_i$.

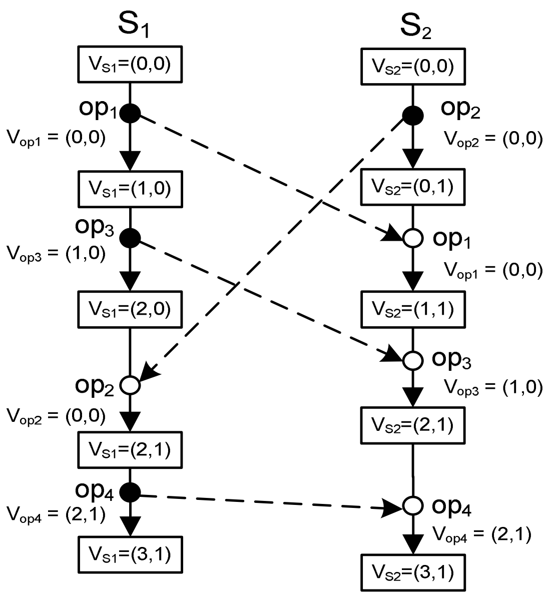


Figure 3-7 State vector

Unlike Lamport's logical clock, the state vector technique is able to detect causality and concurrency. The state vector technique has also been used by REDUCE [137] to determine total ordering relation of operations by comparing the total of the state vector elements. Formal definitions of both relations are as follows.

Definition 3-4. *Causal precedence operations relation "→"*

If op_i and op_j are operations generated by S_i and S_j respectively, op_i causally precedes op_j ($op_i \rightarrow op_j$) iff $V_{op_i}[i] < V_{op_j}[i]$

Definition 3-5. *Total ordering relation*

Let op_i and op_j, be two operations generated at sites S_i and S_j respectively. If V_{op_i} and V_{op_j} are the state vectors of op_i and op_j respectively, and $sum(V_{op_i})$ and $\mathrm{sum}(V_{op_j})$ are the sum of the elements of V_{op_i} and V_{op_j} respectively, then $op_i \Rightarrow op_j$ iff:

- $sum(V_{op_i}) < \mathrm{sum}(V_{op_j})$, or
- $sum(V_{op_i}) = sum(V_{op_j})$ and $S_i < S_j$.

It is worth mentioning that in order to improve scalability, Sun et al. [129] proposed a technique to compress the state vector from a variable size N to a fixed size 2. However, the proposed technique relies on the presence of one site that acts as a notifier site (S_0). The size of the state vector maintained by S_0 is N while the other sites hold state vectors of size 2. Every operation generated from other sites has to be sent to S_0 before being propagated to other sites by S_0. This technique not only introduces a single point of failure, it also means that S_0 needs to have adequate processing power to handle all operation propagations. Therefore, this technique is not suitable for group editors in mobile ad-hoc networks.

3.3. Related Work

There has been much existing work in consistency management (or concurrency control) algorithm for collaborative editors, ranging from the *pessimistic* concurrency control to the *optimistic* counterpart. The pessimistic concurrency control (locking) is preventative where the shared document (or a particular part of the shared document) can only be modified by one person at a time. The optimistic concurrency control, on the other hand, considers that users rarely modify the same part of the document at the same time, hence allowing users to modify any part of the document at any time. When two or more users modify the same part of the document, the algorithm makes sure that the intentions of the users are preserved and reflected consistently in all document replicas. The pessimistic concurrency control (locking) is discussed in section 3.3.1 along with its limitations, followed by the optimistic concurrency control in section 3.3.2. Section 3.3.3 discusses a commonly used optimistic concurrency control, namely Operational Transformation (OT), its development and the limitations of the existing OT-based algorithms.

3.3.1. Pessimistic Concurrency Control (Pre-Locking)

Locking is a pessimistic concurrency control that prevents conflicts in distributed systems and database systems by prohibiting concurrent updates on shared data objects. The traditional locking is also called *pre-locking* meaning that whenever a user wants to edit a part of the document, s/he has to request and be granted an exclusive lock for that part. Pre-locking has been applied to various group editors for consistency management [94, 97, 100, 108, 113]. However, pre-locking is undesirable for the following reasons. Firstly, it imposes overheads in the lock requesting, granting and releasing procedures, especially in replicated architectures where there is no machine dedicated to managing the lock(s).

Secondly, it diminishes concurrency since users cannot modify the locked part of the document. Although the concurrency level can be increased by using a finer lock granularity, the finer the lock granularity, the more locks need to be managed creating more overhead in lock management. Thirdly, although locking prevents conflicting updates from occurring, it has not prevented divergence in a document where the objects are not independent [133]. Finally, locking is not suitable for mobile networks as the frequent disconnections in mobile networks can either prevent applications from obtaining locks on data objects or prevent them from releasing the locks for long periods of time [98].

3.3.2. Optimistic Concurrency Control

Optimistic Locking

A number of variants of locking have been developed by various researchers, including compulsory optimistic locking [55], optional optimistic locking [32], shared locking [133], tentative optional locking [127], and post locking [149]. They are categorised as optimistic locking: either the user does not need to explicitly request a lock or the user does not have to wait for the lock to be granted before s/he can edit a particular part of the document. Due to its non-blocking and responsive nature, optimistic locking is regarded as better suited to an environment where communication latency is high but conflicts are rare [55]. This section discusses optimistic locking, optional optimistic locking, shared locking, and tentative optional locking. Post locking will be discussed in

Chapter 4 since it is not used for concurrency control purposes, but is directly related to the creation of document or object versions in the occurrence of a conflict.

In compulsory optimistic locking [55], if the locking request is successful, the user is able to continue editing the object. If the locking request finally fails, the user is not allowed to continue editing this object, and what they have done while waiting for the lock will be undone. Optional optimistic locking [32], on the other hand, allows users to update any unlocked object without necessarily requesting a lock on it. If no lock is held for that object, the update is valid. If, however, there is a lock on that object, the update is undone. Although they are considered as optimistic and promote more concurrency, both approaches above still need a robust protocol for lock requesting/granting or for checking the availability of the lock on a particular object. This protocol imposes additional overheads if applied in mobile networks, and it also requires all sites to be online for the protocol to work successfully.

In an effort to reduce message roundtrips in the lock requesting and granting process, shared locking uses a local locking table to facilitate the process of checking the availability of the lock. In shared locking [133], whenever a user wants to edit a part of the document, s/he generates a locking operation. If the locking operation does not conflict with any locks in the locking table, the locking operation passed the check and it is sent to all other sites. The user can then start editing the part of the document. If the locking operation conflicts with any locks in the locking table, the locking operation does not pass the check, hence the user cannot edit that part of the document. Upon receiving the locking operation (remote locking operation) from another site, the locking operation is executed and stored in the local locking table. The remote locking operation is always valid, but if it conflicts/overlaps with any of the current locks in the locking table, the conflicting/overlapping locks becomes *shared locks* where both owners can edit the overlapping region concurrently.

Tentative optional locking [127] works similarly to shared locking. The only difference is the way it handles the overlapping locking operations. Instead of using shared locks where both owners of the overlapping locking operations are allowed to update the document concurrently, only one of the overlapping locking operations is accepted (the one with higher priority), while the other is rejected. If there have been concurrent updates on the document due to two or more users concurrently locking and updating the same part of the document, all the updates are preserved (none of them are nullified or undone automatically), and the user whose lock is finally committed decides the end result of the locked region (s/he might incorporate some of the updates from other users if s/he thinks the updates are appropriate). Tentative optional locking has the following drawbacks: it still needs to wait for the lock to be resolved (if there are concurrent locks on the same region) to know whether their updates are to be kept or to be undone, and it requires additional operational transformation based rules for the locking operations to make sure the concurrent locking operations are applied consistently at all sites.

ORESTE

An optimistic concurrency control algorithm for distributed groupware applications called ORESTE has been proposed by Karsenty et al. [73] to ensure consistency of document replicas by ordering the concurrent operations. Similar to operational transformation based algorithms, local operations are immediately executed at the local site to ensure the responsiveness of the application. Unlike the operational transformation approach, however, each remote operation that arrives is executed immediately without waiting for it to be causally ready (i.e. all preceding operations have been executed). It uses Lamport's logical clock for total ordering and undo-redo mechanism to execute out-of-order operations. A remote operation that arrives at one site is executed as soon as possible even if the site has not executed some operations that 'happened before' it. The basic idea of the algorithm is that if the operation arrives out of order, then the operations in the history that 'happened after' the arriving operations will be undone. Then the arriving operation is executed before redoing the undone operations. The operations will eventually be executed at the same order at all sites. Suppose in Figure 3-8 there are four operations, op_1, op_2, op_3 and op_4 with the total order of the operations such that $op_1 \Rightarrow op_2 \Rightarrow op_3 \Rightarrow op_4$. When op_2 arrives at site S_1, site S_1 needs to undo op_3 and op_4 since they 'happen after' op_2. After undoing the two operations, op_2 is then executed before the undone operations are redone.

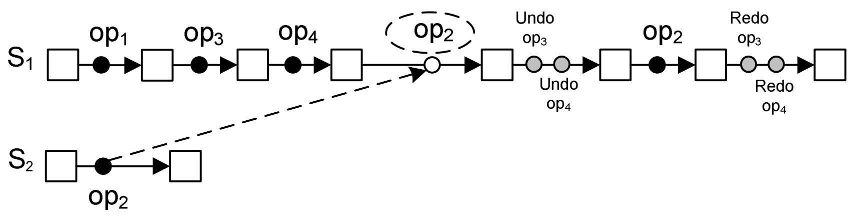


Figure 3-8 Undo and redo in ORESTE

Basically, if an operation op_j happens to be executed earlier than op_i at one particular site ($op_i \Rightarrow op_j$), op_j has to be undone before op_i gets executed and then op_j is redone after the execution op_j. However, the undo and redo of the operations may violate user intentions. The newly inserted operation may alter the context of the document, thus the redone operations may not represent the original intentions of the users. Furthermore, messages arriving out of order is a normal case, thus operations need to be undone and redone quite frequently. Not only does this cause the user experience to suffer, it also consumes processing power since it needs to undo and redo operations. The more operations generated locally before receiving a remote operation, the greater the probability of undoing and redoing operations. Another drawback of ORESTE is that Lamport’s clock cannot be used to detect causality and concurrency (refer to section 3.2.2). Furthermore, ORESTE does not provide any way to preserve the user intention. Consequently, the document might end up with a state that violates the user intention. In the example illustrated by Figure 3-9, the user at site 2 intends to insert ’A’ between ’E’ and ’T’, realised by operation *op2* = *insert*(3,’A’). However, due to the undo and redo of the operation without user intention preservation, the document ends up with a state that violates the user intention (character ’A’ is eventually inserted between ’R’ and ’E’ instead). Therefore, ORESTE is not applicable to dependent-object documents.

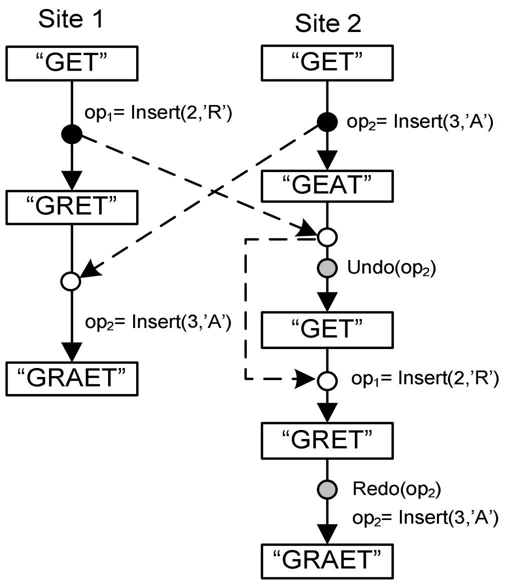


Figure 3-9 User intention not respected

ORESTE, however, is one of the earliest algorithms that tries to conserve device storage space by regularly trimming the operation history. However, since ORESTE is designed for independent-object documents, it simply deletes the old (obsolete) operations from the history. History trimming in dependent-object documents is much more complicated since an operation can only be removed from the history if all operations that are concurrent to it have been executed locally. A more detailed discussion on history trimming for dependent-object document is presented in section 3.5.1.

3.3.3. Operational Transformation Framework

As an alternative to the locking approach, operational transformation was first introduced by Ellis and Gibbs [46] in the dOPT algorithm used by GROVE to allow concurrent updates on document replicas without using locks and without the need for a coordinator site. Operational transformation is attractive to real-time collaborative editing for several reasons: it is suited to replicated architecture as it does not require a central coordinator; it allows users to concurrently edit the same part of the document; and it does not require locks, hence does not impose unnecessary overheads in requesting/granting the lock.

In the operational transformation framework, each site typically goes through the following phases:

1. *Local operation generation*

   When a user modifies his/her local replica, a *local operation* is generated to realise the intention of the user. To ensure responsiveness, the local operation is executed immediately on the local replica so that the user can immediately view the effect of his/her modification on the local document. The operation is then stored in the operations history. This history is necessary since some of the executed operations are necessary for transforming incoming operations.

2. *Operation broadcast*

   The generated local operation is then broadcast to all other participants to notify them of the update made to the document. Each local operation that is broadcast becomes a *remote operation* to other sites.

3. *Remote operation reception*

   The remote operation being broadcast by a site will eventually arrive at another site. Once a remote operation is received, it may not be executed immediately. Due to concurrency, the remote operation might be put into a remote operation queue waiting for execution in the correct order.

4. *Remote operation execution*

   Once a remote operation in the queue is ready to be executed, it is applied on the local document replica. The remote operation is also stored in the operations history.

Whenever a remote operation arrives at a site, the site has to process and apply the operation to its local replica in such a way so as to preserve the intention of the user who generates the operation as well as to ensure that the document replicas are consistent at all sites.

The operational transformation framework adopts the following consistency model [131, 135]: a real-time collaborative editing system is said to be consistent if it satisfies the following consistency properties:

1. **Convergence property**. It requires that all copies of the same document are identical after executing the same collection of operations.
2. **Causality preservation property**. It requires that, for any pair of operations O_i and O_j, if $O_i \rightarrow O_j$, then O_i is executed before O_j at all sites.

3. **Intention preservation property.** It requires that, for any operation *O*, the effects of executing *O* at all sites are the same as the intention of operation *O,* and the effect of executing *O* does not change the effects of other concurrent operations.

dOPT

Operational transformation was first used in the dOPT algorithm [46]. In dOPT, Ellis et al. defined two properties that have to be satisfied by a synchronous groupware system. The first property is called the *precedence property* (or also known as *causality preservation*). It states that if operation *o* causally precedes *p*, then at each site *o* is executed before *p*, and dOPT uses a state vector to achieve this property (section 3.2.2 discusses how state vectors can be used to determine causality of two operations) . When a remote operation arrives at a site, the site has to wait until the remote operation is *causally ready* before it can be executed. An operation is causally ready at a particular site if all other operations that precede it have been executed at that site. Figure 3-10a shows that the document states are consistent when operations arrive in order, while Figure 3-10b shows otherwise. Therefore, causality preservation is important to maintain consistency.

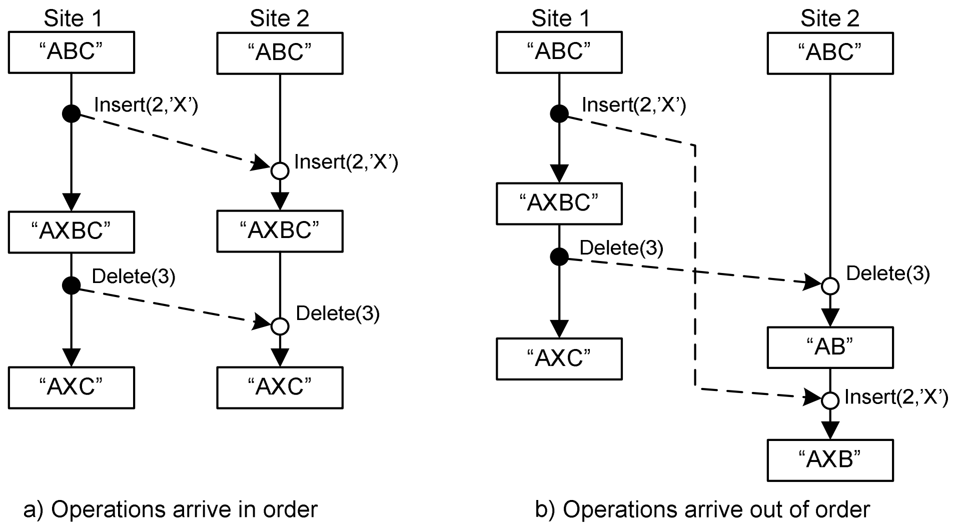


Figure 3-10 Causality preservation

The second property is the *convergence property*, which states that the document replicas are identical at all sites when all sites have executed all generated operations. Operation transformation is used by dOPT to ensure convergence, where the received remote operation is *transformed* against all concurrent operations in the history before it is executed. Consider the example of Figure 3-11a, where two sites generate concurrent operations on the same object "ABC". Site 1 generates op_1 = *insert*(2, 'X') with the intention of inserting 'X' between 'A' and 'B', and site 2 generates op_2 = *delete*(2) with the intention of deleting 'B'. When this operation is broadcast and executed at site 1, the new state is "ABC", which is not what was intended by op_2. To preserve the intention of op_2, operation *delete*(2) should be transformed to become *delete*(3), since 'B' is now in position 3 after site 1 executes op_1 (Figure 3-11b).

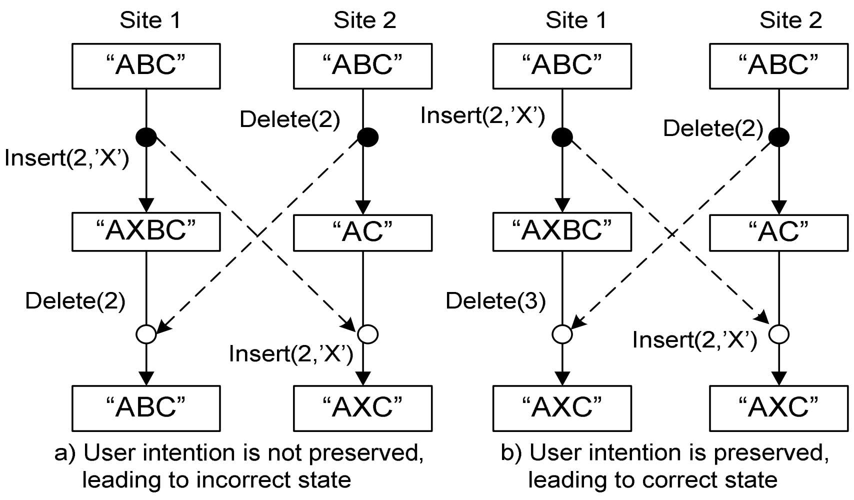

Site 1
Site 2
Site 1
Site 2
"ABC"
Insert(2,'X')
Delete(2)
"AXBC"
"AC"
Delete(2)
Insert(2,'X')
"ABC"
"AXC"
Delete(3)
"AXC"
a) User intention is not preserved, leading to incorrect state
b) User intention is preserved, leading to correct state

Figure 3-11 Document convergence

Forward transformation (*FT*) is used to transform an operation to include the effect of any other previously executed operations. Forward transformation is also known as Inclusive Transformation (*IT*) [132]. Let O_i be a document state and $O_i \bullet op$ is the document state after applying *op* on O_i. Transforming operation op_2 against op_1 means transforming op_2 into its variant $op_2^{op_1}$ such that the effect of op_2 on O_i is the same as the effect of $op_2^{op_1}$ on $O_i \bullet op_1$. It is defined as follows:

$$FT\,(op_2, op_1) = op_2^{op_1}\text{, such that } Intention\,(op_2^{op_1}, O_i \bullet op_1) = Intention\,(op_2, O_i)$$

Forward transforming an operation against a sequence of operations (*L*) simply means transforming the operation against the operations in the sequence consecutively. It is defined as follows:

$$FT\,(op, L) = FT\,(\ldots\,(FT\,(FT\,(op, L[1]),L[2]), \ldots L[\mathrm{n}]) = op^{L}$$

In Figure 3-11b, operation *delete*(2) is transformed forward in site 1 to become *delete*(3) to take into account the effect of the concurrent operation *insert*(2,’X’) that was previously executed. The intention of *delete*(2) on ”ABC” is the same as the intention of *delete*(3) on ”AXC”, which is deleting character ’C’. Operation *delete*(3) is defined by the following transformation: *FT*(*delete*(2), *insert*(2,’X’)) = *delete*(3).

Each operation has to be able to be transformed against all other operations. In dOPT, only two operations are defined, thus there are 2×2 possible forward transformations. Given two concurrent operations op_1 and op_2, with priority p_{op_1} and p_{op_1} respectively, the forward transformation rules are outlined in Figure 3-12. The priority of the operation can be arbitrarily determined, such as using the site id or its timestamp, as long as it is consistent across all sites and it is unique, such that one can determine which operation has more priority over the other.

Since this algorithm does not implement any locking, the responsiveness is good and it is possible for users to modify the document concurrently. The operations initiated by the users are performed immediately on their respective sites and users can modify the document anytime. This algorithm is fully distributed so that when any one site fails, other sites can resume the collaboration without interruption.

An operation does not need to be transformed against the operations that precede it since it has already had the effect of those operations. Thus, when a site receives a remote operation, the site only needs to forward transform the operation against concurrent operations in the history.

```
$FT(op_1 = insert(x_1, a_1), op_2 = insert(x_2, a_2)) = op_1'$, where
    if $x_1 > x_2$ then $op_1' = insert(x_1 + 1, a_1)$;
    if $x_1 < x_2$ then $op_1' = insert(x_1, a_1)$;
    if $x_1 = x_2$ then
        if $a_1 = a_2$ then $op_1' = id$;
        if $a_1 \neq a_2$ then
            if $p_{op_1} > p_{op_2}$, $op_1' = insert(x_1, a_1)$;
            if $p_{op_1} < p_{op_2}$, $op_1' = insert(x_1 + 1, a_1)$;
        endif;
    endif;

$FT(op_1 = insert(x_1, a_1), op_2 = delete(x_2)) = op_1'$, where
    if $x_1 > x_2$ then $op_1' = insert(x_1 - 1, a_1)$ else $op_1' = insert(x_1, a_1)$;

$FT(op_1 = delete(x_1), op_2 = insert(x_2, a_2),) = op_1'$, where
    if $x_1 \geq x_2$ then $op_1' = delete(x_1 + 1)$ else $op_1' = delete(x_1)$;

$FT(op_1 = delete(x_1), op_2 = delete(x_2)) = op_1'$, where
    if $x_1 > x_2$ then $op_1' = delete(x_1 - 1)$;
    if $x_1 < x_2$ then $op_1' = delete(x_1)$;
    if $x_1 = x_2$ then $op_1' = id$;
```

Figure 3-12 Forward transformation rules used in dOPT

The major drawback of dOPT is that it naively transforms every remote operation against concurrent operations in the history without taking the user intention into account. A few researchers, such as Guerraoui et al. in [59], have proven that dOPT is incapable of maintaining document consistency under some scenarios, particularly when operations are not generated at the same state. Consider the example in Figure 3-13 where site 1 generates two operations op_1 and op_3 and site 2 generates op_2. When site 2 receives op_1, it is forward transformed against op_2 since $op_1 \| op_2$, and $op_1^{op_2}$ is executed instead ($op_1^{op_2} = FT(op_1, op_2)$). Consecutively, when site 2 receives op_3, by the same reason, it is forward transformed against op_2 and $op_3^{op_2}$ gets executed in site 2. Meanwhile, site 1 transform op_2 against op_1 and op_3 to become $op_2^{op_1 \bullet op_3}$ ($FT(op_2, op_1 \bullet op_3) = op_2^{op_1 \bullet op_3} = FT(op_2^{op_1}, op_3)$). Unfortunately, using this simple forward transformation strategy, both sites end with inconsistent states. When op_3 is generated, it has already included the effect of op_1 (since $op_1 \rightarrow op_3$) that another character has been inserted as the result of op_1. On the other hand, op_2 does not include any effect of op_1, hence simply transforming op_3 against op_2 will not preserve the correct user-intention.

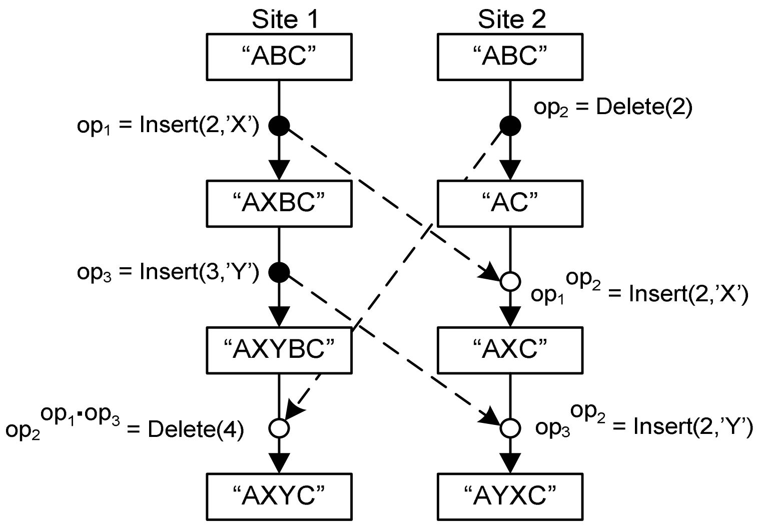


Figure 3-13 The dOPT puzzle

This particular problem is commonly known as the *dOPT puzzle*, and later work [131, 132] has identified that the cause of this problem is that op_3 is not defined at the same state as op_2, or in other words, op3 is not *context-equivalent* to op_2. Sun et al. [137] provides the formal definition of operation context and the context equivalent relation, and explains the precondition and postcondition of a forward transformation as follows.

Definition 3-6. *Generation context of an operation*

An operation *op*, originated at site *S* is said to be generated at context $GC(op)$ if $GC(op)$ is the list of operations that has been executed by site *S* when *op* is generated ($GC(op) = [op_1, op_2, ..., op_k]$, where op_1 is the first operation executed by site *S*, and op_k is the last operation executed by site *S* before generating *op*).

Definition 3-7. *Context equivalent relation*

Let op_i and op_j be two operations with $GC(op_i)$ and $GC(op_j)$ are their respective context, $op_i \equiv_c op_j$ iff $GC(op_i) = GC(op_j)$.

Definition 3-8. *Precondition and postcondition of a forward transformation*

$FT(op_i, op_j) = op_i'$ correctly preserves user intention if the following conditions are satisfied:

Precondition: $op_i \equiv_c op_j$, and

Postcondition: $GC(op_i')$ = $GC(op_j)$+$[op_j]$ and $Intention(op_i, GC(op_i))$ = $Intention(op_i', GC(op_i'))$.

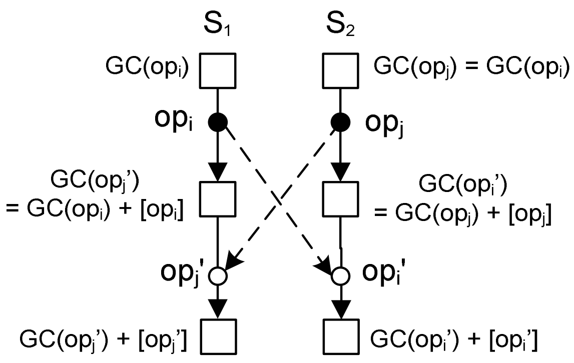


Figure 3-14 Operation context in forward transformation

Since the dOPT puzzle was discovered, it has been realised that operational transformation framework needs more than just the operation transformation rules to maintain document consistency. Each remote operation must be processed, the current document state may need to be modified, and the operations history may need to be reorganised before the remote operation is actually transformed against the concurrent operations in the history. The operational transformation framework, therefore, consists of two major components: the *operation transformations rules* and the *operation integration* algorithm. The former defines the variant result of an operation when transformed against another operation. The latter defines the procedure and the necessary process for integrating each operation (local and remote). It determines how local operations are executed, what a site does when it receives a remote operation, and what a site does before each remote operation is actually transformed. It also determines which operations the remote operation needs to be transformed against. The operation integration algorithm is also known as the control algorithm in some work [83, 84, 85].

Jupiter

The consistency management algorithm of Jupiter [101] is derived from the dOPT optimistic algorithm. The main difference is that since Jupiter utilises a central server, the algorithm is substantially simplified. The server and the clients hold document replicas. Although each client holds a document replica, the communication happens only between the server and the client. Each operation generated by a client is sent to the server. The server transforms the operation accordingly, applies it to its document replica, and broadcasts it to all other participants. Upon receiving the operation from the server, each client may further transform the operation depending on the previously executed operations. Jupiter uses a 2 dimensional *state space* graph to keep track of all possible operation transformation paths, and it ensures that operations that are involved in a transformation must originate from the same point in the graph. This combination of a centralized architecture and optimistic concurrency control provides both easy serialisability of concurrent update streams and fast response to user actions. However, since Jupiter requires a central server, it is not suited to mobile ad-hoc networks.

adOPTed

Ressel et al. [115] proposed the adOPTed algorithm to solve the dOPT puzzle. The adOPTed algorithm constructs an *N*-dimensional interaction graph (where *N* is the number of collaborating sites) that contains all operations in various possible transformation variants, and each site chooses a path to bring the document to the end state. This algorithm allows the concurrent operations to be executed and transformed correctly according to its generation and execution context. This algorithm, however, requires each site to construct a new graph every time a remote operation is received and thus as the number of concurrent operations and participating sites increases, so does the complexity of the graph. This makes it difficult to manage the graph over long collaboration sessions, particularly on resource constrained mobile devices.

They also introduced two transformation properties that have to be satisfied by any transformation functions: TP1 and TP2. TP1 states that if there are two concurrent operations, op_1 and op_2, the transformation function must ensure that the document state is the same regardless of which operation is executed first. Executing op_1 followed by $op_1^{op_2}$ should yield the same document state as executing op_2 followed by $op_2^{op_1}$. TP2 states that if there is another operation op_3 (concurrent to op_1 and op_2) and op_3 is to be executed after a site executes op_1 and op_2, the transformation of op_3 against op_1 and op_2 has to be the same regardless of the execution order of op_1 and op_2. The transformation properties are formally defined as follows.

Transformation Property 1 (TP1):

$$op_i \bullet op_j^{op_i} \equiv op_j \bullet op_i^{op_j} \textbf{ OR } op_i \bullet FT(op_j, op_i) \equiv op_j \bullet FT(op_i, op_j)$$

Transformation Property 2 (TP2):

$$op_k^{op_i \bullet op_j^{op_i}} = op_k^{op_j \bullet op_i^{op_j}} \textbf{ OR } FT(op_k, FT(op_j, op_i)) = FT(op_k, FT(op_i, op_j))$$

LICRA

LICRA [72] is introduced with a proposition that neither locks nor clocks nor global information are required to establish data consistency. For causality, LICRA uses direct dependencies of generated operations. Each generated operation is piggybacked with the id of the previously generated operation, so when the recipient site receives the operation, the site is able to preserve causality by making sure the previous operation has been executed locally. For user intention preservation, it uses operation transformations. Although direct dependencies of generated operations can be used to preserve causality, it can only be used for operations that are generated from the same site. There is no way to determine if an operation generated from one site is concurrent to an operation from another site. Upon receiving an operation, LICRA transforms the received operation against all operations in the history, including the operations that precede it. Naturally, an operation has already included the effect of preceding operations. Reapplying the transformation will 'double' the effect on op, thus violates the user intention.

SOCT2

Suleiman et al. [126] presents SOCT2 (Sérialisation des Opérations Concurrentes par transformation or "serialization of concurrent operations by transformation") algorithm that solves the dOPT puzzle and aims to manage document consistency while respecting user intention without operations undo and redo. Like dOPT, once an operation is generated, it is executed immediately to guarantee a minimum response time and then broadcast as a remote operation. A remote operation is received by another site and its causality is preserved by using the state vector technique. The received remote operation is then executed and added to the operation history. Before execution, the operation needs to be transformed to respect the user intention.

SOCT2 introduces a remote operation integration algorithm. Unlike dOPT that simply transforms the remote operation against all concurrent operations in the history, the integration algorithm of SOCT2 ensures that the remote operation is transformed correctly and executed at the correct state. SOCT2 uses the idea of dividing the history into two groups as introduced by Sun et al [136] whereby before processing the remote operation, SOCT2 separates the history into two sequences (Figure 3-15): the first sequence consists of the operations that precede the to-be-integrated operation and the second sequence consists of the operations concurrent to it. The remote operation is then forward transformed against the operations in the second sequence before it gets executed.

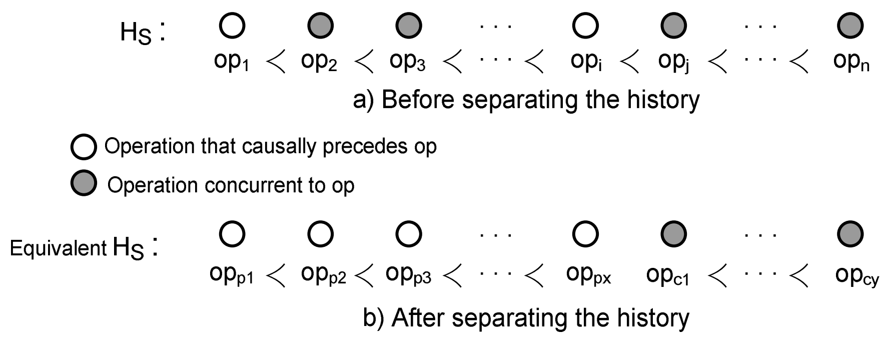

Hs :
op1 < op2 < op3 < ··· < opi < opj < ··· < opn
a) Before separating the history
Operation that causally precedes op
Operation concurrent to op
Equivalent Hs :
opp1 < opp2 < opp3 < ··· < oppx opc1 < ··· < opcy
b) After separating the history

Figure 3-15 History separation in SOCT2

The separation of the history involves *backward transformations* to shift all preceding operations backward to the beginning, and all concurrent operations to the end of the history. Backward transformation, the opposite of the forward transformation as the name implies, is used to transform an operation to exclude the effect of any other previously executed operations. Backward transformation is also known as Exclusive Transformation (*ET*) [132]. Let O_i be a document state and $O_i \bullet op$ is the document state after applying *op* on O_i. Transforming operation op_2 backward against op_1 means transforming op_2 into its variant ${}^{op_1}op_2$ such that the effect of op_2 on $O_i \bullet op_1$ is the same as the effect of ${}^{op_1}op_2$ on O_i. It is defined as follows:

$$BT\,(op_1, op_2) = {}^{op_2}op_1\text{, such that } Intention\,({}^{op_2}op_1, O_i) = Intention\,(op_1, O_i \bullet op_2)$$

Figure 3-16 illustrates a backward transformation of op_j against op_i to exclude the effect of op_i to become op_j'. If site *S* has not executed op_i, op_j' would be the operation executed on $GC(op_i)$ to realise the same user intention as op_j. In Definition 3-9, the precondition and postcondition of a backward transformation are defined as in [137], and similar to forward transformations, backward transformation rules as defined in Figure 3-17.

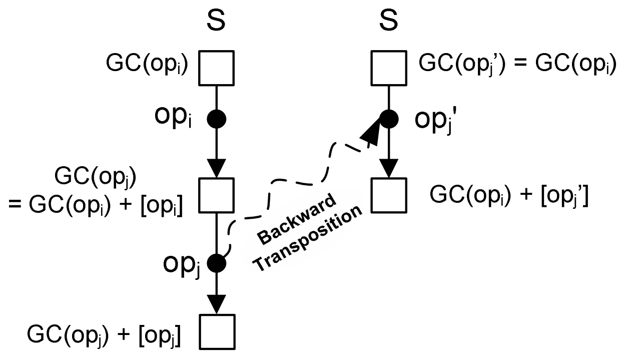


Figure 3-16 Operation context in backward transformation

Definition 3-9. *Precondition and postcondition of a backward transformation*

$BT(op_j, op_i) = op_j'$ correctly preserves user intention if:

Precondition: $GC(op_j) = GC(op_i) + [op_i]$, and

Postcondition: $op_j' \equiv_c op_i$ and $Intention(op_j, GC(op_j)) = Intention(op_j', GC(op_j'))$.

```
$BT\ (op_1 = insert(x_1, a_1), op_2 = insert(x_2, a_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 - 1\ , a_1)$
        else $op_1' = insert(x_1, a_1)$;

$BT\ (op_1 = insert(x_1, a_1), op_2 = delete(x_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 + 1, a_1)$
        else $op_1' = insert(x_1, a_1)$;

$BT\ (op_1 = delete(x_1), op_2 = insert(x_2, a_2),) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = delete(x_1 - 1)$;
        if $x_1 < x_2$ then $op_1' = delete(x_1)$;
        if $x_1 = x_2$ then $op_1' = ø$;

$BT\ (op_1 = delete(x_1), op_2 = delete(x_2)) = op_1'$, where
        if $x_1 \geq x_2$ then $op_1' = delete(x_1 + 1)$
        else $op_1' = delete(x_1)$;
```

Figure 3-17 Backward transformation rules

Backward transformation, together with forward transformation, can be used to swap the execution order of a pair of operations without violating the intention of both operations, to ensure it still results in the same state as the previous order (Figure 3-18). The swap function [112] is useful to re-organise the operations in history according to a certain ordering scheme while preserving the intentions of all operations in the history and consequently resulting in the same document state. The swap function is defined as follows.

$swap(op_1, op_2) = (op_2', op_1')$, where

$op_1' = FT\ (op_1, op_2')$ and $op_2' = BT\ (op_2, op_1)$, such that $op_1 \bullet op_2 = op_2' \bullet op_1'$.

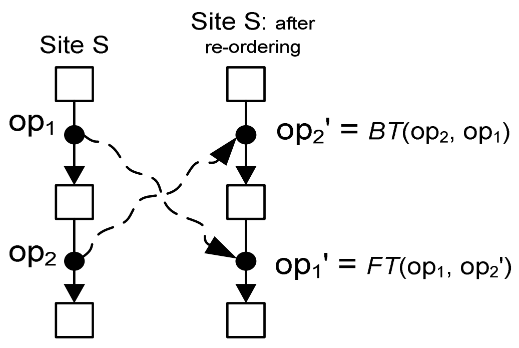


Figure 3-18 Swapping two operations using operation transformations

Using backward transformation to swap operations in the operation history, SOCT2 ensures that the remote operation is transformed against concurrent operations that have been generated at the same context. Figure 3-19 depicts how SOCT2 solves the dOPT puzzle presented earlier in Figure 3-13.

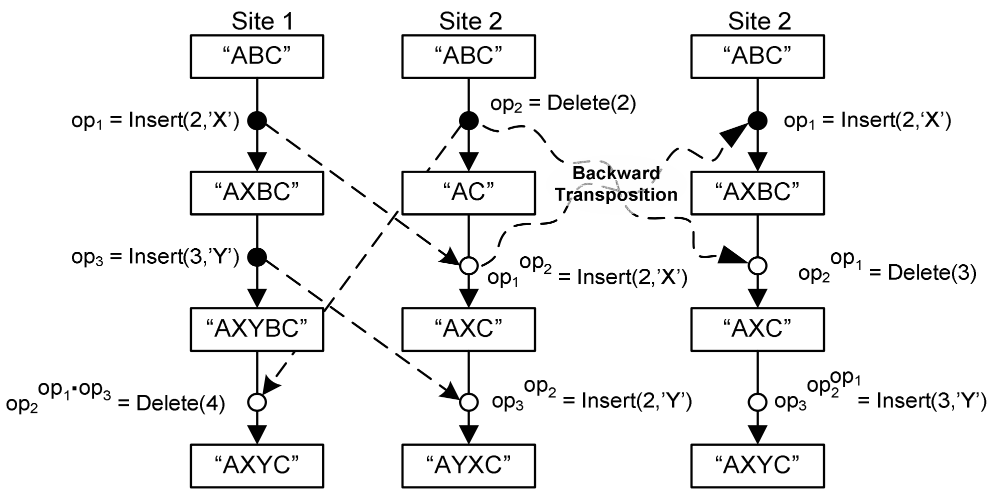


Figure 3-19 SOCT2 uses history reordering to solve the dOPT puzzle

Although SOCT2 solves the dOPT puzzle and aims to ensure document consistency and respect user intention, there are some cases where transformation condition TP2 is violated and consequently two sites end up in different states [142]. Guerraoui et al. [59] has shown that in certain scenarios, it can lead to inconsistency of the replicas as shown in Figure 3-20. This problem is commonly known as the *TP2 puzzle*, where $op_k^{op_i \bullet op_j^{op_i}} \neq op_k^{op_j \bullet op_i^{op_j}}$.

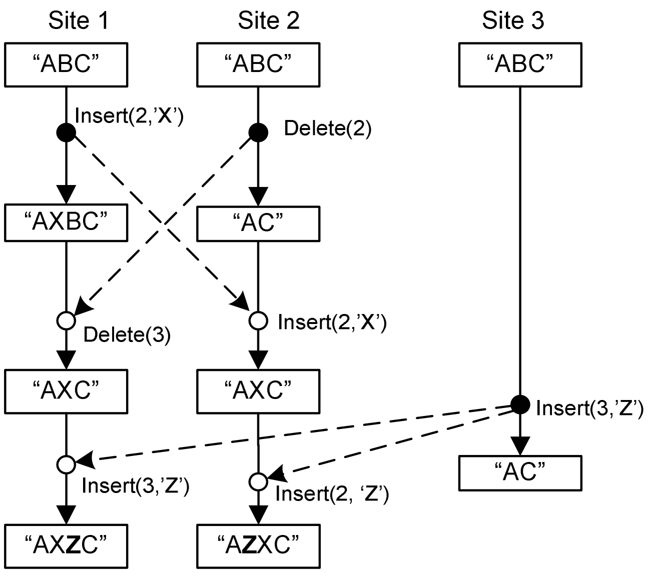


Figure 3-20 The TP2 puzzle

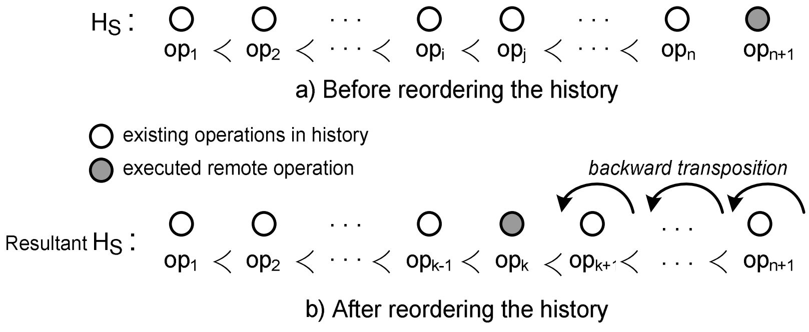
As illustrated in Figure 3-20, the TP2 puzzle occurs when there are three concurrent operations generated by three different sites and one operation is transformed against the other two operations in different orders. Suleiman et al. [126] attempt to solve the TP2 puzzle by introducing modified operation transformation rules. Two parameters are added to each insertion operations, each of which contains the set of operations that have deleted a character before (and respectively after) the inserted character. The proposed transformation rules, however, can only solve the TP2 puzzle when the two operations insert the same character at the same position.

SOCT3

Vidot et al. [142] introduced SOCT3 with the intention of ensuring document consistency under all scenarios. SOCT3 eliminates the need to satisfy the TP2 condition, hence avoiding the TP2 puzzle, by ordering operations in the history according to a specific total ordering scheme such that if $op_i \Rightarrow op_j$, op_i appears before op_j in the histories of all sites.

The total order of the operations is achieved by timestamping each operation with a unique sequence number, assigned by a g*lobal sequencer*, to each operation. As soon as S_i generates an operation op_i, it sends a request to the global sequencer to ask for a sequence number. The sequence number is then returned to the S_i, and op_i is timestamped with the received sequence number before it is broadcast to other sites. Like SOCT2, the recipient site S_j will integrate op_i into its history by firstly reordering the history into two sequences and then forward transforming op_i against the second sequence. However, unlike SOCT2 that places the executed operation at the end of the history, SOCT3 places the integrated operation at its designated position in the history based on its sequence number.

As illustrated in Figure 3-21, a remote operation with sequence number k is executed and placed at the end of the history as op_{n+1}. Then the operation is shifted backward using backward transformation to its appropriate position (position *k*) based on its sequence number. This extra step is to ensure the operations in the history are totally ordered. SOCT3 preserves causality by implementing the state vector technique; it preserves user intention using the operation transformation technique; and it enforces convergence by totally ordering the operations in the history. Figure 3-22 shows how SOCT3 solves the TP2 puzzle by reordering the operations in the history based on a predefined total ordering scheme.



Hs :
op1 < op2 < ··· < opi < opj < ··· < opn opn+1
a) Before reordering the history
existing operations in history
executed remote operation
backward transposition
Resultant Hs :
op1 < op2 < ··· < opk-1 < opk < opk+1 < ··· < opn+1
b) After reordering the history

Figure 3-21 History reordering in SOCT3

Although SOCT3 is proven to be correct [59], it has a few drawbacks. It relies on the global sequencer for its total ordering scheme, hence is not suitable for mobile ad-hoc networks. Additionally, since it uses a centralised sequencing scheme, the collaboration cannot be partial: either all sites collaborate (or are connected) or each one works separately. Furthermore, like SOCT2, SOCT3 requires the history to be copied before the history is reorganised and separated into two sequences, leaving the original history in a proper total order. This process requires additional processing power and memory which may not be suitable for constrained mobile devices.

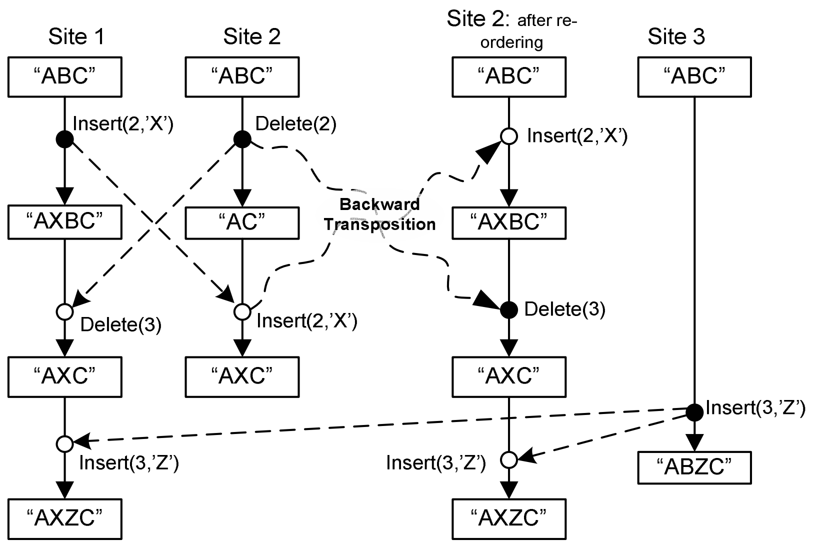


Figure 3-22 SOCT3 solves the TP2 puzzle

SOCT4

The SOCT4 [142] algorithm works similarly to SOCT3 algorithm. However, instead of immediately broadcasting the generated operation, the originator site defers the broadcast until all operations that precede it have been received and executed by that site. The operation is then forward transformed against all concurrent operations in the local history just before it is broadcast so that the recipient site needs only to transform the operation against the local operations that are waiting to be broadcast (i.e. operations with greater sequence numbers). The operations, once delivered, are no longer needed since they are not involved in any future transformations.

SOCT4 has the following advantages:

1. It removes the need for state vectors. Causality is preserved by deferring the operation broadcast.
2. Since the integrated operation has the highest sequence number, it does not have to perform any backward transformation to shift the integrated operation to its correct position in the history.
3. The history can be kept small since the delivered operation is no longer needed and can be discarded.

However, like SOCT3, it relies on the global sequencer which means that it has a single point of failure, and the collaboration cannot be partial. Furthermore, since the generated operation will not be broadcast immediately, it will stay at the originator site for quite some time. In mobile network environments, where the delay is quite significant and disconnection is frequent, the operation may be held indefinitely at the originator site. It may also hold the other sites since the other sites have to defer the operation broadcast until all operations with smaller sequence numbers have been received.

GOT

Sun et al. [137] introduced an alternative algorithm named GOT (Generic Operation Transformation) to ensure replica consistency by preserving causality and user intention. GOT is similar to SOCT2 in the following ways.

- It ensures responsiveness by immediately executing local operations.

- It is fully replicated and does not need any central server to collaborate.
- Users at any site can freely edit their own replica at any time.
- It preserves causality.
- It preserves user intentions by implementing operation transformations.

Like SOCT3, GOT imposes a total ordering scheme such that all operations eventually appear at the same order at all sites' history. However, the handling of the remote operation in GOT is different from SOCT3 as mentioned. When a site *S* receives a remote operation *op*, all operations in the history that 'happen after' *op* ($\forall i$: $op \Rightarrow op_i$) are undone. Operation *op* is then transformed against the remaining concurrent operations before it is executed. Finally, the undone operations are redone by transforming them according to the new context (i.e. to include the newly executed operation). While the undoing/redoing mechanism easily ensures that operations get executed at the same order at all sites, this algorithm is computationally expensive since it requires a large number of undo and redo operations (and their resulting transformations) and is thus not immediately suitable for use on mobile devices with limited processing power and battery life. Furthermore, undo operations have to be defined for each operation primitive.

GOTO

GOTO (GOT Optimized) [131] algorithm was introduced to optimize GOT. GOTO optimizes GOT by eliminating the need to undo and redo the operations. GOTO is similar to SOCT2 where the history is separated into two sequences: the first one contains the preceding operations and the other contains the concurrent operations, and the remote operation is transformed against the second sequence. Similar to SOCT2, GOTO does not impose any total ordering scheme thus there are some cases where condition TP2 is violated.

State Difference based Transformation (SDT)

Although SOCT3/4 and GOT solve the TP2 puzzle and ensure content consistency by totally ordering the execution of the operations, Li et al. [84, 83] in their recent work discovered that they do not necessarily ensure intention consistency (in other words, they might violate the user intentions even though the replicas are consistent), especially in the case where the transformation involves two operations that insert different characters at the same position.

Consider the scenario outlined in Figure 3-23. The user at site 1 inserts character 'X' after 'B' (op_1), the user at site 2 inserts character 'Y' before 'B' (op_2), and the user at site 3 deletes character 'B' (op_3) concurrently. If op_2 and op_3 in site 2 and site 3 are ordered in such a way that $op_2 \Rightarrow op_3$, op_1 will be transformed accordingly such that the document replicas will end up consistently with character 'Y' appearing before character 'X'. When op_3 arrives at site 2, it will be transformed to become $op_3' = delete(2)$ since character 'B' has been shifted forward as a result of op_2. When op_2 arrives at site 3, it will be transformed, executed and reordered such that $op_2 \Rightarrow op_3$. Therefore, when op_1 = *insert*

(2,'X') arrives at either site 2 or site 3, it is correctly transformed against op_2 = *insert*(1,'Y') and op_3' = *delete*(2).

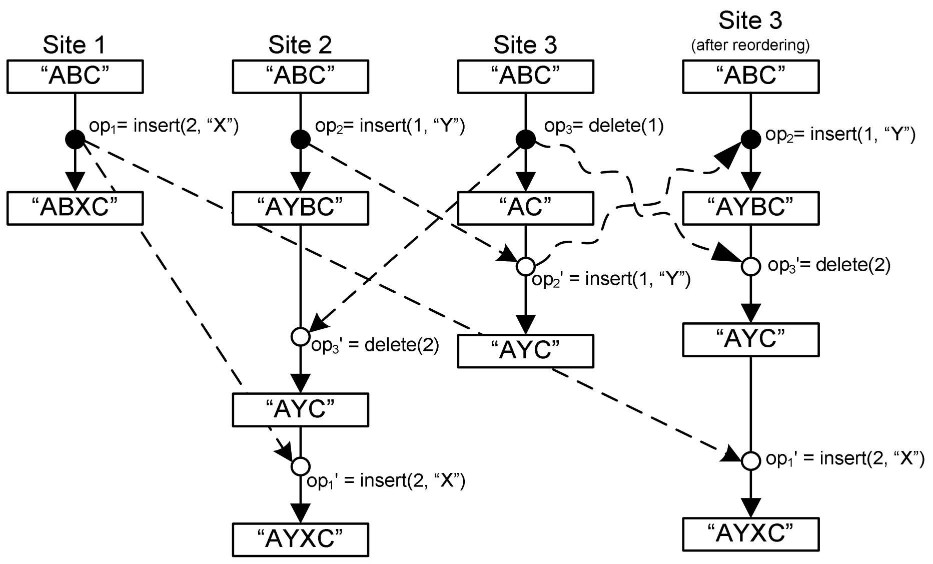


Figure 3-23 Preserving operation effects, adopted from [137]

However, due to implementation of different total ordering scheme, the different site id, or due to an arbitrary reason, op_2 and op_3 might be totally ordered differently. Figure 3-24 shows how the user intentions are violated when $op_3 \Rightarrow op_2$. Unlike the scenario in Figure 3-23, op_1' = *insert*(1,'X') has to be transformed against op_2' = *insert*(1,'Y'), which causes the ERV since eventually character 'X' is placed before character 'Y'. This anomaly is termed as the operation Effects Relation Violation (ERV) puzzle.

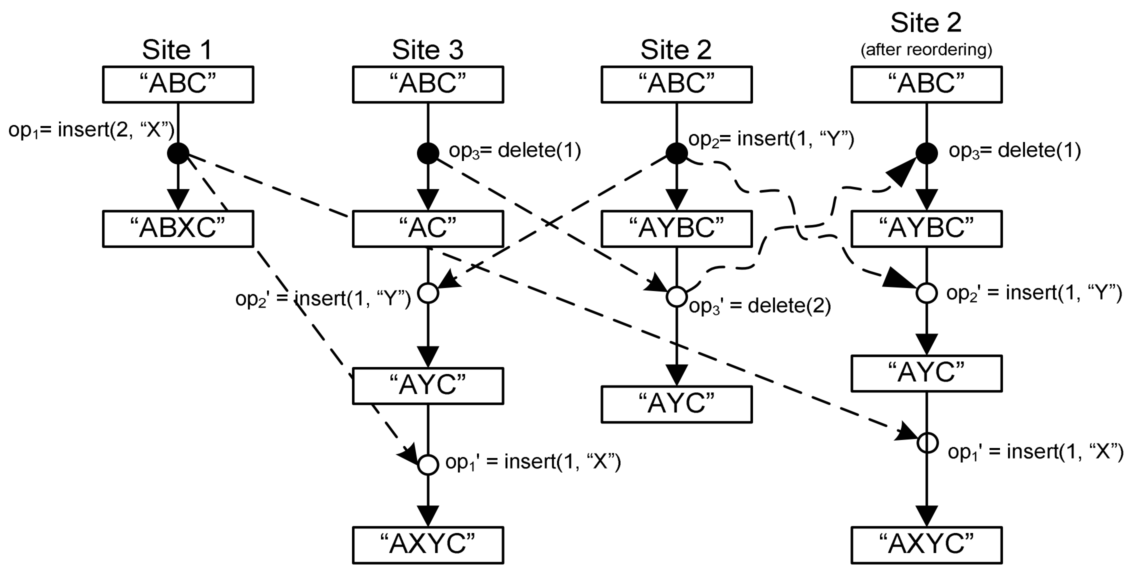


Figure 3-24 The operation effects relation violation (ERV) puzzle

Consequently state difference based transformation (SDT) was proposed in order to solve the ERV puzzle. The original intention of each operation is recovered by computing its β value against a well known document state (the latest synchronization point). The β values of two operations are then compared during a transformation. The fixes to the forward and backward transformation functions rely on the definition and computation of β and δ values. The detail of the how to compute those values are given by Li and Li [83]. Although the algorithm is able to recover the original intention of the transformed operation, the computation of β and δ rely on a common previous state or a well-known document state (the latest synchronization point). However, in real time collaboration where operations are generated by sites independently, two sites might not have one common state. Due to concurrent operations from one and other sites, sites may only be in the same state when there is no operation generated for some time and sites reach quiescence. Furthermore, even when there is one common state, it is difficult for a site to determine the common state of two operations.

Alternatively, the modification to the forward and backward transformation, which is able to solve the ERV puzzle, is presented in section 3.5.2 of this thesis. Compared to SDT, it is considerably simpler, hence requiring less processing and memory overhead. The following figures show how the proposed solution can solve the ERV puzzle. (Figure 3-25 shows the operation effects preservation in the same scenario as the one in Figure 3-23, while Figure 3-26 solves the problem with the scenario presented in Figure 3-24).

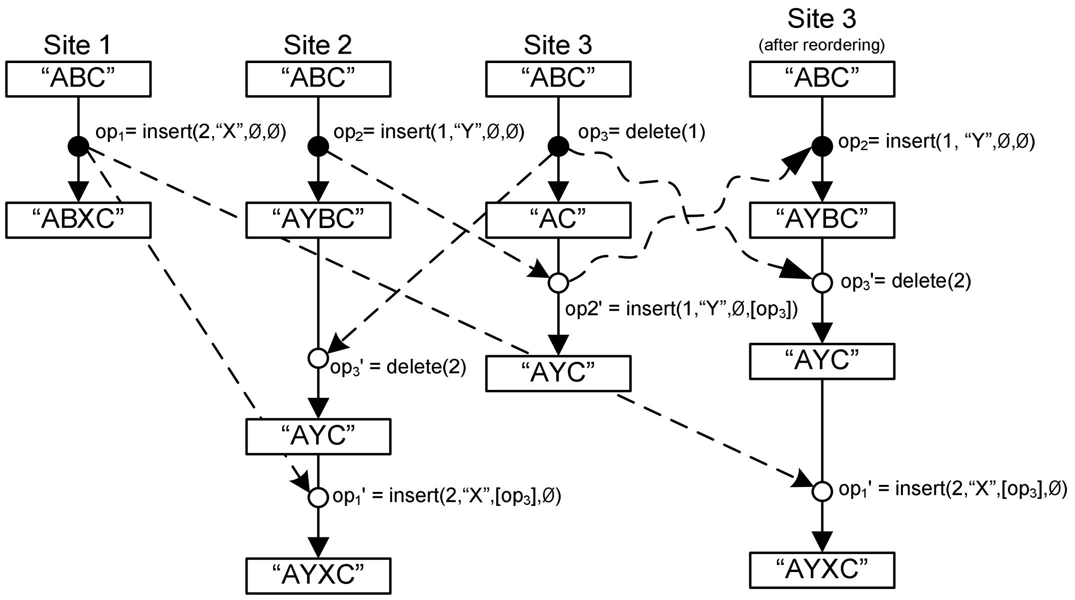


Figure 3-25 Preserving operation effects

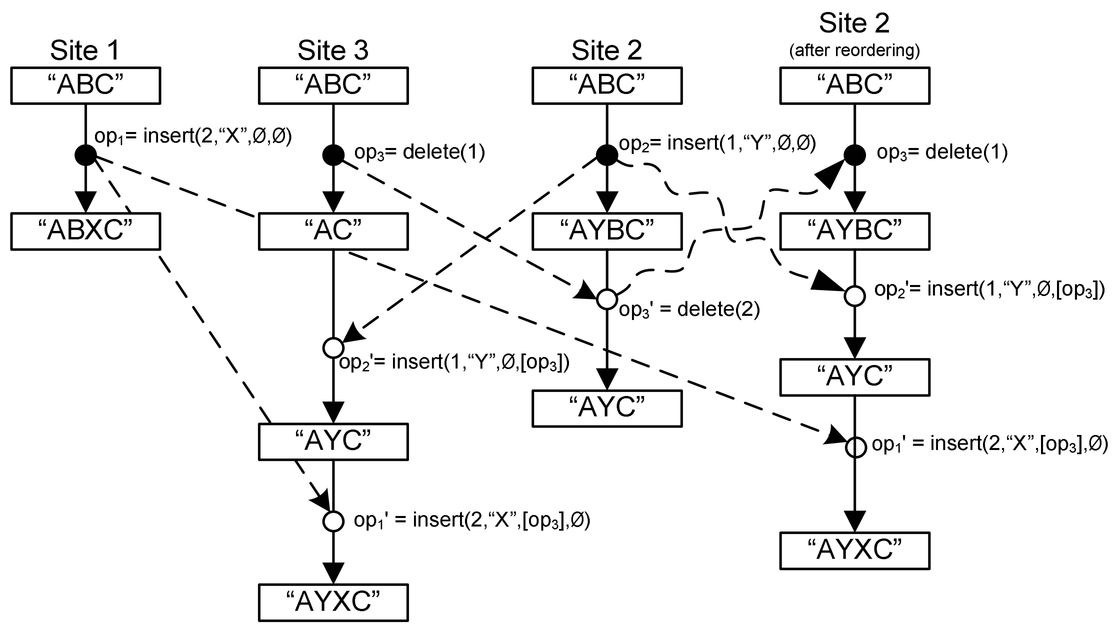


Figure 3-26 Solving the ERV puzzle

Time Interval Based OT (TIBOT)

Li et al. [86], aiming to solve the TP2 puzzle and to reduce the algorithm complexity, proposed a Time Interval Based Operational Transformation (TIBOT) algorithm. In TIBOT, every site maintains a linear logical clock and all clocks are initialized to a common value, e.g., 0. Clocks at different sites may tick at different speeds but must take the same sequence of values, e.g., 0, 1, 2, 3,... . A time interval is the period between two consecutive clock ticks. The length of intervals between two consecutive clock ticks is chosen based on the principle that there are editing operations in as many time intervals as possible. Hence in general logical clocks should tick no faster than the keyboard response time.

The operation propagation rule in TIBOT ensures that a site can only broadcast a locally generated operation after it receives all operations generated by other sites at the previous time intervals. Furthermore, TIBOT ensures that any operation must be transformed against all operations carrying earlier timestamps before it is propagated. So when an operation arrives at a remote site, it only needs to be transformed against all executed operations carrying the same or later timestamps at that site.

TIBOT has several advantages. Firstly, it does not use a state vector. Secondly, the operation context is easily determined by examining the timestamp of the operation. Thirdly, backward transformation is not necessary in TIBOT. Fourthly, TIBOT also solves the dOPT puzzle and the TP2 puzzle by imposing certain synchronization and propagation rules. Finally, the time complexity of TIBOT is $O(n)$ (as compared to $O(n^2)$ in GOTO).

However, TIBOT has several drawbacks that make it unsuitable for mobile networks. The problem of this algorithm lies on the fact that each site has to broadcast at least one message at any given time interval, be it an operation or a zero-operation message. Sites can be blocked for an indefinite time if one of these events occurs: (1) one or more sites crash or get disconnected, (2) one or more operations are missing during transmission, or (3) one or more zero-operation messages are missing during transmission. The above events however are common in mobile network environments. There is another problem for this algorithm due to the fact that a site can only broadcast the local operation after it has received all operations generated from other sites at the previous time intervals. Sites can generate more than one operation at one time interval. Therefore, it is difficult for a site to know whether it has received all operations generated from other sites at the previous time intervals. This algorithm might be fixed by forcing every site to send an additional message at the end of the time interval to indicate that no more operations will be generated in that time interval. However, even with this scheme, the algorithm still fails in the event of site crashes, or site disconnection, and missing operations.

TreeOPT

Most of the existing work on collaborative text editing employs linear representation of the document. Using linear document representation, various OT approaches have been developed to ensure document consistency amidst concurrent operation. Ignat et al. [67] pointed out that a linear representation is not adequate to represent the common pervasive rich editing applications. Furthermore, linear document representation requires each site to maintain a single and potentially large operation history. Every time a remote operation arrives, it has to trace through the history to determine operation concurrency and the need to transform it against concurrent operations. Therefore, a tree-like hierarchical document representation is employed and an algorithm called *treeOPT* is introduced to recursively transform operations at various document levels whenever a remote operation arrives. An operation can be defined in terms of different granularity levels. If a user inserts a word, then the operation is defined at the word level. If a user inserts a character, then it is defined at the character level. If an operation is defined at the character level, the operation must include the position of the character in all document levels, i.e. the position of the character in a word, the position of the word in a sentence, the position of the sentence in a paragraph and the position of the paragraph in the whole document. The greater the number of granularity levels in the document hierarchy, the more relative positions must be included in the operation definition. Therefore, when a character insertion operation arrives at a site, it has to be transformed at the paragraph level, at the sentence level, at the word level and eventually at the character level.

In terms of performance, treeOPT has better complexity – $O(spannedHistory)^2$ – as compared to the existing OT algorithms for single-level documents, such as GOT, SOCT2, SOCT3 ($O(N^2)$). The spanned history is the history distributed on a single path of the tree, which is significantly smaller than the single history buffer as maintained by previous OT algorithms.

However, maintaining the document using a tree-like hierarchical representation has some limitations, especially related to splitting nodes and merging two nodes. In a linear representation, inserting a '.' in a sentence is simply inserting a character, and any other concurrent operations can easily be transformed accordingly to preserve user intentions. On the other hand, in a hierarchical structure, inserting a '.' means splitting a node into two nodes, and some concurrent operations are not easily transformed to preserve user intentions and document structure. Splitting a node happens in events such as when a user inserts a space (‘ ‘) in a word, a user inserts a dot ('.') in a sentence or a user splits a paragraph into two. Joining a node happens in events such as when a user deletes a space character between two words, a user deletes a '.' character, or a user deletes a newline character between two paragraphs.

Although a few possible solutions were presented by Ignat et al. [67], they could not completely address and solve the problem related to splitting and merging nodes. Ignat et al. [67] eventually employ an alternative solution, "no splitting", which means that if a user wants to split a sentence into two sentences by inserting a '.', the two new sentences are still represented as one large sentence with a '.' in the same node (no new node is created). This approach, although it works, does not maintain the document structure correctly. Furthermore, if splitting and joining (deleting a space character between two words) happen often, which is the case for text editing, the document hierarchical structure no longer represents the correct document structure.

3.4. Some Recent Related Work

Three algorithms were recently proposed to support consistency maintenance in real time group editors: Tombstone Transformation Function (TTF) [104], WithOut Operation Transformation (WOOT) [105] and Context-based Operation Transformation (COT) [138]. These algorithms propose novel concepts and principles in ensuring document consistency that have not been used by earlier work. However, due to their very recent publication dates, this thesis only provides informations and claims made by these algorithms. Thorough and empirical evaluation of these algorithms are part of future work, therefore the proposed algorithm presented in this thesis only considers concepts or principles introduced by earlier algorithms. This section discusses the three algorithms and provides comments about their contributions.

3.4.1. Tombstone Transformation Functions (TTF)

It has been shown that OT fails to maintain document convergence in scenarios where insert and delete operations are involved at the same time. Both the TP2 and ERV puzzles are caused by concurrent insert and delete operations. The TP2 puzzle is caused by different order execution of concurrent insertion and deletion operations. The ERV puzzle is caused by a wrong backward transformation of an insert operation against a delete operation.

Existing algorithms have tried to solve the TP2 puzzle by various methods, for example SOCT3 and SOCT4 uses a central sequencer, and GOT uses an undo/do/redo scheme. However, each of them has a few drawbacks as mentioned previously. They either assume high resource availability or present a single point of failure.

The main idea of Tombstone Transformation Functions (TTF) [104] is to keep deleted characters as tombstones, similar to the ideas proposed by He et al. [62] and Wu et al. [146]. The characters that have been deleted are not removed from the document. They are however not shown to users making them appear to be deleted. A delete operation is practically a ‘hide’ operation since it only hides a character instead of actually deleting it.

Hence, the tombstones are kept so that the conflict between delete/insert operations is not ambiguous.

Therefore, as opposed to traditional transformation functions, the TTF are monotonic transformations of the effect position of operations since they only compute additions. The position of one character will grow monotonically to the same value independently of the equivalent transformation path taken. This monotonic property has another interesting consequence: TTF preserves order relationships between characters which is considered in [84] as an instantiation of the intention preservation criterion defined by Sun et al [132].

The paper also discusses inverse-TTF to be used for backward transformation. An optimised TTF is also proposed where each visible character keeps an integer value equals to 1 + the number of invisible characters located between it and the visible character preceding it. The idea is that the deleted characters need not be kept, but each character needs to take into account the number of deleted characters directly preceding it.

The original TTF retains deleted characters as tombstones. Therefore the space requirement will grow indefinitely. This is not suitable for mobile devices. The optimised TTF, on the other hand, reduces/minimises the space requirement by actually deleting the deleted characters and letting the visible characters take into account the number of deleted characters.

However, TTF is only a set of transformation functions; hence it cannot be used alone in ensuring document consistency. Oster et al. [104] stated that TTF can be used with any existing operation integration algorithm to maintain document consistency in synchronous groupware systems. However, based on the discussion above, the existing integration algorithms have shortcomings that make them unsuitable for mobile replicated architecture. Section 3.5.1 therefore presents the proposed operation integration algorithm that is suited to mobile replicated architecture.

3.4.2. WithOut Operational Transformation (WOOT)

WOOT [105] is a new framework that ensures intention consistency but without operational transformations, without vector clocks and without central sites. As the operation transformation approach becomes more complex and difficult to prove, WOOT provides a new direction for collaborative editing without operation transformations.

WOOT is claimed to be particularly adapted to very large peer-to-peer networks, drastically simpler than SDT [84] and easy to implement. Instead of re-computing the orderings at reception using vector clocks, in WOOT the ordering is sent with the operation as this information is known when operations was generated. For example, if a user inserts an ‘X’ between ‘A’ and ‘B’, instead of executing and broadcasting insert(2,’X’) as in OT algorithms, WOOT executes insert(2,"X") and broadcast insert(‘A’<’X’<’B’).

One apparent limitation of WOOT is that it keeps all operations and all characters even though they have been deleted, hence adding up to the space requirement. However, as it is a very recent work, the precise analysis of WOOT algorithm is left to future work.

3.4.3. Context-based Operation Transformation (COT)

COT [138] is motivated by its claim that the theory of causality, which has been the foundation of all prior OT systems, is inadequate to capture essential correctness requirements. It outlines some of the major drawbacks of the theory of causality (i.e. the use of the state vector technique):

1. The theory of causality is not adequate to solve the dOPT puzzle.
2. The concurrency relation does not capture the essential condition for a correct transformation: the two input operations must be defined on the same document state.
3. State vector is only capable of representing original normal editing operations, and not inverse operations.

COT defines the theory of operation context to replace the theory of causality. The operation context theory is similar to the one defined by Sun et al. [132], with the addition of the definition of an inverse operation context. To replace the state vector, COT uses a context vector. An operation context vector (*CV*(*O*)) consists of all operations that have been executed prior to generating operation *O*. COT also introduces a unique way of distinguishing the inverse operation in the context vector.

The algorithm COT-DO has the same idea as previous algorithms that are based on total ordering of operations in the history buffer (GOT, SOCT3). It properly transforms incoming operations by making sure that the incoming operation is transformed against concurrent operations in the correct context. The context vector simplifies the algorithm as it keeps the record of the preceding operations as the context, and therefore, with simple operation set calculation, it is easy to find which operations it needs to be transformed against.

Advantages of COT are as follows: (1) unlike GOT, COT-DO does not involve undoing and redoing the already executed operation, (2) unlike SOCT3, COT-DO does not require a central sequencer for its total ordering mechanism, (3) unlike SOCT2 and SOCT3, COT-DO does not require history reordering, and (4) unlike GOTO-ANYUNDO, the basic COT algorithm does not use ET (Exclusion Transformation) functions [128], thus avoiding the requirement of the Reversibility Property (RP) between IT and ET functions [128].

One apparent limitation of COT is that it introduces more overhead as compared to the use of state vectors. State vectors simply denote the 'number' of executed operations from each site rather than recording all operations that have been executed. In COT, since every operation bears a context vector, the space consumed for the context vector may be higher than its state vector counterpart. However, more work needs to be done to precisely analyse the COT algorithm, and as it is a very recent work, this is left to future work.

3.5. Proposed Algorithm

Based on the discussion above, this thesis aims to develop a consistency management algorithm that is suitable for a mobile replicated architecture. The algorithm should ensure the intention and the content consistency of the document replicas (by addressing the dOPT, TP2 and ERV puzzles previous described). Furthermore, the algorithm must reduce the resource consumption requirement such that it can be implemented on mobile devices with their limited capacity.

SOCT4 has the following advantages:

1. It removes the need for state vectors. Causality is preserved by deferring the operation broadcast.
2. Since the integrated operation has the highest sequence number, it does not have to perform any backward transformation to shift the integrated operation to its correct position in the history.
3. The history can be kept small since the delivered operation is no longer needed and can be discarded.

However, like SOCT3, it relies on the global sequencer which means that it has a single point of failure, and the collaboration cannot be partial. Furthermore, since the generated operation will not be broadcast immediately, it will stay at the originator site for quite some time. In mobile network environments, where the delay is quite significant and disconnection is frequent, the operation may be held indefinitely at the originator site. It may also hold the other sites since the other sites have to defer the operation broadcast until all operations with smaller sequence numbers have been received.

Section 3.5.1 presents the proposed operation integration algorithm, which describes how each operation (local or remote) is processed and executed. Based on the discussion in the previous section, the proposed operation integration algorithm is based on SOCT3, hence it is capable of solving the dOPT and TP2 puzzles, and it includes novel techniques to make it suited to mobile networks. Although SOCT4 is an improved version of SOCT3 and has several advantages over SOCT3 as described in the previous section, SOCT4 relies on the global sequencer which means that it has a single point of failure, and the collaboration cannot be partial. Furthremore, its deferred broadcast mechanism might cause operations to be held indefinitely waiting to be broadcast at the originator site. Therefore, the proposed algorithm is based on SOCT3 since it is more suited to mobile replicated architecture. Section 3.5.2 describes the limitations of the existing operation transformation rules and proposed operation transformation rules such that it is able to handle identical operations and it is able to solve the ERV puzzle.

3.5.1. Proposed Operation Integration Algorithm

SOCT3 with its use of history separation, total ordering and operation shifting, serves as a suitable basis for application in a mobile context. SOCT3 is capable of solving the dOPT and TP2 puzzles, and each device holds its own local replica making it suitable for a replicated architecture. Furthermore, SOCT3 does not use many resources as other existing algorithms with the same capabilities.

Nevertheless, SOCT3 has a number of drawbacks. Firstly, it relies on a central sequencer for its total ordering mechanism. Secondly, it does not control history size and thus the longer a collaboration session runs, the more memory and/or storage space is consumed. Finally, the history separation step requires that the whole history be copied, with the separation performed on the copy so that the original history is left intact. This results in increased memory usage which, as discussed previously, is problematic on constrained mobile devices.

The first problem - the problem of centralised total ordering scheme - can be resolved by using an existing non-centralised total ordering strategy such as using the state vector technique as implemented by GOT, or the Lamport logical clock (*LC*) approach as used by ORESTE (the use of state vector and Lamport's logical clock to determine total ordering relation of operations are discussed in section 3.2.2). Either one of them can be used to determine the order of the operations in the history to ensure all operations are stored in the same order at all sites. Section 3.6 will evaluate the performance of the algorithm and compare the use of a state vector and a logical clock to determine their efficiency.

The second problem – the problem of history size - can be addressed by implementing a history trimming algorithm that prevents the history from growing indefinitely. Such a technique has been introduced in a non-mobile context by Sun et al. [132]. However, since this algorithm is fully distributed and can be applied independently at each site, it appears to be suitable for use in a mobile context.

The history trimming technique basically aims to remove all unnecessary operations from the operations history. In the context of operational transformation, operations are kept in the history because each remote operation needs to be transformed against all concurrent operations to include their effects before it is executed on the current state of the document replica. An operation op_i in the history, therefore, is no longer required when there are no longer future operations that are concurrent to or precede it. In other words, a site, say S_j,, has to make sure that all other sites have already executed op_i before op_i can be deleted from the history of S_j. For this purpose, each site maintains a *state vector table (VT)* which contains information about the state vectors of all other sites. Let $VT_i[j]$ ($1 \leq j \leq N$) be the state vector of site S_j as known by site S_i, and $VT_i[j][k]$ be the number of operations generated from site S_k that have been executed by site S_j as known by site S_i. Whenever a remote operation op from site S_j is executed at site S_i (note that $V_{op} = V_{S_j}$ at the time op was generated), $VT_i[j]$ is updated to be equal to V_{op} to ensure $VT_i[j]$ is up to date as much as possible with V_{S_j}. Let op_a be an operation generated from site S_k. Sites that have already executed op_a will have $V_S[k] \geq V_{op_a}[k]$, thus all operations op_i that op_a precedes will have $V_{op_i}[k] \geq V_{op_a}[k]$. If site S_i receives an operation op_x from site S_m, site S_i will know that site S_m has already executed op if $V_{op_x}[k] \geq V_{op}[k]$.

Each site also maintains a Minimum State Vector (*MSV*). MSV_i reflects the knowledge of site S_i about the number of operations which have been executed at every site ($MSV_i[j]$ = the number of operations generated by site S_j that have been executed by every site as known by S_i). Initially $MSV_i[j] = 0$, $\forall j \in \{0,...,N\text{-}1\}$. After executing an operation and updating other elements of the VT_i, site S_i updates $MSV_i[j]$ as follows:

$MSV_i[j] = \min(VT_i[k][j]), \forall k \in \{0,...,N-1\}$. If the value $MSV_i[j] = m$, then the first m operations generated at site S_j must have been executed at all sites.

Therefore, if an operation *op* is generated from site S_k, *op* can be deleted from the history of S_i if $V_{op}[k] \le MSV_i[k]$ or, in other words, $V_{op}[k] \le VT_i[j][k], \forall j \in \{0,..., N-1\}$. Figure 3-27 outlines the history trimming procedure.

```
void trim_history() {
        $n$ = size of $H_S$;

        for $i$ = 1 to $n$ do
                $<op_i, LC_{op_i}, V_{op_i}, S_{op_i}> = H_S[i]$;
                deleted = false;

                if $V_{op}[k] \le MSV_i[k]$ then
                        $H_S = H_S - H_S[i]$;
                        deleted = true;
                endif;

                if deleted = false then exit;
        endfor;
}
```

Figure 3-27 History trimming procedure

The history trimming technique obviously requires additional memory to maintain *VT* and more processing cycles to perform the trimming operation. However, over time, this technique is expected to both reduce memory usage and improve performance due to the smaller size of the history being processed. The detail of its performance evaluation is presented in section 3.6.

While the first two problems can be resolved by adopting existing techniques, the third problem – the problem of history copying – has not been addressed, and thus a novel technique called *Partial History Copy* is introduced to minimize the size of the history copy.

Let H_{S_i} be the operation history of site S_i and $H_{S_i}[j]$ be the j-th operation in H_{S_i}. Operations in the history are totally ordered such that $H_{S_i}[j] < H_{S_i}[j+1]$. When a remote operation *op* arrives at S_i, using the history separation step of SOCT3, the history is separated into two sequences: seq_1 and seq_2 (H_{S_i} = seq_1 + seq_2). All operations in seq_1 precede *op* and all operations in seq_2 are concurrent to *op*. This is done on the copy of the history so that the original order of the history is preserved, hence a copy of the history is created every time a remote operation arrives and is discarded after the execution of the remote operation. Since copying the entire history consumes both memory space and processing power, a partial history copy technique is proposed so that only the necessary portion of the history is copied. The proposed technique aims to find an operation op_m in the history where all other operations located to the right of op_m ($\forall i$: $op_m \Rightarrow op_i$) are concurrent to *op*. If op_m is identified, then only operations that totally precede op_m ($\forall i$: $op_i \Rightarrow op_m$) need to be copied and rearranged since they consist of operations that precede *op* and operations concurrent with *op*. The following Lemmas are introduced to help find the appropriate op_m.

Lemma 3-1. *If* $op_i \rightarrow op_j$ *, then* $op_i \Rightarrow op_j$.

Proof. According to

Definition 3-1, there are two possible cases where $op_i \rightarrow op_j$:

1. Operations op_i and op_j are generated by the same site and op_i is generated before op_j.

   If Lamport's clock is used for total ordering, then $C(op_i) < C(op_j)$ since op_j is generated after op_i. On the other hand, if state vector is used for total ordering, then $sum(V_{op_i}) < sum(V_{op_j})$ since the elements of V_{op_j} is greater than or equal to the elements of V_{op_i}. Therefore $op_i \Rightarrow op_j$ no matter what technique is used for total ordering.

2. Operations op_i and op_j are generated by different sites and op_j is generated by site S_{op_j} after op_i is received by S_{op_j}.

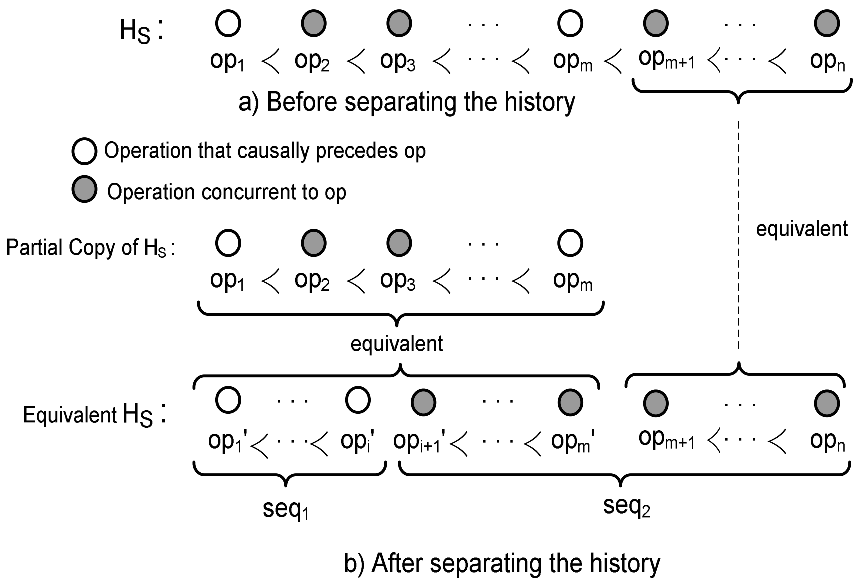
   If Lamport's clock is used, S_{op_j} will update its logical clock C_{S_j} upon receiving op_i such that $C_{S_j} > C(op_i)$, thus $C(op_j) > C(op_i)$, which means $op_i \Rightarrow op_j$. On the other hand, if state vector is used, site S_j will only execute op_i if $V_{S_j}[k] \geq V_{op_i}[k]$ ($\forall k$: $1 \leq k \leq N$). Thus, after executing op_i, $sum(V_{S_j}) > sum(V_{op_i})$. op_j will bear state vector V_{op_j} , equal to V_{S_j} , which makes $sum(V_{op_j}) > sum(V_{op_i})$ and therefore $op_i \Rightarrow op_j$. Therefore $op_i \Rightarrow op_j$ no matter what technique is used for total ordering.

Therefore in either case (op_i and op_j are generated at the same site or not), if $op_i \rightarrow op_j$, then $op_i \Rightarrow op_j$. □

Lemma 3-2. *If* $op_j \Rightarrow op_i$*, then* $op_i \nrightarrow op_j$*.*

Proof. The inverse of Lemma 3-1 is true that if $op_i \nRightarrow op_j$, then $op_i \nrightarrow op_j$.

Since $op_i \nRightarrow op_j$ is equivalent to $op_j \Rightarrow op_i$, the inverse can be restated as: if $op_j \Rightarrow op_i$, then $op_i \nrightarrow op_j$. □



Hs :
op1 < op2 < op3 < · · · < opm < opm+1 < · · · < opn
a) Before separating the history
Operation that causally precedes op
Operation concurrent to op
Partial Copy of Hs :
op1 < op2 < op3 < · · · < opm
equivalent
equivalent
Equivalent Hs :
op1' < · · · < opi' opi+1' < · · · < opm' opm+1 < · · · < opn
seq1
seq2
b) After separating the history

Figure 3-28 Separating the operations history using partial history copy

When a remote operation op arrives at site S_i, there is op_m in the history such that $op_m \Rightarrow op \Rightarrow op_{m+1}$. Since op $\Rightarrow op_{m+1}$, op_{m+1} and all other operations after op_{m+1} do not precede *op* (Lemma 3-2), they stay at their respective position in the history and only $[op_1 \dots op_m]$ needs to be rearranged and therefore copied (Figure 3-28). In other words, only operations that totally precede *op* ($\forall i$: $op_i \Rightarrow op$) need to be copied. The total ordering mechanism defined in Section 3.2.2 is used to determine the total precedence ($op_i \Rightarrow op$). The partial history copy procedure is outlined in Figure 3-29.

```
partial_ history_copy ($H_S$, $op$) {
        $H_S$' = [ ]; /* Initialize an empty history copy */
        $j = 1$;
        while ($op_j \Rightarrow op$ AND $j \leq N$) {
                $H_S$' = $H_S$' + [$op_j$];
                $j = j + 1$;
        }
        return $H_S$';
}
```

Figure 3-29 Partial history copy procedure

Although this technique is expected to minimise memory and processing usage over time, it consumes additional processing power when tracing the history to find the op_m. Therefore this technique is compared with the full history copy approach in the empirical study presented in section 3.6 to quantitatively evaluate this algorithm.

Based on the above discussion, the following subsections present the detail of the proposed operation integration algorithm. Each subsection describes the procedure of each of the following phases: *local operation execution*, *remote operation reception*, and *remote* *operation execution*. The performance evaluation of the proposed algorithm is presented later in section 3.6.

Local Operation Execution

When a user updates the local replica, the site generates an operation that realizes the user's intention. The generated operation is timestamped with the site's logical clock, $LC_{op} = LC_{S_{op}}$, and it will carry the state vector V_{op} for causality preservation purposes. The operation is immediately executed at the local replica. It is then broadcast as a tuple <*op*, LC_{op}, V_{op}, S_{op}>, where *op* is the operation, LC_{op} is the logical clock of the operation (equal to the logical clock of the site when the operation is generated), V_{op} is the state vector of the originator site when the operation is generated and S_{op} is the id of the originator site. Figure 3-30 presents the procedure invoked by a site during this phase.

```
void execute_local_operation(op) {
        $LC_{op} = LC_S$;
        $LC_S$++; /* Increment logical clock of site S */

        $V_{op} = V_S$;
        $V_S[S]$++; /* Update the state vector */

        execute(op);
        broadcast(<op, $LC_{op}$, $V_{op}$, $S_{op}$>);

        /* append operation into the site history */
        $H_S = H_S$ + <op, $LC_{op}$, $V_{op}$, $S_{op}$>;
}
```

Figure 3-30 Local operation execution procedure

Remote Operation Reception

Ellis et al. [46] defined: $V_{S_i} \geq V_{S_j}$ if each component of V_{S_i} is greater than or equal to the corresponding component in V_{S_j} ($V_{S_i}[k] \geq V_{S_j}[k]$, $\forall k$: $1 \leq k \leq N$). This means that site S_i has already executed all operations that have been executed by site S_j. To preserve causality, a remote operation *op* that arrives at site *S* will only be executed if all operations that causally precede it have already been executed by site *S*. In other words, site *S* will only execute *op* when $V_S \geq V_{op}$. The following procedure (Figure 3-31) is invoked by a site during this phase.

```
void rcv_remote_op(<op, LC_op, V_op, S_op>) {
        wait until $V_S[i] \geq V_{op}[i]$, ($\square i$: $1 \leq i \leq N$);

        /* execute remote operation by calling
         * the procedure defined in the next phase */
        execute_remote_op(<op, LC_op, V_op, S_op>);
}
```

Figure 3-31 Remote operation reception procedure

Remote Operation Execution

This phase is invoked when the remote operation *op* is causally ready. The remote operation reception procedure ensures that there is no operation in H_S that *op* causally precedes. Operations in H_S fall into two categories: operations that causally precede *op* and operations that are concurrent to *op*. Based on SOCT3, the execution of remote operation *op* involves four steps:

1. Transform *op* to take into account all concurrent operations in H_S.
2. Execute *op* and place it at the end of H_S.
3. Re-order H_S based on the total ordering scheme ($op_i \Rightarrow op_{i+1}$).
4. Trim the operation history.

In step 1, the history H_S has to be separated into two sequences $H_{S,\mathrm{p}}$ and $H_{S,\mathrm{c}}$ such that $H_{S,\mathrm{p}}$ consists of operations that causally precede *op*, $H_{S,\mathrm{c}}$ consists of operations concurrent to *op*, and $H_{S,\mathrm{p}} \bullet H_{S,\mathrm{c}} = H_S$. Backward transpositions are used to move preceding operations backward in H_S so that all preceding operations appear before the concurrent operations (Figure 3-15). Operation *op* is then forward transposed against all operations in

$H_{S,c}$ to become $op' = op^{op_{c1} \bullet op_{c2} \bullet \ldots \bullet op_{cy}}$. The procedure for separating history, adopted from SOCT3, is outlined in Figure 3-32.

```
void transpose_backward(j) {
        /* Get operation in position j and j-1 in the history */
        $< op_j, LC_{op_j}, V_{op_j}, S_{op_j} >= H_S[j]$;
        $< op_{j-1}, LC_{op_{j-1}}, V_{op_{j-1}}, S_{op_{j-1}} >= H_S[j-1]$;
        /* Swap operations in position j and j-1 in the history */
        $(op_j, op_{j-1}) = swap(op_{j-1}, op_j)$;
        $H_S[j-1] = < op_j, LC_{op_j}, V_{op_j}, S_{op_j} >$;
        $H_S[j] = < op_{j-1}, LC_{op_{j-1}}, V_{op_{j-1}}, S_{op_{j-1}} >$;
}

int separate($H_S$, $<op, LC_{op}, V_{op}, S_{op}>$) {
        $n_1 = 0$;
        $n$ = size of $H_S$;
        for $i = 1$ to $n$ do
                $< op_i, LC_{op_i}, V_{op_i}, S_{op_i} >= H_S[i]$;
                if $V_{op_i}[S_{op_i}] < V_{op}[S_{op_i}]$ then /* $op_i$ prec. $op$ */
                        for $j = i$ downto $n_1 + 2$ do
                                transpose_backward($j$);
                        endfor;
                        $n_1 = n_1 + 1$;
                endif;
        endfor;
        return $n_1$;
}
```

Figure 3-32 History separation procedure

In step 2, site S executes op' instead of op and places it at the end of H_S as op_{n+1}. In step 3, using backward transformation, op_{n+1} is then shifted backward in the history until it reaches position k where $op_{k-1} \Rightarrow op \Rightarrow op_{k+1}$ (Figure 3-21). After the remote operation is executed and placed in the history, the history trimming may be executed. It does not have to be executed after every remote operation execution (its frequency may be configured depending on the implementation). The more often it is invoked, the lesser is the average history size, but the more computations the device needs to make (hence more processing power consumed). As the number of participating sites increases, the less often the history trimming procedure needs to be invoked since the more concurrent operations may be generated at one given time. The complete procedure invoked for this phase is outlined in Figure 3-33.

```
void execute_remote_op(<op, LC_op, V_op, S_op>){
    $H_S' = partial\_history\_copy(H_S, op)$; /* partially copy $H_S$ in $H_S'$*/
    $n_1 = separate(H_S', <op, LC_{op}, V_{op}, S_{op}>)$;
    $n$ = size of $H_S$;

    for $i = n_1 + 1$ to $n$ do
        $<op_i, LC_{op_i}, V_{op_i}, S_{op_i}> = H_S'[i]$;
        $op = forward\_transform(op_i, op)$;
    endfor;

    $execute(op)$;
    $H_S = H_S + <op, LC_{op}, V_{op}, S_{op}>$;
    $n = n + 1$;

    $V_S[S_{op}]$++; /* Update the state vector */
    $VT_S[S_{op}] = V_{op}$; /* Update the vector table for history trimming purpose */
    $VT_S[S_{op}][S_{op}]$ ++;
    $LC_{op} = \max(LC_S, LC_{op} + 1)$; /* Update the logical clock */
    $j = n$;

    /* Place the remote operation in its proper location in the history */
    while ($op_j \Rightarrow op_{j-1}$) {
        /* Shift op backward until pos. k where $op_{k-1} \Rightarrow op \Rightarrow op_{k+1}$ □/
        $transpose\_backward(j)$;
        $j = j - 1$;
    }

    /* Trim the history as necessary */
    trim_history();
}
```

Figure 3-33 Remote operation execution procedure

3.5.2. Proposed Operation Transformation Rules

As discussed above, the operational transformation framework consists of two major elements: the operation transformation rules and the operation integration algorithm.

Although much has been done to improve the operation integration algorithm for processing remote operations, there has not been much discussion or evaluation on the correctness of the existing operation transformation rules. During the development and the evaluation of the proposed operation integration algorithm presented in this thesis (section 3.5.1), two problems were identified that may cause operations to be incorrectly transformed in some scenarios. The first problem occurs in a scenario that involves swapping a *delete* operation with an *insert* operation. The solution to this problem can also be used to solve the ERV puzzle. The second problem occurs when two or more users generate identical (duplicate) operations (operations with the same intentions). In each of the following sections, each problem is discussed with a simple scenario and the solution is proposed to handle each respective problem.

Swapping of Deletion and Insertion Operations

Suppose there are two sites, site 1 and site 2, participating in a collaboration session (Figure 3-34). The user at site 1 generates an operation op_1 = *insert*(3,'X') with intention to insert 'X' between 'B' and 'C'. Concurrently, the user at site 2 generates an operation op_2 = *delete*(3) to delete character 'C'. When op_1 arrives at site 2, it is forward transformed against op_2 to become *insert*(3,'X'): *FT*(*insert*(3,'X'), *delete*(3)) = *insert*(3,'X'). The user at site 1 then generates another operation op_3 = *insert*(4,'Y'). Site 2 needs to reorder op_2 and op_1 to allow op_3 to be forward transformed against op_2 at the correct context. Reordering op_2 and op_1 means swapping the two operations as follows:

swap(op_2 = *delete*(3), op_1 = *insert*(3,'X')) = (op_1', op_2'), where

op_1' = *BT*(op_1, op_2) = *BT*(*insert*(3,'X'), *delete*(3)) = *insert*(3,'X'), and

op_2' = *FT*(op_2, op_1') = *FT*(*delete*(3), *insert*(3,'X')) = *delete*(4).

Thus, when op_3 arrives at site 2, it is forward transformed against the final variant of op_2, which is *delete*(4), to become *insert*(4,'Y'). Figure 3-34 shows that both sites result in a consistent state.

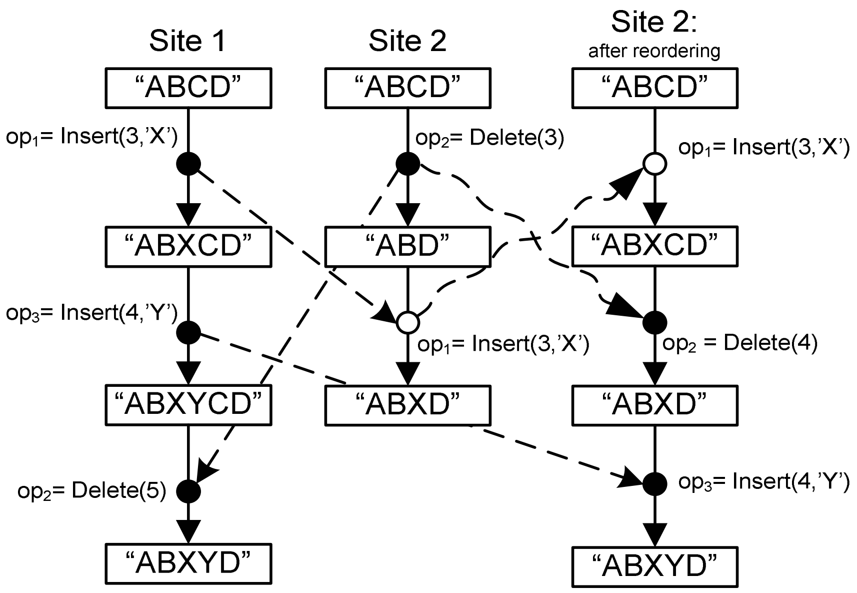


Figure 3-34 Correct backward transformation

However, if the user at site 1 generates op_1 = *insert*(4,'X') instead of *insert*(3,'X') (Figure 3-35), op_1 will also arrive at site 2 and be transformed to become *insert*(3,'X'):

FT(*insert*(4,'X'), *delete*(3)) = *insert*(3,'X'). Thus when site 2 swaps op_2 and op_1, the result is the same as the above example (Figure 3-34) as follows:

swap(op_2 = *delete*(3), op_1 = *insert*(3,'X')) = (op_1', op_2'), where
op_1' = *BT*(op_1, op_2) = *BT*(*insert*(3,'X'), *delete*(3)) = *insert*(3,'X') and
op_2' = *FT*(op_2, op_1') = *FT*(*delete*(3), *insert*(3,'X')) = *delete*(4).

After swapping, op_1 becomes *insert*(3,'X') violating the intention of the user of site 1. Furthermore, when op_3 arrives at site 2, it is transformed against op_2 to become *insert*(3,'X') and that leads site 2 to a state inconsistent with site 1.

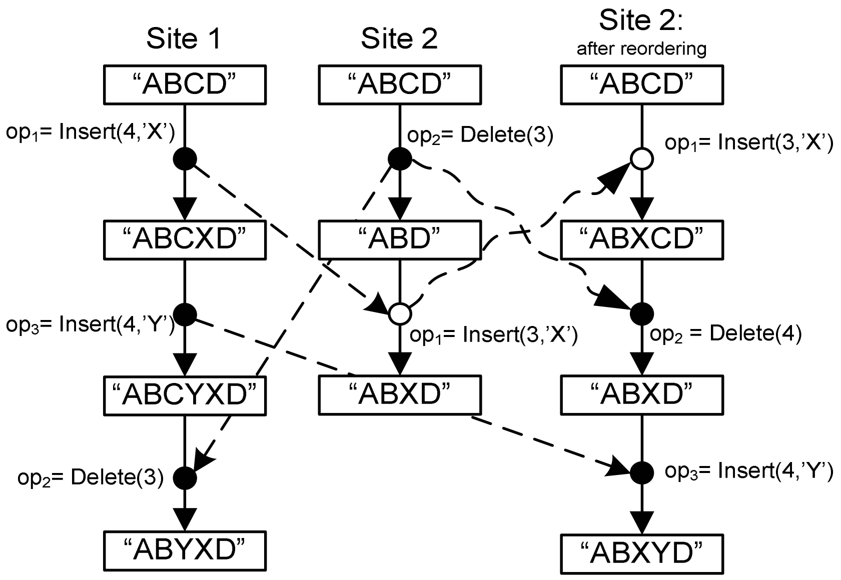


Figure 3-35 Incorrect backward transformation

This problem occurs because two different insertion operations (*insert*(4,'X') and *insert*(3,'X')), transformed forward against *delete*(3), are transformed into the same operation *insert*(3,'X') (Figure 3-36). Therefore, the backward transformation needs to ensure that an operation, which was previously transformed forward, will revert to its original operation after being transformed backward. In the case of Figure 3-35, op_1 is originally *insert*(4,'X'). After forward transformation against op_2, it becomes *insert*(3,'X'). Then after swapping with op_1, instead of reverting to its original form of *insert*(4,'X'), it becomes *insert*(3,'X') which violates the user intention. A correct backward transformation needs to ensure that op_1 becomes its original form of *insert*(4,'X') as illustrated in Figure 3-37.

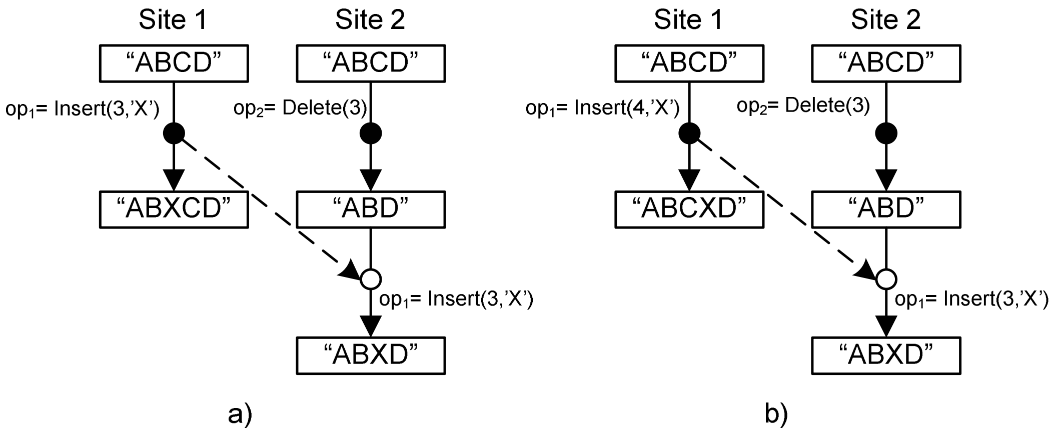


Figure 3-36 Transformation of an insert against a delete operation

In Figure 3-36a, the intention of operation op_1 is to insert 'X' after 'B' and before 'C', while in Figure 3-36b, the intention is to insert 'X' after 'C' and before 'D'. However, since site 2 deletes 'C', both insert operations end up inserting 'X' at the same position, which is after 'B' and before 'D'. In order to distinguish the two different scenarios, *op₁* needs to have the information whether *op₂* deletes a character before or after the inserted character. In Figure 3-36a, *op₂* deletes a character after the inserted character. Meanwhile in Figure 3-36b, *op₂* deletes a character before the inserted character. To store this information, each insertion operation will have two extra parameters, *p* and *q*, where *p* is a set of ids of operations that delete a character before the inserted character and *q* is a set of ids of operations that delete a character after the inserted character, thus the new insertion operation is written as *insert*(x_i, a_i, p_i, q_i). This technique was first introduced by Suleiman et al. [126] to solve the conflict between two insertion operations of a character at the same position. However, they did not mention this problem and they did not discuss any further applicability of this technique for solving this problem, nor did they provide the backward transformation algorithm for swapping operations in the history.

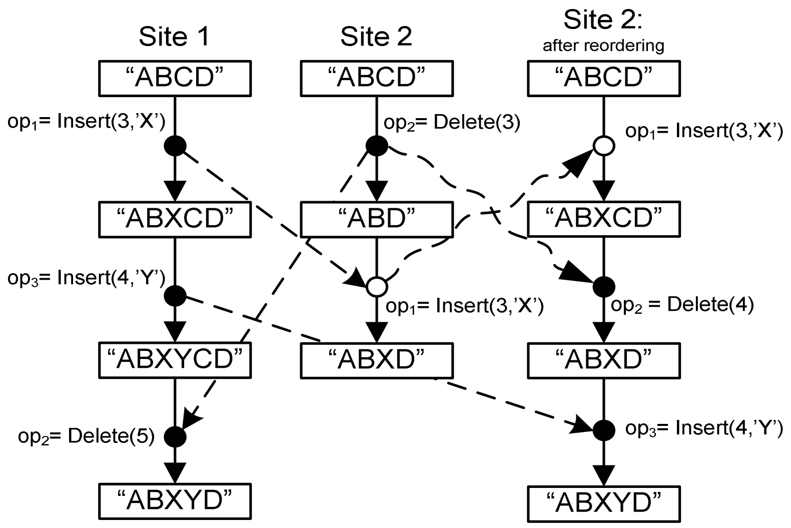


Figure 3-37 Correct backward transformation

Using the new insertion operation (with the additional two parameters, *p* and *q*), some of the forward transformation rules and the corresponding backward transformation rules, especially the ones that involve insert operation, are modified as follows:

```
$FT(op_1 = insert(x_1, a_1, p_1, q_1), op_2 = delete(x_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 -1, a_1, p_1 + [op_j], q_1)$
        else $op_1' = insert(x_1, a_1, p_1, q_1 + [op_2])$;

$FT(op_1 = insert(x_1, a_1, p_1, q_1), op_2 = insert(x_2, a_2, p_2, q_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 + 1, a_1, p_1, q_1)$;
        if $x_1 < x_2$ then $op_1' = insert(x_1, a_1, p_1, q_1)$;
        if $x_1 = x_2$ then
                if $p_1 \cap q_2 \neq \emptyset$ then $op_1' = insert(x_1 + 1, a_1, p_1, q_1)$;
                else if $p_2 \cap q_1 \neq \emptyset$ then $op_1' = insert(x_1, a_1, p_1, q_1)$;
                else if $a_1 = a_2$ then $op_1' = id$;
                else if $a_1 \neq a_2$ then
                        if $p_{op_1} > p_{op_2}$, then $op_1' = insert(x_1, a_1, p_1, q_1)$;
                        if $p_{op_1} < p_{op_2}$, then $op_1' = insert(x_1 + 1, a_1, p_1, q_1)$;
                endif;
        endif;

$BT(op_1 = insert(x_1, a_1, p_1, q_1), op_2 = delete(x_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 + 1, a_1, p_1 - [op_2], q_1)$;
        if $x_1 < x_2$ then $op_1' = insert(x_1, a_1, p_1, q_1 - [op_2])$;
        if $x_1 = x_2$ then
                if $op_2 \in p_1$ then $op_1' = insert(x_1 + 1, a_1, p_1 - [op_2], q_1)$;
                if $op_2 \in q_1$ then $op_1' = insert(x_1, a_1, p_1, q_1 - [op_2])$;
                if $op_2 \notin p_1$ AND $op_2 \notin q_1$ then $op_1' = insert(x_1, a_1, p_1, q_1)$;
endif;

$BT(op_1 = insert(x_1, a_1, p_1, q_1), op_2 = insert(x_2, a_2, p_2, q_2)) = op_1'$, where
        if $x_1 > x_2$ then $op_1' = insert(x_1 - 1, a_1, p_1, q_1)$
        else $op_1' = insert(x_1, a_1, p_1, q_1)$;
```

Figure 3-38 illustrates the use of the newly defined forward and backward transformation rules, and shows that the transformations lead to consistent states and are able to bring the transformed operation back into its original form. The new parameters, *p* and *q*, provide information to the insert operation of whether it inserts the character before or after the deleted character, so it can be transformed backward correctly against the delete operation.

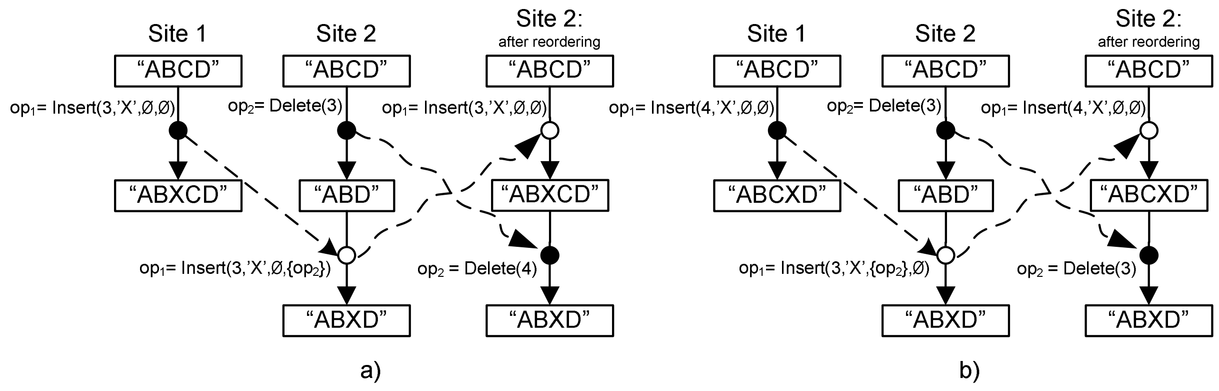


Figure 3-38 Backward transformation of insert against delete operation

This solution also solves the ERV puzzle: it retains the original intention of the insert operation by storing all concurrent operations that delete any characters before or after the inserted character. Figure 3-25 and Figure 3-26 show how the proposed technique is able to solve the ERV puzzle.

This problem has previously been identified and called 'lossy transformation' by Sun et al [128]. A solution was also provided to recover the lost transformation information caused by operations deleting overlapping characters. To correctly backward-transform op_1 against op_2 (BT(op_1, op_2)) requires: (1) detecting whether the transformation of the two operations are 'lossy', (2) copying the history starting from the first operation concurrent to op_1, (3) shifting op_2 forward in the history copy to the right most position, and finally (4) transforming the original version of op_1 against the previously organised history copy. The idea was to redo the forward transformation of the original form of op_1 against all concurrent operations in the history excluding op_2. This solution, however, may impose more overhead as compared to the proposed transformation rules since the proposed transformation rules simply transform the operations with the help of two additional parameters without the need of lossy detection, history copying and redoing the original forward transformation.

Identical Operations

The other problem with the existing operation transformation rules involves two identical operations. An operation is identical to the other operation if both of them realise the same user intention. Formally, two operations are identical (duplicating) if:

- they are concurrent to each other, and
- the former carries out the same operation as the latter (both are deletions or insertions), and
- they delete/insert the same character at the same position in the same context.

In Figure 3-39a, op_1 and op_2 are identical operations: they delete the same character at the same position (they are both *delete*(3)), they are concurrent to each other, and they are generated at the same context. In Figure 3-39b, however, even though op_1 and op_3 have different syntax, they are identical because they realise the same user intention, which is inserting character 'X' between 'C' and 'D'. Their syntaxes are different due to different generation context. If op_3 were generated before op_2 ($GC(op_3) = GC(op_1)$ consequently), op_3 would have been *insert*(4,'X'), which is the same as op_1.

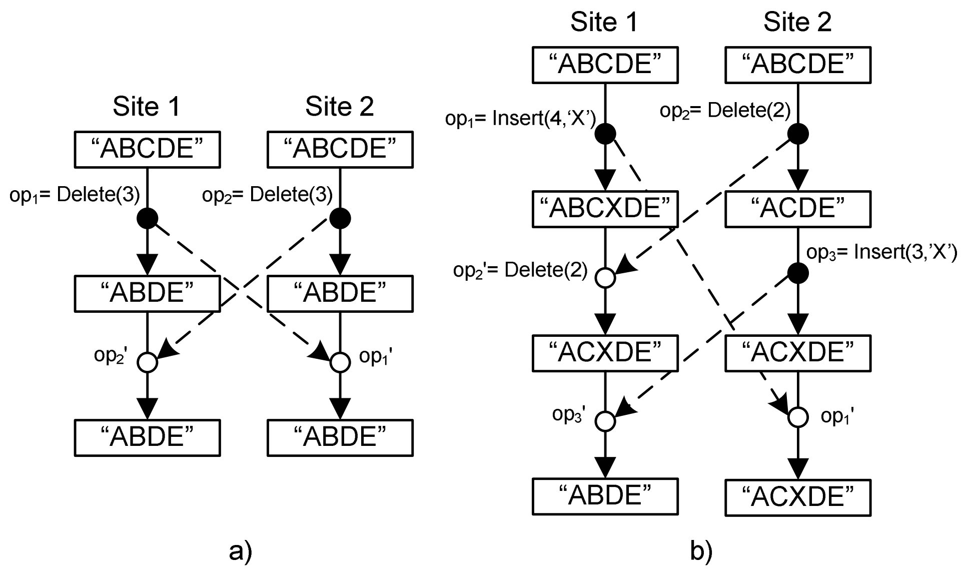


Figure 3-39 Identical operations

The way to handle identical operations has not been discussed in existing work. When a site receives a remote operation, it will be forward transformed against all concurrent operations in the history. When an operation is found to be identical to the remote operation, the remote operation is simply nullified. The forward transformations for identical operations are defined as follows:

> *FT*(*op_1* = *insert*(*x_1*, *a_1*), *op_2* = *insert*(*x_2*, *a_2*)) = *op_1*' where
> if *$x_1 = x_2$* and *$a_1 = a_2$* then *op_1*' = *id*;
>
> *FT*(*delete*(*x_1*), *delete*(*x_2*)) =
> if *$x_1 = x_2$* then return *op_1*' = *id*;

Using the previously defined forward transformations rules, it can be seen in Figure 3-40 that site 1 transformed *op_2* against *op_1* to become *op_2 = id*, site 2 transformed op1 to become *op_1 = id*, and both sites end up in consistent states. However, if the history is to be reordered at site 2 based on the total ordering scheme, *op_2* and *op_1* need to be swapped: s*wap*(*op_2*, *op_1*) = (*op_1*', *op_2*'), where *op_1*' = *BT*(*op_1*, *op_2*) and *op_2*' = *FT*(*op_2*, *op_1*'). Backward transformation rules that involve operation *id* have never been discussed; therefore the result of the backward transformation of *op_1* is unknown.

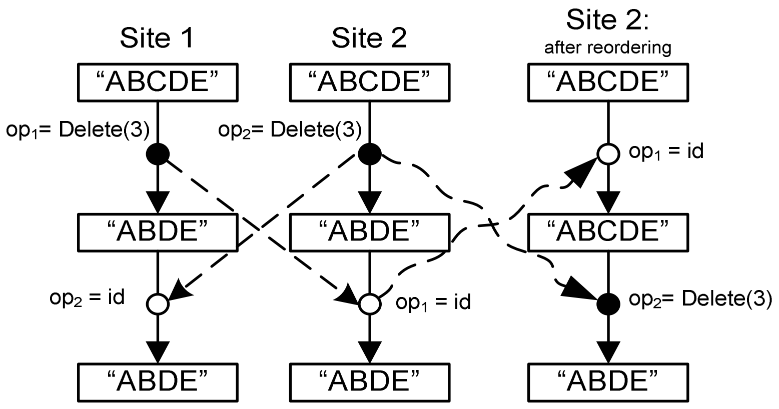


Figure 3-40 Backward transformation of identical operations

In Figure 3-40, it is assumed that operation *id* will stay as it is, and site 2 still leads to the correct state after reordering. In Figure 3-41 however, site 1 leads to an inconsistent state after the backward transformation that involves operation *id*. When site 1 receives op_3, op_3 needs to be transformed against op_1 since $op_1 \| op_3$. Since $op_2 \rightarrow op_3$ and $op_1 \| op_3$, op_1 and op_2 need to be swapped so that op_3 has the same context as op_1 before site 1 transforms op_3. After swapping operations, op_3 is transformed against op_1 to become *insert*(3,'X'). However, the execution of this operation leads site 1 to a different state from site 2. Therefore, it is necessary to formulate the correct operation transformations rules for identical operations.

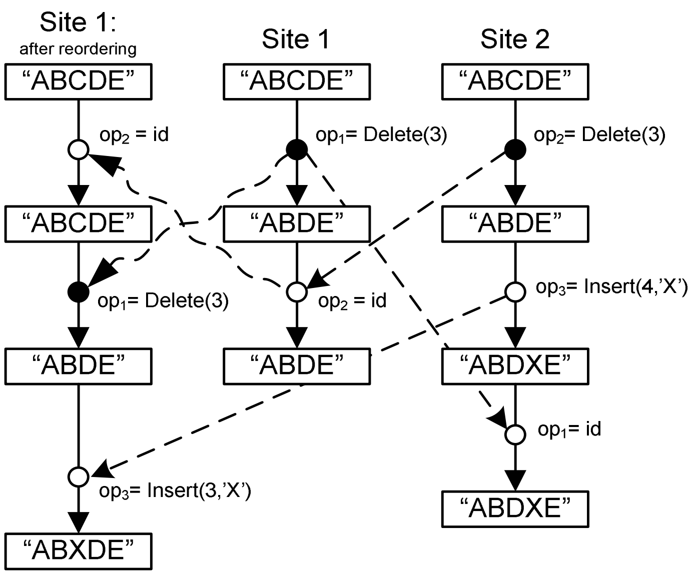


Figure 3-41 Identical operations lead to inconsistent states

Consistency can only be reached if the history can properly be reordered such that $GC(op_1) = GC(op_3)$. It is obvious that $GC(op_3)$ includes op_2 (*delete*(3) operation), which is identical to op_1. Thus when op_3 is received at site 1, site 1 has to reorder the history such that $GC(op_1) = GC(op_3)$, which means that $GC(op_1)$ has to include the *delete*(3) operation. A proper operations swap is illustrated in Figure 3-42 where both sites end in a consistent state.

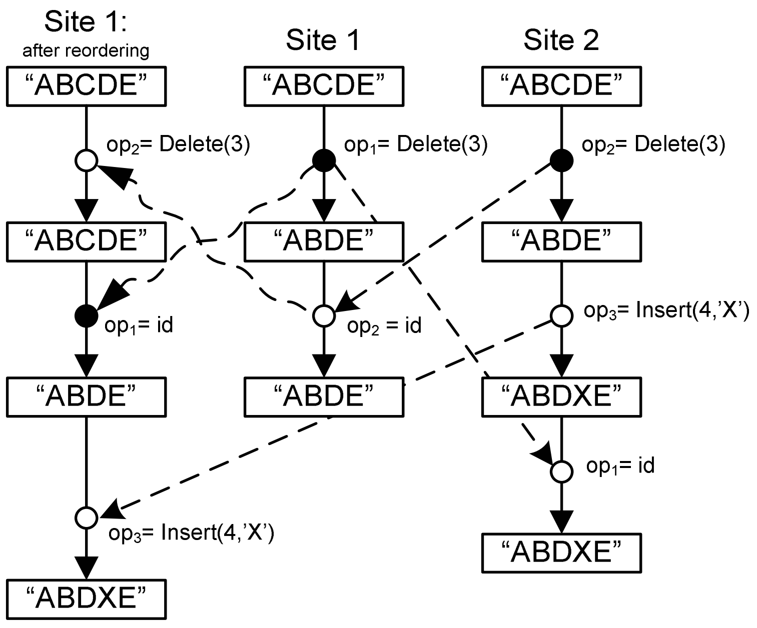


Figure 3-42 Identical operations lead to consistent states

However, the *id* operation is generic and there is no way the site can know which operation it is identical to. Therefore, to solve this problem, the identical operation once transformed has to have the information of which operation it is identical to. In a multi-user collaboration session, two or more users may generate identical operations concurrently. Hence, the transformed operation may contain a list of operations which it is identical to, and a new operation named *dup*(*ops*) is proposed, where *ops* is the unique ids of the operations that it is identical to. When an operation is transformed against an identical operation, say op_x, it will become operation *dup*([op_x]) where *ops* contains operation op_z. The forward transformation rules for identical operations are therefore modified as follows.

FT(*op1* = *insert*(*x1*, *a1*), *op2* = *insert*(*x2*, *a2*)) = *op1*' where
    if *x1* = *x2* and *a1* = *a2* then *op1*' = *dup*([*op2*]);

FT(*op1* = *delete*(*x1*), *op2* = *delete*(*x2*)) = *op1*' where
    if *x1* = *x2* then *op1*' = *dup*([*op2*]);

Since *dup* is a new operation, forward transformation rules need to be defined to properly transform *dup* operation against another operation or vice versa. There are a few possible transformation scenarios that involve *dup* operations.

1. Transformation of a *dup* operation against an operation that is not a *dup* operation.
2. Transformation of a non *dup* operation against a *dup* operation.
3. Transformation of a *dup* operation against another *dup* operation and they are not identical (i.e. their original operations realise different user intentions).
4. Transformation of a *dup* operation against another *dup* operation and they are identical (i.e. their original operations realise the same user intention).

In the first, second, and third scenarios, the transformation will not change the operation that is being transformed. The operation that is being transformed will remain the same. In the fourth scenario, however, both *dup* operations involved are identical; therefore they will add each other into their lists of identical operations. This is to make sure each *dup* operation has the information of what operations it is identical to so that it can be backward transformed correctly against its identical operation.

Therefore, the transformation rules that involve *dup* operations are defined as follows.

```
$FT(op_1, op_2 = dup(ops_1)) = op_1'$, where $op_1' = op_1$;

$FT(op_1 = dup(ops_1), op_2) = op_1'$, where
        if $op_2$ is not a dup operation, $op_1' = op_1$;
        if $op_2 = dup(ops_2)$, then
                if $ops_1 \cap ops_2 = \emptyset$ then $op_1' = op_1$;
                if $ops_1 \cap ops_2 \neq \emptyset$ then
                        $op_1' = op_1$; $ops_1 = ops_1 + [op_2]$; $ops_2 = ops_2 + [op_1]$;
```

Figure 3-43 illustrates forward transformation of an operation against its identical operation. When site 1 receives op_2, it is transformed to become $dup([op_1])$ since op_2 is identical to op_1. When op_3 arrives at site 1, it is first transformed against op_1 to become $dup([op_1])$. Then it is transformed against op_2 causing both op_2 and op_3 to become $dup([op_1, op_2])$. By this time, op_3 and op_2 know that they are identical to each other and they are identical to op_1.

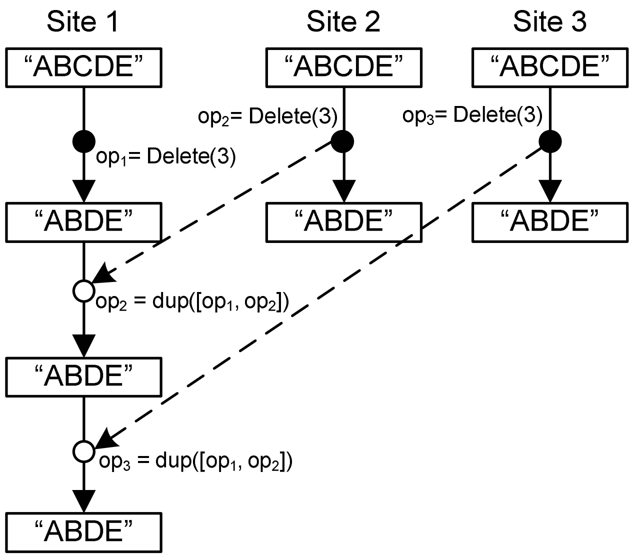


Figure 3-43 Identical operations signified by *dup* operations

Similarly, it is necessary to define more backward transformation rules that involve a *dup* operation, as follows:

$BT(op_1 = dup(ops_1), op_2 = dup(ops_2)) = op_1'$, where $op_1' = op_1$;

$BT(op_1, op_2 = dup(ops_2)) = op_1'$, where $op_1' = op_1$;

$BT(op_1 = dup(ops_1), op_2) =$
if $op_2 \in ops_1$, then $op_1' = op_2$
else $op_1' = op_1$;

Using the backward transformation rules, an operation received out of order can be reorganised in the history such that operations in the history are totally ordered (Figure 3-44). The scenario depicted in Figure 3-45 is the same as the one in Figure 3-41 but the proposed backward transformation rule is used to reorder the history. When site 1 receives op_3, op_1 and op_3 have to be swapped to ensure op_1 has the same context as op_3 before transforming *op₃* against *op₁*. The swapping is done such that the actual operation has to appear before the other identical operation(s) in history. So when op_1 is swapped with op_2, op_2 becomes *delete*(3) and op_1 becomes identical to op_2.

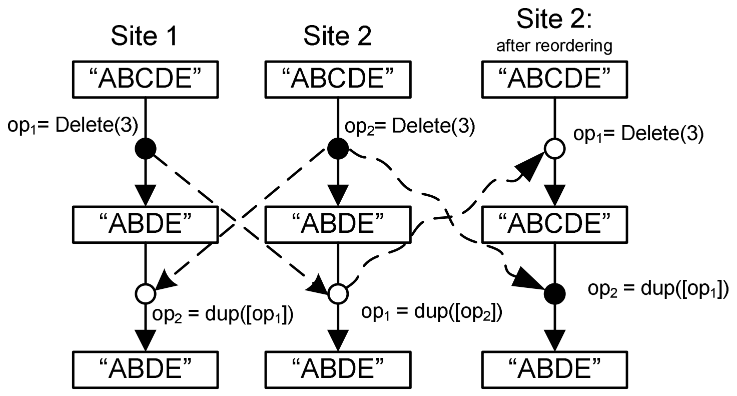


Figure 3-44 Backward transformation of a *dup* operation

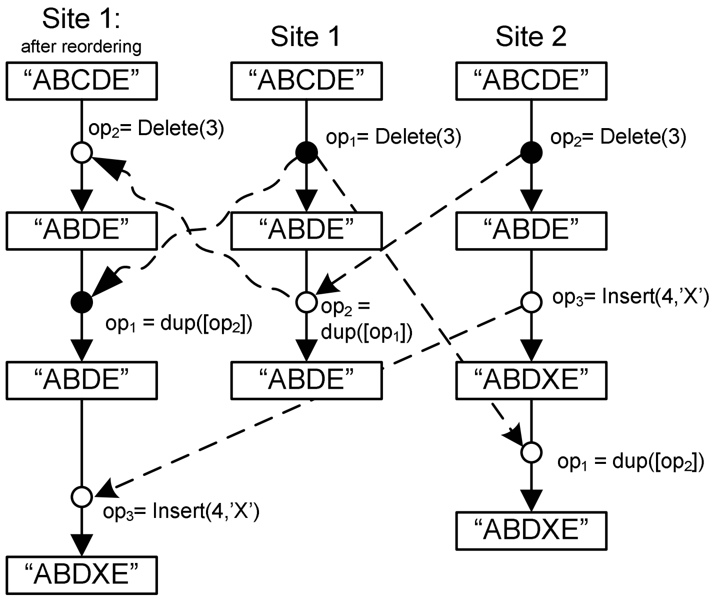


Figure 3-45 Backward transformations of *dup* operations

3.6. Performance Evaluation

The proposed algorithm can be summarised as follows: (1) it is based on SOCT3 algorithm which uses history separation to solve the dOPT puzzle and history reordering to avoid the TP2 puzzle, (2) it uses a fully distributed total ordering scheme, (3) it applies a history trimming procedure to reduce the space requirement, (4) it introduces a partial history copy procedure to reduce the memory requirement during the execution of remote operations, and (5) it uses modified operation transformation rules to ensure operations are transformed correctly to preserve user intentions.

The proposed operation integration algorithm, however, has alternatives for implementation. Therefore, the impact on storage and processing of each of the alternatives has been quantitatively evaluated to determine the most efficient design alternative. Firstly, the fully distributed total ordering scheme can be achieved using the state vector technique or Lamport's logical clock. Secondly, the algorithm works with or without history trimming. With history trimming, the algorithm reduces the space requirement to maintain the operation history. However, it requires additional processing power. Therefore, an empirical evaluation is necessary to justify the use of the history trimming procedure. Thirdly, the partial history copy reduces the memory requirement during each remote operation execution. However, like the history trimming technique, it requires additional processing power.

The design factors and alternative implementations discussed in this section can be summarised as follows:

1. State Vector(SV) vs. Lamport's Clock(LC),
2. History Trimming(HT) vs. No History Trimming (NoHT), and
3. Full History Copy (FC) vs. Partial Copy (PC).

Based on those aspects, the following eight algorithm designs are possible from combinations of the above techniques:

1. SV, NoHT, and FC
2. LC, NoHT, and FC
3. SV, HT, and FC
4. LC, HT, and FC
5. SV, NoHT, and PC
6. LC, NoHT, and PC
7. SV, HT, and PC
8. LC, HT, and PC

This section presents an empirical study which compares a number of candidate algorithm variations in order to determine which one is most efficient in terms of performance and resource utilisation and thus most suitable for use in a mobile context. The experiments were based on simulations written in the Java programming language and were run on a 1.6GHz PC with each site is represented by a Java thread.

3.6.1. Independent Variables

Three independent variables are manipulated for each of the eight algorithm combinations identified in the previous section: *number of sites*, *number of operations* and *broadcast delay*. The higher the number of sites, the greater is the number of remote operations and thus concurrent operations. The higher the number of generated operations at each site, the larger is the history size. The impact of history trimming is expected to be more significant as the number of operations, and thus the history size, increases. The number of concurrent operations increases as this delay increases. The chosen broadcast delays are intended to be representative of realistic delays in a mobile environment, with the main intention being to investigate the trend in algorithm performance as delay increases.

3.6.2. Dependent Variables and Expected Outcomes

ISO 9126-1 [70] considers efficiency as a quality attribute comprising the capability of software to provide appropriate response and processing times (performance characteristic) and the capability of software to use appropriate amounts and types of resource during its execution (resource utilisation characteristic).

The efficiency characteristic of *performance* is operationally defined as the dependent variable e*xecution time* measured in seconds. The algorithm with the highest performance is the one with the lowest processing time, which relates to reduced power consumption and an enhanced user experience. The following are expected outcomes in terms of performance:

- **P-1**: LC will be faster than SV, since SV requires additional processing effort for the summation of the state vector elements.
- **P-2**: HT will be faster than NoHT. HT reduces history size, thus the remote operation process is expected to be faster.
- **P-3**: PC will be faster than FC. PC copies only part of the history and thus the time taken to perform history separation will be shorter.

The second efficiency characteristic of *resource utilisation* is operationally defined using two variables, *history size after operation execution* and *history copy size during operation execution*, both of which relate to memory usage.

The following are expected outcomes with regards to the resource utilisation:

- **M-1**: History size after operation execution is less for HT since HT regularly trims the history.
- **M-2**: PC uses less memory during an operation execution, since it does not copy the entire history.

Furthermore, the expected outcomes for the *performance* characteristic (**P-1**, **P-2**, and **P-3**) also indirectly influence resource utilisation since the reduced processing overhead of an operation results in lower processor utilisation.

3.6.3. Results

Figure 3-46 shows the results in terms of performance, demonstrating that **P-2** is satisfied. The design alternatives involving history trimming (HT) perform better in terms of execution time than those without (NoHT). On average, HT reduces the execution time by almost 40% (Figure 3-46) with the difference increasing as the number of sites and number of generated operations (and thus the total number of operations being exchanged) increases.

The longer the collaboration runs, the greater is the difference in history size between HT and NoHT (Figure 3-47), thus the execution time difference between HT and NoHT also increases (Figure 3-48 and Figure 3-49). However, as the broadcast delay increases, the performance of HT gets closer to NoHT Figure 3-50).

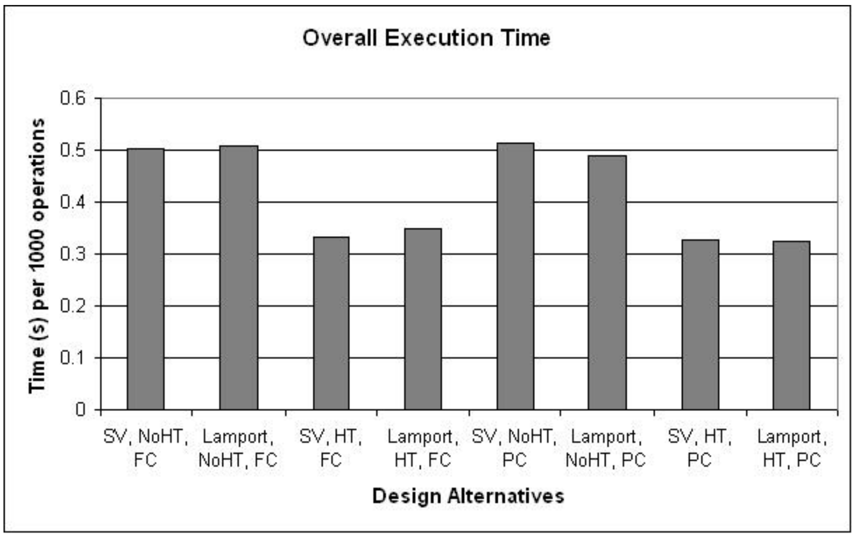


Figure 3-46 Average overall execution time

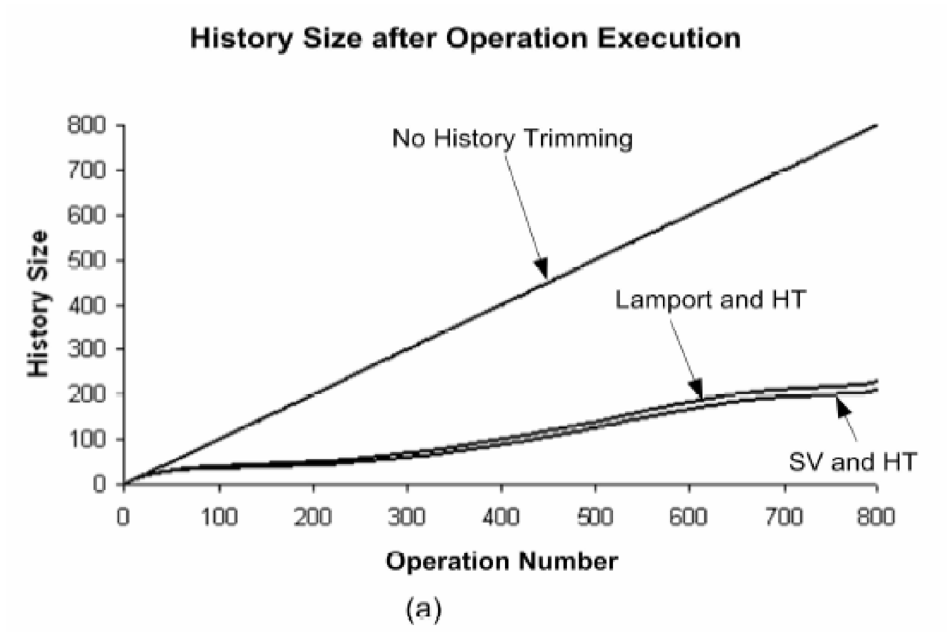


Figure 3-47 The size of the history size after operation execution

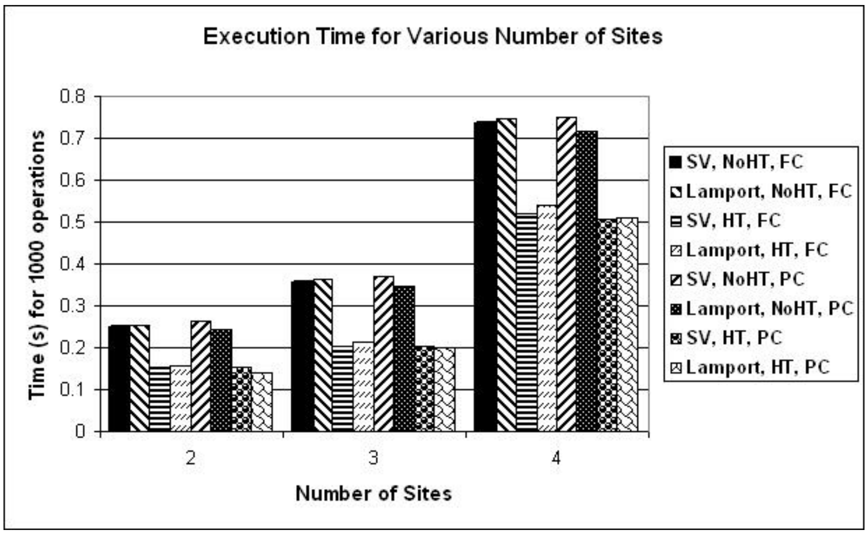


Figure 3-48 Execution time vs. number of sites

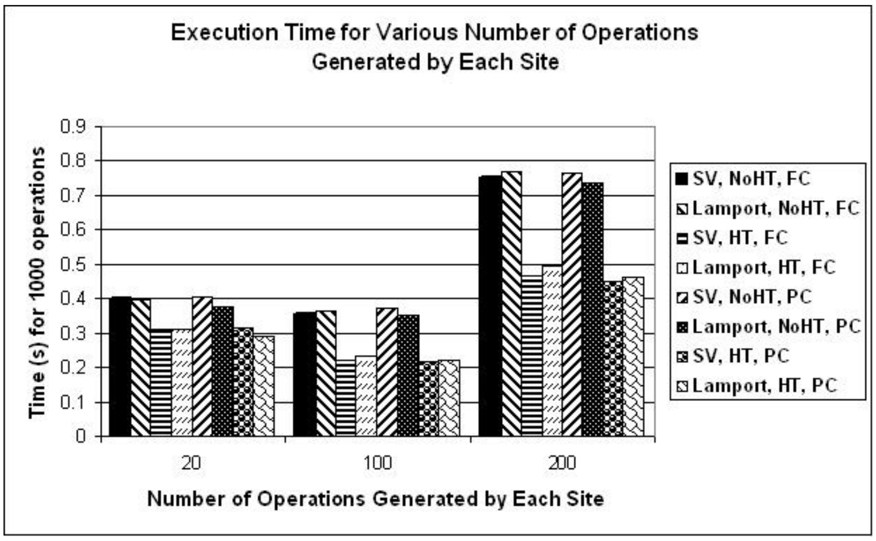


Figure 3-49 Execution time vs. number of operations

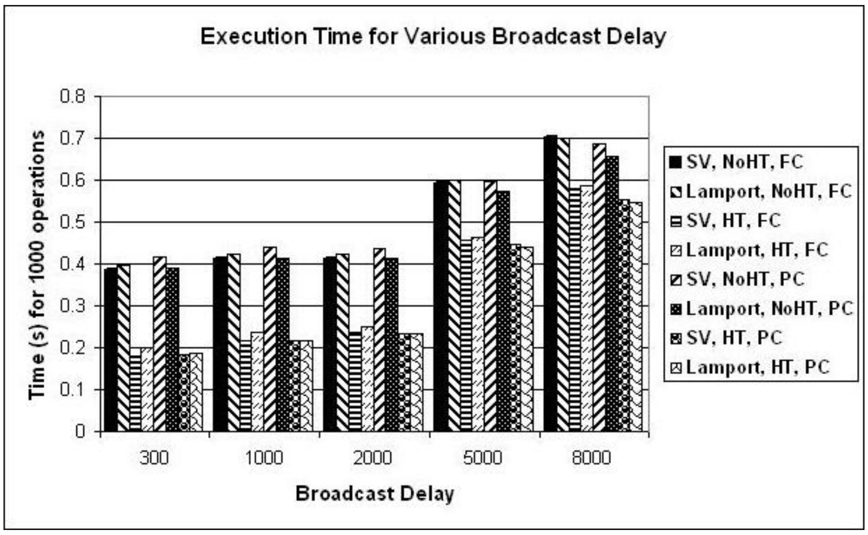


Figure 3-50 Execution time vs. broadcast delay

When the delay is 300ms, the difference is approximately 50% and when the delay is 8000ms, the difference is less than 20%. This is because the longer the broadcast delay, the less often the history gets trimmed. An operation in the history of a site can only be trimmed if that site knows that all other sites have already executed that operation, as derived from the state vector of the received operations (see section 3.4). Consequently, when the network delay is high, this information arrives later, thus the history is trimmed later than when the network delay is low.

While **P-2** is strongly displayed by the graphs, expected outcome **P-1** does not hold since there are situations where SV is better than LC, and vice versa. Therefore, based on execution time, there is no clear reason to favour SV or LC for total ordering. The same is true with **P-3** where PC does not improve the performance in terms of execution time. Although PC saves processing power by not copying the entire history, it involves additional condition checking while copying the history and thus does not improve processor usage overall. Therefore, to help determine which total ordering technique is better overall, the resource utilisation of each technique in terms of memory usage must be considered.

Figure 3-47 supports the prediction that without trimming, history size grows linearly towards infinity since each executed operation is stored in the history for the entire duration of the collaboration session. Depending upon the implementation, this may also impact on storage requirements and increase processing overhead as parts of the history are paged to and from permanent storage. In contrast, the designs that implement History

Trimming prevent this from happening, thus supporting **M-1** when either SV or LC is used for total ordering. Of particular interest is that history trimming is more effective when SV is used in preference to LC. Therefore given that SV and LC exhibit similar performance characteristics in terms of execution time, SV is more efficient, and thus a better solution overall, since it results in less memory utilisation.

Figure 3-51 supports the expectation **M-2** that PC reduces the size of the history copy and thus requires less memory to process remote operations regardless of whether or not HT is used. Therefore, since PC is neutral in terms of performance, as measured by execution time, due to its reduced memory utilisation it is superior in terms of efficiency to FC.

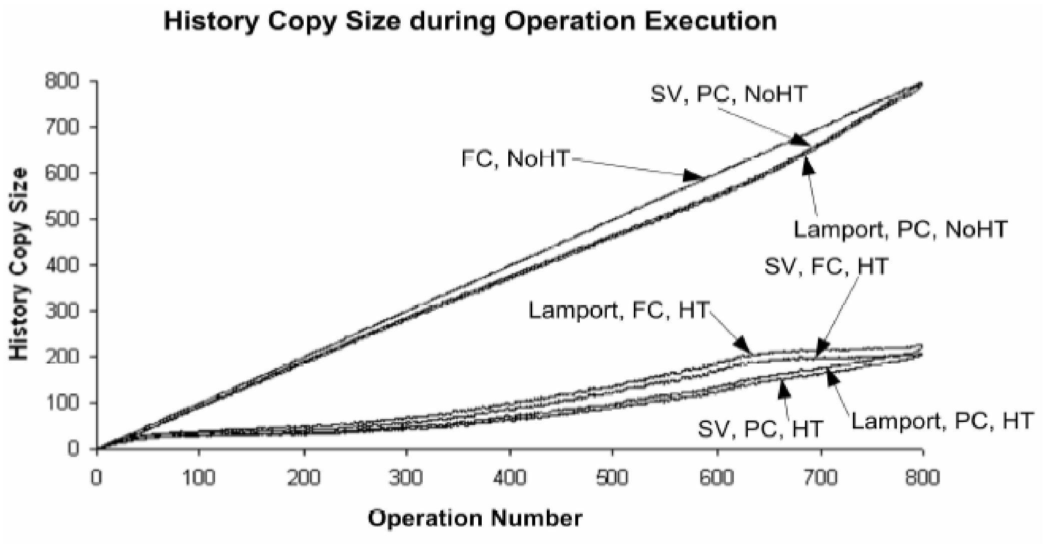


Figure 3-51 The size of the history copy for history separation purpose

In summary, HT saves processing power and consumes less memory and is thus a clear choice in comparison to NoHT. For total ordering, SV is better than Lamport since even though they are similar in terms of performance, SV has lower memory utilisation requirements. Similarly, although PC is not superior in terms of performance, it reduces memory usage as compared to FC. Therefore in summary, the algorithm design alternative that implements History Trimming and Partial History Copy, and uses State Vector for total ordering, is the most efficient and thus best choice for implementing a real-time collaboration algorithm in a mobile environment. This design alternative is also applicable to non-mobile environments making it usable by groupware systems in all types of network environments to efficiently maintain document consistency.

3.7. Conclusion

This chapter has addressed the design of consistency management algorithms for use in a real-time mobile collaboration application. Various existing work and their limitations have been described. This chapter presents a new consistency management (or concurrency control) algorithm that addresses the limitations of existing work and is more efficient in terms of performance and resource utilisation, and is thus more suitable for use in a mobile context.

The concepts of SOCT3 are used as a basis for the operation integration algorithm as it has been proven to be correct. The dependence of the algorithm on a central server is removed by incorporating known total ordering techniques so that it is applicable in a purely replicated mobile network. To reduce resource consumption, the history trimming technique is applied to reduce the memory space taken by the operation history. Furthermore, a novel partial history copying technique is introduced to reduce processing power consumption and memory space requirements during each remote operation processing.

The empirical testing indicates that the combination of history trimming, partial history copy and state vector for total ordering produces the most efficient design for use in a mobile environment. Not only does this algorithm reduce overall execution time, it also reduces resource utilisation in terms of memory or fixed storage usage thus serving as a benchmark for comparison in any further research on this topic.

During the development of the proposed algorithm, two problems to the existing operation transformation rules have been discovered: the swapping deletion and insertion operation, and the transformation of identical operations. Therefore, both problems and their proposed solutions have also been presented in this chapter.

While the proposed consistency algorithm has been evaluated based on its efficiency, future work could involve the testing of additional ISO 9126-1 quality characteristics such as reliability and security. Such work would involve further use of simulation and possibly live testing using a real application, in addition to revised experimental designs based on the derivation and operational definition of a new set of variables to quantify the effects on the different quality attributes.

As mentioned in section 3.4, future work may also involve a detailed evaluation of the most recently published work such as TTF and COT which include an empirical comparison with the proposed algorithm, and possibly an improvement to the proposed algorithm based on the newly derived concepts/principles.

Chapter 4

Conflict Management

4.1. Introduction

As described in the previous chapter, a document consistency management algorithm is an important element in real-time mobile collaborative editing, especially in a peer-to-peer network (replicated architecture). Each device holds a document replica and can concurrently generate operations. The concurrent operations have to be applied correctly in each document replica such that all replicas are consistent.

While the consistency management algorithm proposed in Chapter 3 is able to ensure consistency of the replicas, it only works when all the concurrent operations can be executed at the document replica. However, depending on the intention of the users that generate the concurrent operations, the operation may be conflicting with one or more concurrent operations. If the intention of one operation conflicts with the intention of another concurrent operation, each site has to process and resolve the conflict consistently.

In a centralised architecture, the server carries out the conflict management and resolution, and the participants may only be involved if voting is required. In a replicated architecture, on the contrary, each site has to carry out conflict management consistently and each site is involved in the conflict resolution process. Consequently, the conflicting operations need to be properly detected and each conflict has to be resolved consistently across all sites.

Conflicts can be categorised into two types: *exclusive* and *non-exclusive* conflicts. An *exclusive* conflict occurs when the conflicting operations cannot be realised at the same time, and if serially executed, the effect of the later operation will override the earlier operation. In contrast, a *non-exclusive* conflict occurs when the conflicting operations can be realised at the same time and both operations can be applied to the target without one being overridden by the other.

The consistency management algorithm proposed in the previous chapter is able to handle non-exclusive conflicts by properly transforming the conflicting operations. Therefore, the aim of this chapter is to discuss the problem of an *exclusive conflict* and to propose a conflict management algorithm that allows conflicts to be detected, managed, and resolved consistently across all sites in a mobile replicated architecture while respecting user intention. The remainder of this chapter is organised as follows: section 4.2 describes the conflict problem; section 4.3 outlines the existing work in conflict management; section 4.4 presents the proposed conflict management algorithm; and section 4.5 concludes the chapter and outlines future work.

4.2. Conflict Problem

A conflict occurs when two or more users have different intentions for editing the same part of the replicated document, for example, two users change the size of a textbox differently, or two users insert different words in a sentence. In practice, the definition of a conflict is application dependent, with possible factors being domain specific semantics, implementation details, document granularity and desired level of concurrency. A particular application may consider insertions of two different letters in the same word as a conflict while another application may not. Nevertheless, in the general case, in an object based document, a conflict occurs when two or more users are concurrently modifying the same object.

An object in a document, however, may comprise other objects. Therefore, conflicts may be defined in different levels of the object hierarchy. For example, conflict might only occur when the object being modified is the lowest in the object hierarchy. To further promote concurrency, a conflict can be defined to occur only when operations are concurrently modifying the same attribute of the same object to different values [130]. Regardless of the application domain, a conflict can generally be defined as the following: a conflict occurs when two or more users concurrently modify the same target with different intentions. The target is application-specific and could be an object, an object attribute, a word in a document, a letter in a document, a paragraph, or even a whole document.

As mentioned previously, conflict can be categorised into two types: *exclusive* and *non-exclusive* conflicts. Suppose *Alice* and *Bob* are currently collaborating on an object-based text document, with words being the targets, and they are concurrently modifying a word "and". *Alice* is adding a letter 'h' to make the word "hand" while *Bob* is changing the letter 'd' to 't' to make the word "ant". The operations are conflicting as they have different intentions, but they can be realised at the same time resulting in the word "hant". The word may then be marked as being in conflict and the conflict can then be resolved. This is an example of a non-exclusive conflict where the conflicting operations can be realised in the same document at the same time and both operations can be applied to the target without overriding each other. However, if *Alice* is changing the font size of the word to size 12 and *Bob* is changing the font size to size 14, an exclusive conflict occurs. Both operations cannot be realised together, and one operation will override the other depending on the execution order. In other words, an exclusive conflict happens if the operations causing the conflict are non-transformable against each other.

Non-exclusive conflicts can be resolved by using the proposed algorithm as described in Chapter 3 where operational transformation is used to transform one of the conflicting operations against the other to preserve both intentions. Not only does it ensure consistency in the presence of transformable operations, it also minimises resource usage, making it suitable for use in mobile environments. Users can then choose to keep all operations or to undo some operations or to generate operations to fix errors created by the conflict. Exclusive conflicts, on the contrary, cannot be resolved using the operation transformation technique since concurrent operations are transformed and executed in a serialised manner. Therefore one of the intentions may be overridden by the other. For example, in SpeakEasy [45], synchronization conflicts – such as two users adding the same component, or one offline member removing a component while another adds it—are dealt with in a very simplistic way. The update with the latest timestamp “wins” and it provides a user interface to undo any undesired synchronization changes. This violates the intention of at least one user.

This chapter, therefore, discusses the problem of exclusive conflicts and proposes a conflict management technique to handle exclusive conflicts and consequently facilitate users to resolve the conflict.

4.3. Related Work

Transactional processing has been a major topic in dealing with concurrent updates to a document/database [36]. However, transactional processing uses a centralised server and it is used mainly in database applications where users read the document from the server before they make changes and write back to the server. Transactional concurrency control focuses on grouping the operations together into an atomic transaction and serializing the transactions. However, in real time collaborations, each operation is processed as it arrives at the local replica so as to promote concurrency.

In replicated architecture real-time group editor research, the approaches adopted by the existing consistency management algorithms, including conflict resolution, can be categorised into: (1) locking approaches, (2) operational transformation approaches, and (3) multi-versioning approaches. Each of the following subsections outlines the existing work in each approach and explains their limitations and challenges.

4.3.1. Locking Approach

Locking is a pessimistic approach that prevents conflicts in distributed systems by prohibiting concurrent updates on shared data objects. The locking approach is a conflict prevention approach rather than a conflict resolution approach, and it does not promote concurrency as only one person can modify an object at one time. Most existing collaborative graphics editing systems have adopted a conflict prevention approach based on locking. Example systems based on locking include: Aspects [144], Ensemble [100], Group-Draw [57], and GroupGraphics [109]. In these systems, the user has to place a lock on an object before editing it, thus preventing other users from generating conflicting operations on the same object. Locking has also been applied to group text editors for consistency management [94, 97]. However, locking is undesirable for the following reasons. Firstly, it imposes overheads in the lock requesting, granting and releasing procedures, especially in replicated architectures where there is no machine dedicated to lock management. Secondly, it diminishes concurrency since users cannot modify the locked part of the document. Finally, the locking technique itself has not prevented divergence from occurring in a document where the objects are not independent [127].

More optimistic locking strategies such as optimistic shared-locking [133] and tentative optional locking [127] have also been proposed. Their non-blocking property allows users to continue updating the document while waiting for the lock. However, eventually they still need to wait for the lock to be resolved (if there are concurrent locks on the same region) to know whether their updates are to be kept or to be undone. Furthermore, they require additional operational transformation based rules for the locking operations to make sure the concurrent locking operations are applied consistently at all sites. Locking and its optimistic variants are discussed in great detail in section 3.3.1 and section 3.3.2.

4.3.2. Operational Transformation Approach

Operational transformation was first introduced by Ellis and Gibbs [46] in the dOPT algorithm to allow concurrent updates on document replicas. It possesses three consistency properties: convergence, causal preservation and intention preservation. It preserves causality by implementing vector clocks, and preserves user intention by transforming concurrent operations consistently at all sites thereby enforcing document convergence. Most operational transformation based algorithms [34, 126, 132, 142] serialise concurrent operations: concurrent operations are executed sequentially with the later operations being transformed to include the effect of the earlier operation according to a certain total order scheme. They use different strategies to serialise the concurrent operations: undo/do/redo [132], history separation [126], global sequencer [142] and distributed total ordering schemes [34]. The operation transformation technique together with its development over time is described in detail in section 3.3.3.

The operational transformation approach is optimistic and is able to handle non-exclusive conflicts by transforming concurrent operations. However, the operational transformation approach in itself cannot handle exclusive conflicts. The existing operational transformation based algorithms transform and serialise concurrent operations, including the conflicting ones. By serialising the concurrent operations, non-exclusive conflicts are preserved and the operations are applied to the document even though doing so may create semantic errors in the document state and realise neither user's intention. The users can then resolve the error by deleting one character or undoing an operation. Serialising exclusive conflicting operations, however, means that the operation executed later overrides the effect of the operation executed earlier thereby explicitly violating the intention of at least one user.

Recently, Ignat et al. [68] proposed a flexible conflict definition and resolution approach to be applied with the treeOPT algorithm. TreeOPT is an OT based algorithm that maintains consistency in a multi-level document structure. At each granularity level, the operations are transformed, serialised and executed to maintain consistency. With the multi-level document structure, the conflict can be detected in an appropriate document node hence it can handle exclusive conflicts. TreeOPT however uses a document repository similar to CVS where the shared document is checked-out, modified and committed, making it not suitable for mobile replicated architecture. During the commit phase, the conflict is detected and merged either automatically or with user intervention. As an effort to handle conflicts in mobile network, Ignat et al. proposed a P2P version of the algorithm [66]. The proposed strategy utilises the Tombstone Transformation Framework (TTF) algorithm to simplify its consistency maintenance algorithm and to allow implementation in peer to peer networks, hence also in mobile replicated architecture. Since TTF is beyond the scope of this paper and is left to future work, the conflict management proposed by Ignat et al. [66] is to be discussed and evaluated in future work.

4.3.3. Multi-versioning Approach

As an alternative to serialising conflicting operations using operational transformation, a multi-versioning approach is used to preserve the effect of both conflicting operations. The multi-versioning approach handles exclusive conflicts by creating multiple versions of the document and executing the operations in parallel, applying each operation to the different versions of the document [31, 130]. This approach is attractive as it preserves the intentions of all users and can be used in various conflict resolution strategies, such as voting or priority based authorization [149].

GRACE [130] was among the first to implement a multi-versioning scheme in a collaborative editing context. When two conflicting concurrent operations, *op1* and *op2*, are targeting an object *X*, the all-operation-effect is achieved by means of multiple versioning: two versions of *X* (*X1* and *X2*) are created, with *op1* and *op2* being applied to

X1 and *X2* respectively. The effects of both operations are accommodated in two separate versions. In the case of exclusive conflict, preserving all users' work is better than discarding any user's work. This system-level multi-versioning scheme provides better feedback to the users, helps the users to better understand the nature of their conflict, and to better adjust and coordinate their actions accordingly. The users can then choose by means of a conflict resolution strategy, either by voting or a group leader decision, which object version to be the final version.

Whenever a conflict occurs (and thus object versions are created) and is resolved, the unnecessary object versions are discarded and the object reverts to a single copy. However, if the conflict is left unresolved for a period of time, the object versions will remain until it is resolved. If any user generates an operation on that object during this time, then it has to be determined which object versions are affected by the operation. In GRACE, a Consistent Object IDentification scheme (COID) was implemented to make sure correct object versions are affected consistently. The longer the object in conflict is left unresolved and the more concurrent operations are generated on that object, the more complicated the object versioning will become since further versions may be created on a particular object versions creating some sub-versions and sub-sub-versions and so on. Not only does this increase processing requirements which is undesirable in a mobile environment, it also potentially consumes large memory space to store all object versions (and their subversions).

In an effort to restrict the number of versions created in a multi-versioning approach, Xue et al. [149] introduced the notion of conflict control locking to restrict the number of versions that can be created, to manage the created versions, and to facilitate the resolution of the conflicts. When an exclusive conflict occurs at a site, two versions of the targeted object will be created. As soon as the object versions are created, the system locks the object (and all its versions). After that, the user at that site is not allowed to generate any further operation targeting either of the object versions. This conflict control is also known as post-locking since the lock is applied after a conflict occurs. Due to communication delays, operations may arrive in different orders at different sites, hence newly arrived operations may be independent of both or one of the conflicting operations. If the new operation is conflicting with existing conflicting operations, a new version is created and is locked by the system. If it is not conflicting, it should be applied to all the current versions. Post-locking reduces the complexity caused by multiple version creation.

Post-locking has been used in systems such as POLO [148] and LOVOT [150]. POLO is applicable to independent-object documents, whereas LOVOT is applicable to dependent-object documents. Whenever a site receives a conflicting operation, once the object versions are created they are locked locally by the system until the conflict is resolved, with the intention that the user can modify other objects in the meantime. Note that the object versions are not necessarily post-locked at the same time at all sites since the post-lock is not propagated to other sites, but rather invoked on a case by case basis by the individual sites as they receive the conflicting operations. While post-locking simplifies the conflict resolution process by locking versions to protect them from further modification, it suffers from a partial intention problem. That is, the object being modified is automatically locked whenever a conflict occurs, and the user might not have fully realised his/her intention on the target object.

Consider the example in Figure 4-1 where *Alice* and *Bob* are collaborating on a graphic editor. As shown in Figure 4-1a, *Alice's* intention is realised by operations *op_1* and *op_2*: to move an object *X* to a certain position and to change the caption of *X*. *Bob*, on the other hand, generates *op_3* and *op_4* to move *X* to another position and change the size of *X*. Due to concurrency, *op_1* arrives to *Bob* before he can generate *op_4* and *op_3* arrives to *Alice* before she generates *op_2*. As shown in Figure 4-1b, post-locking will lock object *X* before *Alice* and *Bob* can generate the second operations to fully realise their intentions, hence, they will not know what is the full intention of the other party (moving an object may not necessarily be enough to show the complete intention of the user).

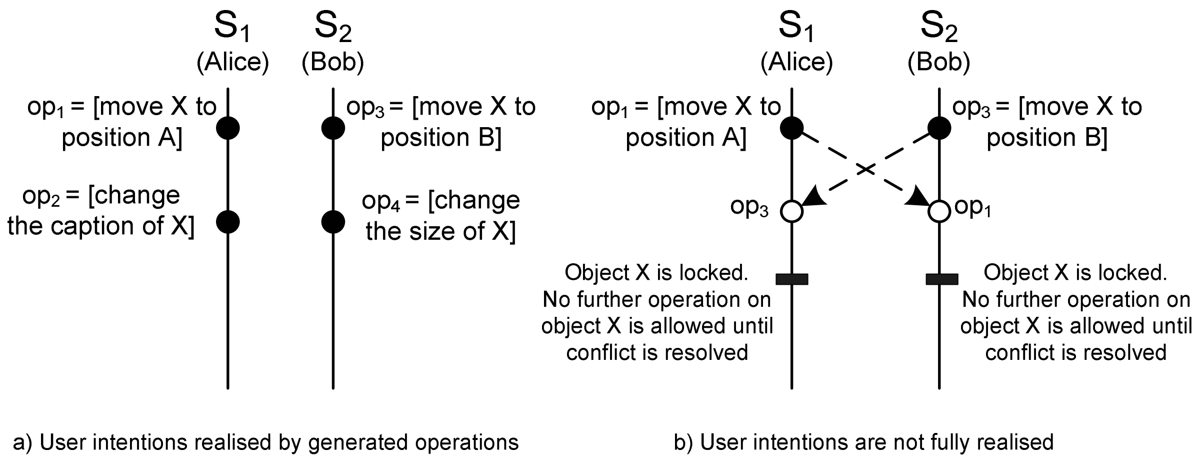

S1
(Alice)
S2
(Bob)
op1 = [move X to position A]
op2 = [change the caption of X]
op3 = [move X to position B]
op4 = [change the size of X]
a) User intentions realised by generated operations
S1
(Alice)
S2
(Bob)
op1 = [move X to position A]
op3 = [move X to position B]
op3
op1
Object X is locked. No further operation on object X is allowed until conflict is resolved
Object X is locked. No further operation on object X is allowed until conflict is resolved
b) User intentions are not fully realised

Figure 4-1 User intentions

Therefore, conflicts could be better resolved if the conflicting intentions have been completely realised and thus every user can see the full intention of all other users. Furthermore, post locking relies on a lock synchronization process. It requires the lock to be synchronised before conflicts can be resolved meaning that in mobile network environments, which are characterised by frequent disconnections, sites might have to wait indefinitely for the lock to be synchronized. The conflict resolution procedure can only be invoked if the locks are synchronized, meaning that all sites have received all conflicting operations and thus have access to the document versions created by those operations. Due to concurrency and network latency, the locks might not be synchronised at the same time at all sites and thus the conflict resolution procedure might never occur if one or more sites miss even a single conflicting operation.

To solve this problem, section 4.4.1 in this thesis presents a novel way to utilize the post-locking mechanism to restrict the number of object versions while at the same time allowing users to generate enough operations to fully realise their intentions on a particular object. Furthermore, with the use of a conflict table, a certain conflict resolution strategy can be employed such that the lock synchronization is not necessary to resolve the conflict.

4.3.4. Conflict Resolution Strategies

Unlike operation transformation which arbitrarily serialises conflicting operations, multi-versioning handles user intention conflicts by creating document or object versions. Consequently, users must eventually resolve the conflict by deciding which version is to be selected as the correct group-intended version. A simple way of resolving conflict is priority based authorization, which gives specific members, such as a group leader or administration group, the right to resolve conflicts. Although this approach reduces network usage due to messages round-trips, it introduces additional points of failure and may not be the most effective working strategy from a usability point of view [149]. Alternatively, conflicts can be resolved by negotiation [33] or reaching consensus amongst the group. For example, voting [149] aims to reach a group intended version where the decision is supported by at least the majority of users. Implementing such strategies in a distributed system is not easy, especially in a peer-to-peer mobile network environment [53]. In particular, such strategies require complex semantics, consume considerable bandwidth, and require good connectivity among devices. Furthermore, negotiation may continue indefinitely if a consensus or suitable outcome is never reached. Although the conflict management technique presented in section 4.4.2 is generic and can be used with various conflict resolution strategies, section 4.4.3 of this thesis presents one such conflict resolution strategy that requires only sites involved in the conflict to be involved in the conflict resolution (i.e. does not require all sites to resolve the conflict), and does not depend on a particular group leader.

4.4. Proposed Algorithm

This section proposes a conflict management algorithm that handles exclusive conflicts, while respecting user intention at the semantic level. The algorithm presented in this section detects and handles the conflict and keep the users informed of the conflict status. The algorithm is generic, meaning that it can be used with any kind of conflict resolution strategy. The next section (section 4.4.3), however, presents one possible conflict resolution strategy that is suitable for mobile replicated architecture. The detail of the algorithm is as follows.

Firstly, a multi-versioning approach, in which each user's intention is realised in a different version of the document object, is implemented. Note that the application may, however, choose to use a multi-versioning approach to handle some non-exclusive conflicts as shown in the following example. Suppose *Alice*, *Bob* and *Cameron* are collaborating and they are concurrently modifying the word "and". *Alice* is trying to create the word "strand", *Bob* is making the word "grand", and *Cameron* is changing the word into "errand". Using operational transformation approach, these operations can be transformed and executed at all sites without having to create object versions. However, depending on the current site states and operation timestamps, the new word would contain all the changes and become something like "sgetrrand" or "stgerrand". The conflict is technically non-exclusive as the conflicting operations can be realised together consistently in the same object using operation transformation approach. However, *Alice* would not be able to determine what *Bob* and *Cameron* are trying to do. *Alice* would not have any idea if *Bob* is trying to make a word "grand". Furthermore, she would not know whether *Bob* is typing his letters concurrently with or after hers. Therefore, in this scenario, the conflict is better treated as an exclusive one and user intentions are better preserved using the multi-versioning approach.

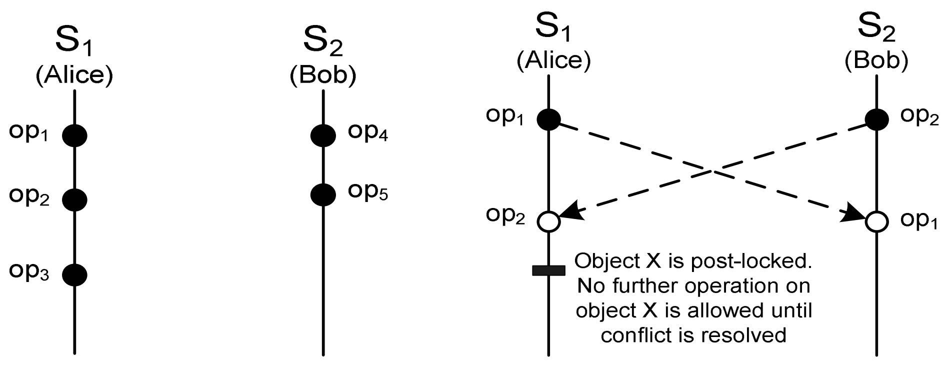
Secondly, the proposed algorithm implements a variation of post-locking called *delayed post-locking*, which addresses the partial-intention problem described in section 4.3.3. In general, whenever object versions are created, whether it is due to an exclusive or non-exclusive conflict, the object in conflict must be locked to avoid further update and to trigger conflict resolution. A post-lock can be placed either at the object level or at the object's attribute level depending on the application. If an application uses the attribute of an object as the base of the conflict (target), it may choose to lock the attribute of the object rather than the whole object, leaving the other attributes editable by the users. The finer the granularity of the lock, the higher the level of the concurrency supported, however the less likely a conflict will be noticed, thereby increasing the potential difficulty of resolving it. The specifics of delayed post locking, which are unique to the newly proposed algorithm, are described in section 4.4.1.

Finally, the novel use of a conflict table to store all conflict information for the purpose of facilitating conflict resolution is also described in section 4.4.2 as part of the description of the general scheme for managing the storage and resolution of conflicts in order to support different conflict resolution strategies, with section 4.4.3 presenting one such strategy that is suitable for mobile environments.

4.4.1. User Intention Completion

A conflict can only be resolved properly when the intentions of all users involved in the conflict are completely realised (i.e. the users have finished generating all necessary operations to the object and the other users have received all of them). A practical example scenario of an incomplete intention (partial intention) has been mentioned in section 4.3.3. This section presents the problem theoretically and proposes a solution to the problem of partial intention.

Suppose *Alice* at site S_1 and *Bob* at site S_2, concurrently update an object X (Figure 4-2a). To realise her intention, *Alice* needs to execute (generate) three operations on X: op_1, op_2 and op_3 sequentially. Concurrently, *Bob* generates operations op_4 and op_5 to realise his own intention on X. Using the conventional post-locking approach [149], when op_4, which conflicts with op_1, arrives at S_1 before *Alice* generates op_2, X is automatically locked, therefore *Alice* cannot fully realise her intention (Figure 4-2b). Consequently, *Bob* will not have the complete picture of what *Alice* intends to do. The conflict could be better resolved semantically if *Alice's* intention is fully realised (i.e. *Bob* fully knows what *Alice* wants to do). This also happens if op_1 arrives at S_2 before op_5 is generated and op_1 is conflicting with op_5, causing *Alice* not to fully know what *Bob* intends to do.



a) Concurrent Updates b) Partial User Intention

Figure 4-2 Conflicting operations

Therefore, a strategy called *delayed post-locking* is proposed as the solution to this problem. This strategy allows users to completely realise their intention without being interrupted by incoming conflicting operations. The delayed post-locking technique uses a lock called a *user intention lock* (UI-Lock) to prevent any interruption from incoming operations, thereby allowing the user to fully realise his/her intention. The rest of this subsection proposes two alternatives for implementing user intention locks.

Manual UI-Lock

Each user is able to apply a UI-Lock to his/her document when s/he wants to generate more than one operation to realise an intention. When a UI-Lock is placed on a certain object, all incoming remote operations that target the UI-Locked object are held in the remote operation queue even though they are causally ready. When the UI-Lock is released, the operations waiting in the remote queue can then be executed accordingly.

Using the above example, *Alice* places a UI-Lock on X and generates *op_1*, *op_2* and *op_3*. Although *op_4* arrives before *op2* is generated, it cannot be executed because X is UI-Locked (Figure 4-3). This gives *Alice* time to generate necessary operations, *op_2* and *op_3*, before being interrupted by incoming remote operations (and potentially post-locked due to conflict). The operations generated during the application of the UI-Lock are technically concurrent with the incoming remote operations even though *op_4* arrives at *S_1* before *op_2* is generated. When *op_1* arrives at *S_2*, *Bob* knows that there are more than one operation concurrent and conflicting with *op_4*, therefore the conflict can be resolved after all operations have been received and it can be better resolved since *Bob* will know the full intention of *Alice*. A UI-Lock is a local and temporary lock, that when *Alice* places a UI-Lock on object *X*, *Bob* does not know anything about it and *Bob* only knows that *op_1* and *op_2* are concurrent with *op_4*.

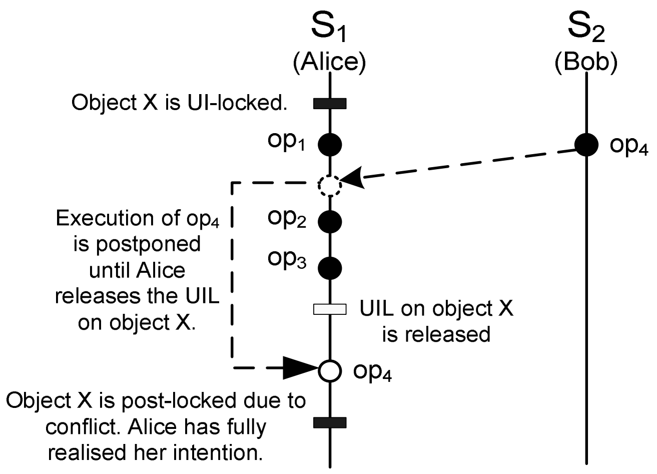

S1
(Alice)
S2
(Bob)
Object X is UI-locked.
op1
op4
Execution of op4 is postponed until Alice releases the UIL on object X.
op2
op3
UIL on object X is released
op4
Object X is post-locked due to conflict. Alice has fully realised her intention.

Figure 4-3 Manual user intention lock

Automatic UI-Lock

The manual UI-Lock described above requires manual intervention from the user. In practice, the application of the UI-Lock can be automated. One option is to automatically place the UI-Lock on an object whenever there is an incoming conflicting operation that targets the same object. Using the previous example, object *X* is UI-Locked automatically when *op4* arrives and *op4* is held in the remote queue (Figure 4-4). *Alice* is then notified that there is a conflicting incoming operation in the remote queue and *Alice* is given chance to complete any necessary operations on object *X* before the object is post-locked due to conflict. After generating op_2 and op_3, *Alice* can then release the UI-Lock to allow the remote operations to be executed as usual. A timeout period can also be applied to the UI-Lock that when the timeout has elapsed and the user does not do anything to the object, the UI-Lock is automatically released so as not to hold up the remote operations in the queue. Without losing generalities, this scheme of automatic UI-Lock is used for the rest of the thesis. Another possible option is to automatically place a UI-Lock on the object currently being modified. Any conflicting remote operation targeting the object is held up in the queue. Once the user modifies another object, the UI-Lock on the object is released and the remote operation can be released from the queue.

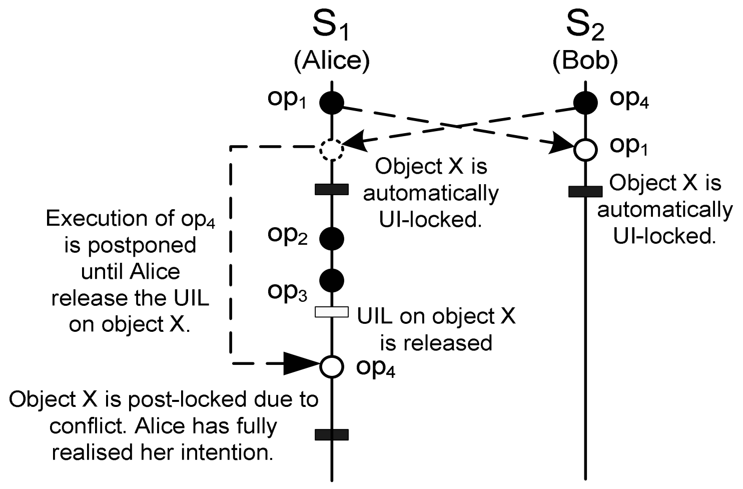

S1
(Alice)
S2
(Bob)
op1
op4
op1
Object X is automatically UI-locked.
Object X is automatically UI-locked.
Execution of op4 is postponed until Alice release the UIL on object X.
op2
op3
UIL on object X is released
op4
Object X is post-locked due to conflict. Alice has fully realised her intention.

Figure 4-4 Automatic user intention lock

Nevertheless, regardless of when the UI-Lock is placed and released, this strategy ensures that when a conflict on an object arises, the user has already generated all necessary operations on that object. This strategy can also be extended to allow a user to inspect the incoming operations before generating additional operations. When *X* is locally-locked automatically and *op_4* is in the remote queue, *Alice* can choose to have *op_4* executed to see what *op_4* is trying to do. If *Alice* is happy with *op_4* and does not want to generate further operations, *Alice* can undo *op_1* and release the UI-Lock, and *op_4* is taken out from the remote queue. Otherwise, *Alice* can undo *op_4* and store *op_4* back to the remote queue and then generates *op_2* and *op_3* before the local lock is released. This gives flexibility on the user to either realise his/her intention or pre-empt his/her intention knowing that the intention of the other user is more desirable.

4.4.2. Conflict Management

This section describes the general data structures and techniques required to manage the storage and resolution of conflicts, in order to support different conflict *resolution* strategies, as described in the following section.

Participants

Each site maintains a list of participants (PL_S = participant list of site *S*) that stores the ids of the other sites as well as other information, such as site name and site connectivity information (IP address and port number).

State Vector

Besides its own state vector, each site maintains a state vector table (*VT*) that consists of *N* state vectors, one per site. Each site also maintains a minimum state vector (*MSV*) to reflect the number of operations generated by each site that have been executed at every site. State vector, state vector table and minimum state vector have been used to support consistency management (refer to section 3.5.1), and they can also be used in the conflict management algorithm presented in this section. Hence they do not impose an additional storage overhead.

Conflict Table

Whenever a conflict occurs, the object in conflict is locked, the user is notified (e.g. by highlighting the object), and only the operations causing the conflict are stored in a conflict table. This reduces memory consumption compared with existing solutions because the object versions can either be inferred by looking at the operations involved, or explicitly by executing the operations on a temporary document copy which can be discarded following the inspection.

Each site maintains a Conflict Table (*CT*) to store the conflict information. Whenever a remote operation arrives at a site, the site checks whether the remote operation is conflicting with any concurrent operation that has been executed. If it does (a conflict occurs), each conflicting operation is stored in the conflict table to signify the different possible object states realised by each operation involved in the conflict. Due to concurrent operation generation, a user of a particular site may generate more than one operation on a particular object to realise his/her full intention. Hence, operations from the same site are grouped together to represent one object version.

Each entry in CT_k (Conflict Table of site S_k) is a tuple representing the conflicting object version: $CT_k[i]$ = <*target, siteId, opIds, status, res*>. $CT_k[i][target]$ is the target object on which the conflict occurs. Two operations are conflicting if they have the same target but modify it differently. Depending on the definition of the conflict (application specific), $CT_k[i][target]$ can be the object id or the combination of the object id and object attribute. For example, $CT_k[i][target]$ = <*X, FontSize*> if two or more operations are modifying the font size of object *X* to different values. Based on the document model mentioned in section 3.2.1 and the previously stated assumption, this thesis defines:

1. $CT_k[i][target]$ = <*objId, attr*> if the operations are modifying the same object's attribute, and
2. $CT_k[i][target]$ = <*objId*> if the operations are inserting characters at the same position of the same word.

$CT_k[i][objId]$ is the id of the target object and $CT_k[i][attr]$ is the attribute of objId being modified. $CT_k[i][siteId]$, $CT_k[i][opIds]$, $CT_k[i][status]$, and $CT_k[i][res]$ are the site id, operation ids, status and the resolution of $CT_k[i]$ respectively. The following definitions are used to help in explaining the proposed algorithm.

Definition 4-1 *Conflict Relation*

$CT_k[i]$ and $CT_k[\mathrm{j}]$ are *conflicting*, $CT_k[i] \otimes_{ct} CT_k[\mathrm{j}]$ iff $CT_k[i][target] = CT_k[\mathrm{j}][target]$.

Definition 4-2 *Site's involvement in a conflict*

Site S_k is involved in $CT_k[i]$, $S_k \in_{ct}^{S} CT_k[i]$ iff there exists $CT_k[j]$ such that $CT_k[j] \otimes_{ct} CT_k[i]$ and $CT_k[j][siteId] = S_k$.

Definition 4-3 *Operation's involvement in a conflict*

Operation op_x is involved in $CT_k[i]$, $op_x \in_{ct}^{op} CT_k[i]$ iff there exists $CT_k[j]$ such that $CT_k[j] \otimes_{ct} CT_k[i]$ and $op_x \in CT_k[j][opIds]$.

$CT_k[i][status]$ signifies whether or not the intention on the particular object version has been completely realised. This information allows users to make better conflict resolution decisions in terms of which document version should be accepted. $CT_k[i][status]$ can be one of the following values:

1. *Partial*. $CT_k[i][status] = partial$ if it is in neither complete nor resolvable. This signifies that this entry has been recently created, and the user of site $CT_k[i][siteId]$ might not have generated all necessary operations to fully realise his/her intention (partial intention problem).
2. *Complete*. $CT_k[i][status] = complete$ if site $CT_k[i][siteId]$ has finished generating conflicting operations. In other words, site $CT_k[i][siteId]$ has already executed at least one of the conflicting operations ($\exists op_x \in \{CT_k[j][opIds], \forall CT_k[j] \otimes_{ct} CT_k[i]$ and $CT_k[j][siteId] \neq CT_k[i][siteId]\}$). Suppose $S_j = CT_k[i][siteId]$, as soon as S_j executes one of the conflicting operations, the target object in S_j is locked, therefore S_j cannot generate further operations on the object, thus S_j has finished generating conflicting operations.
3. *Resolvable*. $CT_k[i][status] = resolvable$ iff site S_k has the right to resolve the conflict (i.e. potentially accept $CT_k[i]$). Although a conflict can be resolved by various conflict resolution strategies (as mentioned in section 4.3.4), a conflict is eventually resolved by one mobile site S_k selecting one of the Conflict Table entries $CT_k[i]$ to be the final version for that particular object. This decision is then broadcast to all other sites. Note that the criteria to determine whether any particular site should be

allowed to choose a solution (i.e. $CT_k[i][status]$ = *resolvable*) is referred to in this thesis as a *conflict resolution strategy*, with one such strategy being presented in section 4.4.3.

$CT_k[i][res]$ denotes whether $CT_k[i]$ is accepted as the final version. $CT_k[i][res]$ = *accept* if $CT_k[i]$ is accepted as the final version, *reject* if it is rejected or *none* if the conflict has not been resolved.

From the above example (Figure 4-3 or Figure 4-4), when op_4 is executed at site S_1, two entries in CT_1 are created:

1. $CT_1[0]$ = <*X*, S_1, [op_1, op_2, op_3], *complete*, *none*>, and
2. $CT_1[1]$ = <*X*, S_2, [op_4], *partial*, *none*>.

When a conflicting remote operation arrives from a site, and there is already a *CT* entry of that site, the operation is simply appended to the *opIds* for that *CT* entry. For example, when op_5 arrives and is executed in site S_1, op_5 is simply added to $CT_1[1]$ to become <*X*, S_2, [op_4, op_5], *partial*, *none*>. Both entries are conflict-related since they are involved in the same conflict (i.e. they have the same target). Sites S_1 and S_2 are involved in $CT_1[0]$ and $CT_1[1]$ because they generate the conflicting operations. Operations op_1, op_2, op_3, op_4 and op_5 are *involved* in $CT_1[0]$ and $CT_1[1]$ as they are the conflicting operations.

Figure 4-5 illustrates the creation and modifications of *CT* entries when the sites receive each of the conflicting operations. Using delayed post-locking discussed in section 4.4.1, the first conflicting remote operations, op_4 and op_1, will be executed only after the local site generates all necessary operations (i.e. op_1 is processed at S_2 after op_5 is generated and op_4 is processed at S_1 after op_3). When op_1 arrives at S_2, S_2 realises that op_1 is conflicting with op_4 and op_5; therefore two conflict entries are created in the conflict table (one entry for each site). Notice that when the conflict table entry is just created, the status of the conflict table entry associated with S_1 is still partial since S_2 is not sure whether op_1 is the only conflicting operation from S_1. When op_2 and op_3 arrive at S_2, there is no need to create a new conflict table entry since a conflict entry that targets object X and associated with S_1 has already been created; or in other words, there exists a version of object X initiated by S_1. Operation op_2 and op_3 are simply appended to the existing conflict table entry. Notice that until this point, the status is still partial since S_2 cannot determine whether op_3 is the last conflicting operation from S_1. However, when op_7 arrives at S_2, S_2 realises that S_1 has finished generating the operation on object X based on the state vector of op_7, therefore the status of the respective conflict table entry CT_2 is changed to *complete*.

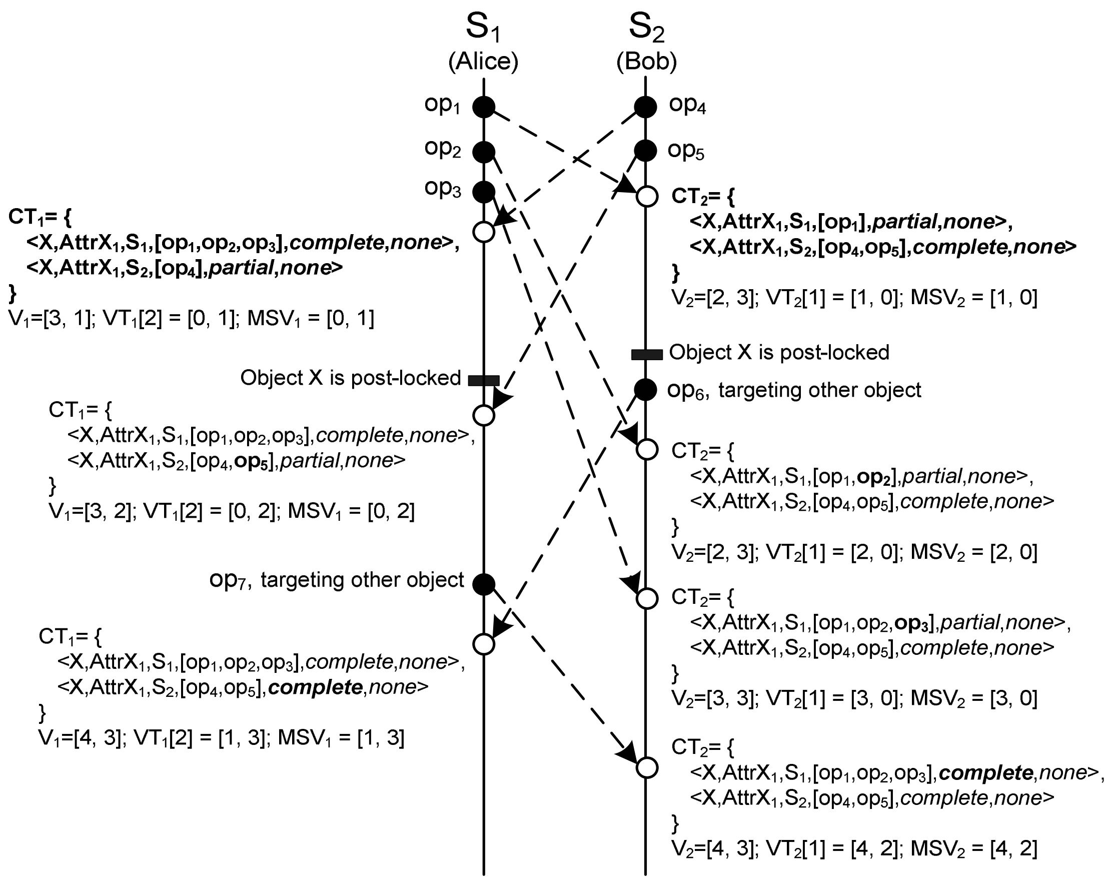
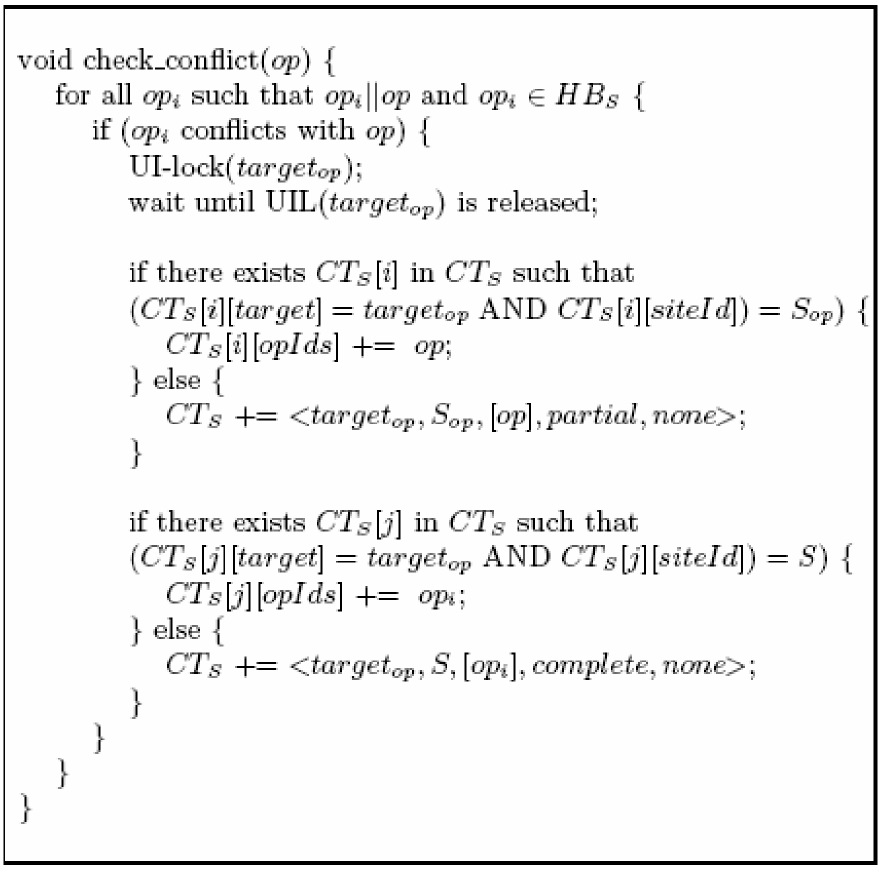

S1 (Alice)
S2 (Bob)
op1
op2
op3
op4
op5
CT1= { <X,AttrX1,S1,[op1,op2,op3],complete,none>, <X,AttrX1,S2,[op4],partial,none> }
V1=[3, 1]; VT1[2] = [0, 1]; MSV1 = [0, 1]
CT2= { <X,AttrX1,S1,[op1],partial,none>, <X,AttrX1,S2,[op4,op5],complete,none> }
V2=[2, 3]; VT2[1] = [1, 0]; MSV2 = [1, 0]
Object X is post-locked
Object X is post-locked
op6, targeting other object
CT1= { <X,AttrX1,S1,[op1,op2,op3],complete,none>, <X,AttrX1,S2,[op4,op5],partial,none> }
V1=[3, 2]; VT1[2] = [0, 2]; MSV1 = [0, 2]
CT2= { <X,AttrX1,S1,[op1,op2],partial,none>, <X,AttrX1,S2,[op4,op5],complete,none> }
V2=[2, 3]; VT2[1] = [2, 0]; MSV2 = [2, 0]
op7, targeting other object
CT2= { <X,AttrX1,S1,[op1,op2,op3],partial,none>, <X,AttrX1,S2,[op4,op5],complete,none> }
V2=[3, 3]; VT2[1] = [3, 0]; MSV2 = [3, 0]
CT1= { <X,AttrX1,S1,[op1,op2,op3],complete,none>, <X,AttrX1,S2,[op4,op5],complete,none> }
V1=[4, 3]; VT1[2] = [1, 3]; MSV1 = [1, 3]
CT2= { <X,AttrX1,S1,[op1,op2,op3],complete,none>, <X,AttrX1,S2,[op4,op5],complete,none> }
V2=[4, 3]; VT2[1] = [4, 2]; MSV2 = [4, 2]

Figure 4-5 Handling a conflict using a conflict table

Figure 4-6 outlines the *check_conflict(op)* procedure that (1) checks whether or not an incoming operation *op* causes, or is involved in, a conflict in site *S*, and (2) adds the *op* into a *CT* entry accordingly. Note that without losing generality, automatic UI-Lock is being used as an example in this procedure.



```
void check_conflict($op$) {
    for all $op_i$ such that $op_i||op$ and $op_i \in HB_S$ {
        if ($op_i$ conflicts with $op$) {
            UI-lock($target_{op}$);
            wait until UIL($target_{op}$) is released;

            if there exists $CT_S[i]$ in $CT_S$ such that
            ($CT_S[i][target] = target_{op}$ AND $CT_S[i][siteId]) = S_{op}$) {
                $CT_S[i][opIds]$ += $op$;
            } else {
                $CT_S$ += <$target_{op}, S_{op}, [op], partial, none$>;
            }

            if there exists $CT_S[j]$ in $CT_S$ such that
            ($CT_S[j][target] = target_{op}$ AND $CT_S[j][siteId]) = S$) {
                $CT_S[j][opIds]$ += $op_i$;
            } else {
                $CT_S$ += <$target_{op}, S, [op_i], complete, none$>;
            }
        }
    }
}
```

Figure 4-6 Conflict checking procedure

As previously stated, a conflict is eventually resolved by selecting one Conflict Table entry to be the final version for that particular conflict. If $CT_k[i]$ is selected to be the desired version, a Conflict Resolution Operation (*CRO*), $op_r = accept(op_x)$, is generated where op_x is one of the operations in $CT_k[i][opIds]$. If $CT_k[i]$ is accepted, then $CT_k[i][res] = accept$, and the *CT* entries that are conflict-related to $CT_k[i]$ are rejected ($CT_k[j][res] = reject, \forall CT_k[j] \otimes_{ct} CT_k[i]$). The *CRO* is then broadcast to all other sites and when it arrives at a site, that site will accept the Conflict Table entry which *op*$_x$ belongs to, and reject the other conflict-related Conflict Table entries.

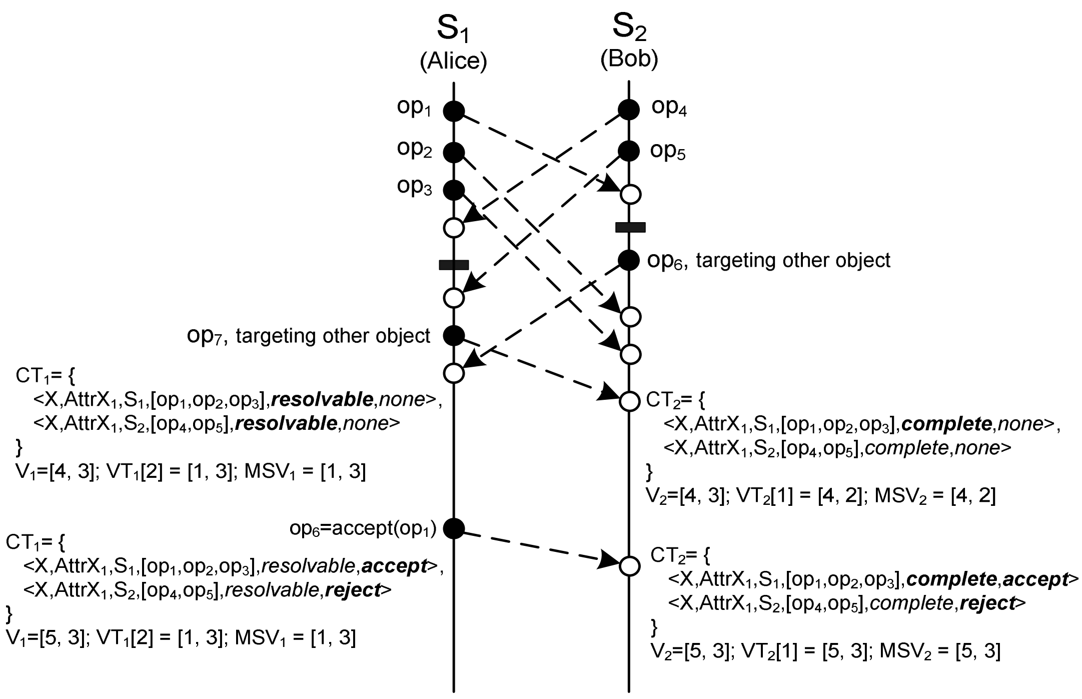

S1
(Alice)
S2
(Bob)
op1
op2
op3
op4
op5
op6, targeting other object
op7, targeting other object
CT1= {
<X,AttrX1,S1,[op1,op2,op3],resolvable,none>,
<X,AttrX1,S2,[op4,op5],resolvable,none>
}
V1=[4, 3]; VT1[2] = [1, 3]; MSV1 = [1, 3]
CT2= {
<X,AttrX1,S1,[op1,op2,op3],complete,none>,
<X,AttrX1,S2,[op4,op5],complete,none>
}
V2=[4, 3]; VT2[1] = [4, 2]; MSV2 = [4, 2]
op6=accept(op1)
CT1= {
<X,AttrX1,S1,[op1,op2,op3],resolvable,accept>,
<X,AttrX1,S2,[op4,op5],resolvable,reject>
}
V1=[5, 3]; VT1[2] = [1, 3]; MSV1 = [1, 3]
CT2= {
<X,AttrX1,S1,[op1,op2,op3],complete,accept>,
<X,AttrX1,S2,[op4,op5],complete,reject>
}
V2=[5, 3]; VT2[1] = [5, 3]; MSV2 = [5, 3]

Figure 4-7 Resolving conflict

Figure 4-7 illustrates an example of a conflict resolution process, with *op*$_6$ being the *op*$_r$, and the changes made to the conflict table entries. Note that while the proposed conflict management strategy can be used with various conflict resolution strategies, Figure 4-7 illustrates one example of a possible conflict resolution strategy as discussed in section 4.4.3. The *CRO op*$_r$ is then appended to *CT*$_k$[*i*][*opIds*] for garbage-collection purposes. Since the conflicting operations are non-transformable, all operations that are in conflict to $CT_k[i][opIds]$ are simply undone (if they have been executed) and operations $CT_k[i][opIds]$ are executed and stored in the history replacing the undone operations.

Conflict Table Garbage Collector

An entry in $CT_k[i]$ needs to be kept in CT_k until site S_k is confident that the conflict has been resolved and the entry is not needed anymore to process future operations. If $CT_k[i]$ is rejected, then site S_k can remove $CT_k[i]$ if S_k is confident that there will not be any future conflicting operation coming from site $CT_k[i][siteId]$ (i.e. $CT_k[i][status]$ is *complete* or *resolvable*). If $CT_k[i]$ is accepted, however, $CT_k[i]$ can only be removed if all sites have executed all operation in $CT_k[i][opIds]$ including the *CRO* (*CRO* is appended to $CT_k[i][opIds]$ once it is accepted), which also means that all sites have received results of the conflict resolution. In summary, $CT_k[i]$ can be removed from CT_k iff:

1. (CT_k[i][res] = reject) AND ((CT_k[i][status] = complete) OR (CT_k[i][status] = resolvable)), OR
2. ($CT_k[i][res] = accept$) AND ($MSV[S_{op_x}] > V_{op_x}[S_{op_x}], \forall op_x \in CT_k[i][opIds]$).

4.4.3. Conflict Resolution

Having described in the previous section a general conflict management mechanism, this section presents an example of a specific conflict resolution strategy. Compared to a voting strategy (consensus reaching strategy), the conflict resolution strategy presented in this section uses less resources as it requires less message roundtrips and does not require a complicated consensus reaching algorithm. As such, it is suitable for mobile network environments. In addition, conflict is resolved without waiting for all sites to receive the conflicting operations thereby increasing user responsiveness. However, despite these advantages the algorithm does not facilitate negotiation or the reaching of consensus, as such this is left to future work.

Under this strategy, each site has a conflict resolution priority level (PR_{S_k} = priority level of site S_k), which is ordinal relative to other sites. The sites' priorities can be static (e.g. based on *siteId*) or dynamic (e.g. based on which site generated the conflicting operation the earliest). To avoid relying on a single site for conflict resolution (usually the site with the highest priority), only the priority level of the sites actually involved in a given conflict are considered. In other words, the user at site S_k has the right to resolve conflict if site S_k is involved in the conflict and site S_k has the highest priority among all sites involved in the conflict. If there is another site that has a higher priority than S_k, say S_r, site S_k still has the right to resolve the conflict if S_k is sure that S_r is not involved in the conflict.

Therefore, more formally, site S_k has the right to resolve conflict (generate op_r = *accept*(op_x), where $op_x \in CT_k[i][opIds]$)iff:

1. S_k is involved in the conflict ($S_k \overset{S}{\in}_{ct} CT_k[i]$) and $PR_{S_k} \geq PR_{S_x}$, AND
2. PR_{S_k} is the highest among all sites involved in the conflict ($S_x \in \{CT_k[j][siteId]$, $\forall CT_k[j] \otimes_{ct} CT_k[i]\}$), AND

3. Sites with higher priority (if any) are not involved in the conflict. In other words, sites with higher priority have executed at least one of the operations involved in the conflict. For all $S_x \in PL_k$ and $PR_{S_k} \geq PR_{S_x}$, there exists op_j such that $VT_k[x][S_{op_j}] > V_{op_j}[S_{op_j}]$ and $op_j \in^{op}_{ct} CT_k[i]$.

When $CT_k[i][status]$ = *complete* and the user at site S_k has the right to resolve the conflict, the status of CT_k becomes $CT_k[i][status]$ = *resolvable*. As described in the previous section, once $CT_k[i]$ has been accepted, a *CRO* is generated and broadcast to all other sites.

The conflict resolution strategy described above has the following advantages. Firstly, when a site has the right to resolve conflict, it can resolve the conflict anytime without having to wait until all sites receive all conflicting operations. Secondly, sites that are not involved in the conflict do not need to resolve conflict; therefore it does not always have to be a pre-determined or delegated group member who resolves conflicts. Thirdly, the rightful site can resolve conflict anytime without having to wait for the lock to be synchronized. A site whose priority is not the highest still has to make sure that the higher-priority sites are not involved in the conflict. This can be determined by the state vector of the operations generated by those sites (note that each operation carries the state vector of the originator site at the time it is generated). Therefore, it has to wait for operations to be generated by higher-priority sites before it can conclude that it has the right to resolve conflict. This is, however, still preferable to having to wait for all sites to receive all conflicting operations, and this process can be expedited by sending a state vector request to higher-priority sites so they can send their state vector update without having to generate operations.

4.5. Conclusion

In contrast to a centralised architecture where the server carries out the conflict management, in a replicated architecture each site has to carry out the conflict management consistently. Consequently, conflicts have to be properly detected and resolved consistently across all sites.

The algorithm presented in this chapter has built upon the multi-versioning approach to more effectively handle conflicts in replicated object based collaboration applications. In particular the concepts of delayed post-locking and conflict tables have been introduced to address shortcomings in existing approaches.

Delayed post-locking uses *user intention locks* (UI-Lock) to locally lock the object being edited so that while the UI-Lock is in place, all incoming operations that target that object will be held in the queue until the UI-Lock is released. This gives time for the user to generate necessary operations to fully realise his/her intention on the object so that more complete information is available to other users to assist in the conflict management process.

Furthermore, each site maintains a *conflict table*, which facilitates the user in dealing with conflict by letting him or her know whether other users have fully realised their intention on the object in conflict. The conflict table allows conflicts to be better organised, supports a simpler and more flexible conflict resolution process, and keeps users better informed of the status of the conflict. It also informs the user when s/he has the right to resolve the conflict, insofar as being allowed to select the final version of the object.

Most importantly, the proposed algorithm satisfies the following conditions: it does not suffer from a partial-intention problem; it does not need to depend on a group leader or other pre-specified conflict resolution roles; the conflict can potentially be resolved without having to wait for all sites to receive all conflicting operations (dependent upon the chosen conflict resolution strategy); and finally, the algorithm provides better information to users so that they can resolve the conflict knowing the status of the conflict. Combined with the consistency management algorithm presented in Chapter 3, non-exclusive and exclusive conflicts can be handled effectively.

Future work will also look at alternative conflict resolution strategies, such as voting and/or group leader's decision with particular emphasis on their effectiveness from a usability point of view, and their performance and impact on resource consumption within a mobile environment. As mentioned in section 4.3.2, future work will also evaluate the conflict definition and resolution in multi-level document proposed in [66] and extend the algorithm proposed in this thesis to handle conflicts in hierarchical documents.

Chapter 5

Membership Management

5.1. Introduction

In Chapter 3, a consistency management algorithm has been proposed to ensure consistency of document replicas across all mobile collaboration participants. The devised consistency management algorithm takes advantage of the operation transformation technique that allows each operation to be broadcast independently without the need for a centralised server or any dedicated machine to do the centralised consistency management. In Chapter 4, a conflict management algorithm was proposed to handle exclusive conflicts and to facilitate users in resolving conflicts.

However, the devised algorithms, like most existing algorithms that work in wired networks, still assume the following.

- The number of participants is fixed.
- Participants join the collaboration in the beginning of the collaboration session.

- Participants leave the collaboration voluntarily only when they want to quit the session.

Those membership behaviours however do not hold in wireless network environments, especially in mobile ad-hoc network environments. A site (participant) can join the session any time during the session whenever it is within the transmission range of a current session participant and it decides to join the session. Sites frequently get disconnected from other sites because they rely on the radio frequency based wireless transmission to connect to other sites. This causes fluctuation in the number of currently active participants. A site can leave the session voluntarily (with adequate notification) and involuntarily (sudden disconnection) and it is impossible to distinguish a crashed process from one that is just very slow [53] since there is no way to determine how long that site will be disconnected.

In a centralised architecture, whenever a site wants to join a collaboration session, it simply contacts the server and sends a joining request to get the latest state of the document, and can then start collaborating. There is nothing that the other sites need to do to accommodate the new site. The server can easily include the new site in the session by accepting updates from the new site and sending the document state to the new site whenever the document state is changed. If a site decides to leave the session, it simply notifies the server and leaves the collaboration. Depending on the applied algorithm and collaboration workflow, the other sites may not need to know about this and they can continue collaborating. In contrast, in a fully replicated architecture, there is no server that can easily manage the membership. Each client device has to adjust its behaviour every time there is a membership change since the participants need to know all sites that they must inform of document updates. Whenever a site joins the session, all other participants need to know about the new site so that all future updates will also be sent to the new site and the new site can be considered during the process of each remote operation (i.e. to determine the precedence and concurrency of operations). Furthermore, mobile networks are characterised by fluctuating bandwidth and frequent disconnections, causing messages to be lost or corrupted during transfer. The application has to ensure all messages arrive at all intended destinations despite these characteristics.

This chapter aims to explore the requirements of a membership management algorithm and to devise an algorithm that is capable of handling the various membership events in a mobile real-time collaboration environment while still maintaining document consistency across all participants.

The remainder of this chapter is structured as follows. Section 5.2 discusses the membership problems in wireless networks in detail. Section 5.3 presents existing work that has been done and why it is not suitable for mobile real-time collaboration. Section 5.4 outlines the algorithm proposed to handle all dynamic membership issues while maintaining document consistency. Section 5.5 discusses some possible variations in the implementation of the proposed algorithm. Section 5.6 reports the result of evaluating the performance of the proposed algorithm. Finally, Section 5.7 concludes this chapter and discusses some future work.

5.2. Membership Problems

This section discusses the various events that affect group membership, including the cause of each event, how each event should be handled, and how the impact of each event adds to the requirement of the proposed algorithm.

5.2.1. Disconnection

In a mobile network environment, a mobile device is connected to another device if it is within the wireless transmission range of the other device, or if there is any intermediate device that is willing to route the packet from one device to another. Disconnection in wireless networks can be caused by the following reasons.

- *User mobility*. Mobile users are naturally mobile and if the mobile user moves out of the transmission range of the other device, it is disconnected from that device. Due to user mobility, a mobile user can move between connected and disconnected states quite rapidly.
- *Available bandwidth*. Disconnection can also be caused by fluctuating bandwidth. Wireless transmission relies on the strength of the radio frequency. Any noise in the radio frequency can affect the signal strength, thus affecting the bandwidth available for the wireless transmission. This causes the bandwidth to fluctuate over time and if the noise is severe, the signal could fade away and cause disconnection.

- *Device failure*. If a device crashes, then it cannot receive and send messages until it is restored, thus it is disconnected.

Unfortunately, whenever a device is disconnected from the other device, it is difficult, if not impossible, to determine the cause of the disconnection. There is no way to distinguish whether the disconnection is caused by a device being outside the transmission range, unavailable bandwidth or the failure of the other device [53].

When a site S_0 is disconnected, it stops receiving messages from the devices it is disconnected from. Site S_0 is not able to broadcast messages to the other devices either. In a collaboration session, this would mean that the disconnected site S_0 will miss some operations that are being broadcast while it is disconnected. This case is called a "missing operation". Not only that site S_0 misses the operation while it is disconnected, the other devices also miss some operations that site S_0 generates during the disconnection period. Therefore, the algorithm must ensure that an operation generated by one site will eventually arrive at all participants no matter the condition of the wireless network.

The requirements are formally outlined as follows. Requirement 5-1 ensures that all operations are delivered to all participants while Requirement 5-2 implies that if an operation is generated while a device is disconnected, the device will have to be able to get the missing operation after it is reconnected.

Requirement 5-1. ***Delivery guarantee***

Let op_i be an operation generated by site S_i, all other sites S_j ($\forall j$: $1 \leq j \leq N$) will eventually receive op_i.

Requirement 5-2. *Missing Operation*

Let op_i be an operation generated by site S_i while S_i is disconnected from S_j, S_j will eventually receive op_i after S_i is reconnected.

5.2.2. Late Join

In wireless network environments, especially in ad-hoc environments, two or more sites can start a session as soon as they are connected to each other (they are within the transmission range of each other). They do not need an established infrastructure to start communicating. This provides a greater flexibility in wireless communication. However, this also means that a device can arbitrarily join a currently running session whenever it is within the transmission range of any of the current participants. This is a case of a "*late join*".

Although there are many ways for a site to discover a currently running collaboration session, the discovery protocol is beyond the scope of this thesis; therefore it is assumed that a new site S_i decides to join the session after contacting a current collaboration participant S_j.

A correct membership management algorithm has to ensure that S_i is brought up to speed with the other participants such that it is able to correctly participate in the collaboration session (Requirement 5-3). A correct membership management algorithm also needs to ensure that all other participants know that there is a new site joining and therefore make necessary adjustment so that the collaboration session can accommodate the new site correctly (Requirement 5-4). In other words, when a new site joins a currently running session, the membership management algorithm has to ensure that the collaboration session eventually comprises the current participants and the new participant and they can collaborate in the session correctly as if all of the participants, including the new one, have been in the session from the start of the session.

Requirement 5-3. *New Joining Site*

Let S_i be a new site that joins a currently running collaboration session, S_i has to satisfy the following requirements:

- S_i has to acquire the latest state of the document, and
- S_i has to be able to process all future incoming operations while maintaining document consistency, and
- S_i has to know other sites that should receive all locally generated operations.

Requirement 5-4. *Adding a new site*

Let S_i be a new site that joins a currently running collaboration session, all other participants have to ensure the following:

- they have to eventually know that there is a new site S_i, and
- they have to add S_i to their list of participants such that they will send the future operations to all other participants, including S_i, and
- they have to add a new element associated with S_i into their state vector such that they could take S_i into account in determining operations precedence and concurrency in order to correctly preserve user intentions.

5.2.3. Leaving the Session

A site S_i may decide to leave the session when it is no longer interested in the session. Commonly, the site that decided to quit will send a 'quit' notification to the rest of the participants indicating that it is leaving the group. Upon receiving the notification, a site will remove the quitting participant from its participants list and delete the element associated with the quitting site from the state vector. Eventually, the session will go on with one less participant, and the other participants can resume collaboration correctly after making the necessary adjustment.

This event is easily handled in wired network environments, where an adequate 'quit' notification is broadcast whenever a site leaves the group and the 'quit' notification is highly-likely to arrive at all participants. This however would not be the case for mobile network environments. Firstly, the 'quit' notification is not guaranteed to be received by all participants, thus not all participants will know about this event and thus might not be able to adjust themselves accordingly. Secondly, there might not even be an adequate 'quit' notification at all. In a wired network, the sites are always connected, thus S_i could send an adequate 'quit' notification. On the other hand, in a wireless network, connectivity is not guaranteed, therefore S_i might not be able to send the 'quit' notification in the first place. Furthermore, a site may get disconnected indefinitely and thus the user of that site might decide to stop participating due to frustration or other reasons. Thus, if S_j has not received any operation from S_i for a considerably long period, S_j will not have any idea whether S_i has left the group, S_i is being disconnected from the group, or the user at S_i is simply inactive.

To correctly handle this event, the following requirement (Requirement 5-5) has to be satisfied by a membership management algorithm.

Requirement 5-5. *Quitting Site*

Let S_i be a site that decides to leave the group, the following must hold:

- all sites eventually know that S_i has left the group, and remove S_i from their list of participants, and
- all sites continue the session correctly without S_i.

Requirement 5-5 requires that when S_i quits the session, all sites must agree that S_i has left the group. However, in a wireless network, this kind of agreement is not necessarily easy to reach. The following are possible scenarios in regards to S_i leaving the session.

1. S_i broadcasts a 'quit' notification and *all* participants receive the notification.
2. S_i broadcasts a 'quit' notification and *not all* participants receive the notification.
3. S_i does not broadcast a 'quit' notification at all.

In the best scenario, scenario 1, all sites will know that site S_i has left and therefore adjust their states accordingly. On the other hand, in scenario 3, no site knows that S_i has left. The collaboration can still continue even though the bandwidth may be wasted trying to send operations to S_i. In scenario 2, however, some sites know that S_i has left the group and have removed S_i from their data structures accordingly, while some other sites still assume S_i is still in the groups.

Therefore, when site S_j receives the quit notification of site S_i, S_j will remove S_i from its participant list and eventually all sites must receive the quit notification, either directly from S_i or indirectly from S_j. During this period, however, the collaboration must still continue without any interruption. This process is formally stated in the following Requirement 5-6.

Requirement 5-6. *Removing a Site*

If site *Si* has assumed that site *Sj* has left the group and removes *Sj* from its internal data structure, then:

- all other sites will eventually remove *Sj* from their participant lists, and
- the collaboration still continues correctly during and after the removal process.

5.3. Related Work

The two major approaches to managing group communication (and membership) in distributed system environment are Group Membership Services (GMS) [10, 11, 41, 42, 50, 71, 75, 76, 99, 116, 117] and IP Multicast [40]. While IP Multicast manages the group membership in the network layer where each site needs to send messages to only one multicast address, GMS works in the application layer where each site is aware of who the other members are and sends messages to a targeted group id. The first two subsections discuss these two areas of existing work and outlines why they are not suitable for mobile real-time collaboration.

The following subsection discusses two session management techniques: explicit and the implicit session management. It also discusses how a site discovers and joins a collaboration session under each session management technique. Finally, the last subsection discusses some of the groupware systems that are known to have mechanisms to handle membership problems and explains why they are not applicable to mobile replicated architecture.

5.3.1. Group Membership Service

GMS provides multipoint-to-multipoint group communication by organizing processes/sites into groups. Whenever a site needs to communicate with the other group members, it sends a message targeted to the group and GMS delivers the message to the group members [75]. A process becomes a group member by requesting to join the group and it can cease being a member by requesting to leave the group or by failing. Roman et al [117] defines the group membership maintenance problem as the requirement for each host to have knowledge of what other hosts are members of its group and for such knowledge to be consistent across the entire group at all times.

A group membership service is responsible for ensuring that the above requirement is met by *capturing* any changes in the group membership and *notifying* all group members of the new membership configuration. Coulouris et al [36] outlines the main tasks of a GMS as follows:

- *Providing an interface for group membership changes*

  The membership of a group is dynamic due to sites that voluntarily join and leave the group, those that need to be excluded due to failures and those that need to be included after repairs [12]. To be able to capture any dynamic changes in the group membership, the GMS has to provide a means for the group members to notify the

GMS when a member needs to create or destroy process groups and when a member needs to be added into a group or a member decides to withdraw from the group.

- *Implementing a failure detector*

  One of the important elements of a GMS is a failure detector to monitor the group members and to correctly detect when a group member crashes or becomes unreachable due to a network failure. In either case, such group member would not be able to use the interface provided by the GMS as mentioned above. Therefore GMS needs to have some mechanism to detect this so that it captures the correct network conditions at any given time.

- *Notifying members of group membership changes*

  After receiving any notification from any particular member and/or detecting some unreported events (i.e. disconnection or site crash), GMS constructs a *report* to be sent to the other group members. This report is called a *view*. A view basically consists of a view identifier and a list of current active members. If two members receive the same view, they are thought to have the same perception of the group membership [76]. Therefore, a GMS tracks changes in group membership and transforms them into views that are agreed upon as defining the group's current composition [12].

In terms of membership, the distributed system configuration is categorized into two: *primary-partition* system and *partitionable* system. In primary-partition systems, there can be at most one view of the group active at any time [121]. In contrast to a primary-partition system, a partitionable system is defined as a system that allows multiple views of the same group to exist concurrently, i.e. several different views of the membership of the group may evolve concurrently and independently from each other [11, 41, 42, 50, 71]. In such systems, membership of a group may change dynamically not only due to individual process failures and recoveries, but also due to subsets of correct processes becoming isolated and later re-merging [12].

Implementing a GMS in a LAN environment is feasible due to the stability provided by LANs. However, implementing a GMS in a less stable environment is a challenging task. Keidar et al. [75] pointed out the following issues that need to be addressed to successfully implement a GMS in a wide area network (WAN):

- *High latency*. Message latency tends to be large and highly unpredictable in a WAN, as compared to a LAN. In addition, message loss is quite common in WANs. Each time a message is lost, the message needs to be retransmitted, thus delaying the message even further. In order to reach a consensus or agreement, sites repeatedly exchange messages, be they notification messages or acknowledgement messages. The higher the latency, the more difficult, if not impossible, for this to be done. Furthermore, the message might be out of date by the time it reaches the destination. A GMS needs to ensure that it still works well in such a hostile network environment.

- *Frequent changes*. Connectivity changes are more likely in a WAN than in a LAN. The membership algorithm needs to capture these changes and construct a new view to reflect the changes. The more frequent the changes are, the more frequent the views change and the more resources needed by sites to engage in membership changes.
- *Instability*. Bandwidth fluctuation and path congestion contribute to instability in a WAN. A group membership algorithm should be able to handle such network instability that occurs unpredictably and indefinitely.

Those challenges are even more severe in mobile network environments, especially since mobile users periodically move from one place to another which creates further instability and frequent disconnection. In addition to those challenges, GMS has some characteristics that are inapplicable in wireless networks, especially ad hoc networks such as the following.

- *It requires sites to periodically broadcast beacons to signify their presence to other sites*. This beacon is useful in determining whether one site is currently reachable from (i.e. connected to) another site. For example, Roman et al. [117] requires every host to periodically broadcast a 'hello' message that contains its location information and its group id. This requires adequate available bandwidth to allow the regular broadcast and good connectivity for successful beacon delivery. Such requirements however are not necessarily satisfied in wireless network.

- *It requires a membership server* [75] *or a site to serve as the group leader* [117]. The leader of a group frequently checks the status of the members. If there are any changes to the group membership configuration, the server or the leader sends notifications to the members. This will only work if the presence of a dedicated server is guaranteed and if the connectivity between the server and the members is stable, which are not the common case in wireless networks especially ad hoc wireless network environments.
- *It is almost, if not, impossible to maintain a consistent view of group membership in asynchronous systems, especially in the presence of unannounced disconnections* [117]. However, unannounced disconnections commonly occur in ad hoc wireless networks. Some existing work [10, 40, 41, 50, 116] claims that if an unannounced disconnection occurs, the group membership problem can be solved by removing or killing the processes that are suspected to have crashed. However, Chandra et al. [29] prove that the primary partition group membership problem cannot be solved in asynchronous systems with unannounced failures (either due to crashes or sudden disconnection), even if the faulty processes (or those suspected to be faulty) are removed or killed. Neiger [99] propose a primary partition group membership specification that is solvable in asynchronous systems when there is a process crash. However, with the proposed specification, all other processes are blocked whenever a single process crashes. Therefore, the proposed specification does not promote liveness and is not fault-tolerant. In fact, it is impossible to create a GMS that is

able to always inform members of the correct state of the membership and to always agree on the views delivered to different members [29] and in order for “useful” communication to happen, group members must be given the same views [76]. It is possible for a host that is still in the old configuration to receive a message from a host that has already reached the new configuration. In such a case, the recipient must postpone the processing of this "future" message until the new configuration is established, thus pretending that the message is "received" in the new configuration.

- *The view agreement is only reached when the network becomes stable*. When the network is unstable, the membership changes constantly, thus making it difficult to reflect the correct membership at any given time. However, in a mobile network environment, the network might never become stable. Moreover, all of the existing algorithms require servers to know that an agreement has been reached before generating the new view. This requires multiple rounds of message exchanges [76], and due to network instability, the actual group membership might have already changed before the agreement on the previous view is reached.
- *One important element of GMS is a notification service that is responsible for monitoring sites and detecting any site failures*. However, it is impossible to distinguish a crashed process from one that is just very slow or currently disconnected and Fischer et al. [53] show that any problem requiring "all correct processes" to take some action cannot be solved deterministically.

5.3.2. IP Multicast

IP multicast is a weak case of a group membership service [76]. It does not provide group members with the information about who the other members are. Each site needs to know only one multicast address to send messages to. IGMP is the protocol used to manage the membership of the group.

The main aim of IP multicast is to provide communication channel transparency to the members such that the members would not have to worry about sending messages to multiple destinations, instead it sends messages to only one multicast address and the network routers will do the rest ensuring the messages arrive at all desired destinations. In order for IP multicast to work, there needs to be some routers that are willing to forward multicast messages and are running a certain IP multicast algorithm such that it ensures all desired destinations receive the messages. Furthermore, these routers have to always be connected so that the packet forwarding will work well and the packet can be forwarded through to the final destinations. However, the presence of such routers is not necessarily guaranteed in mobile network environments, and not all mobile devices are willing to route packets. Moreover, even when routers are present, they are not necessarily connected all the time.

In ad hoc wireless network environment, each device has to build a multicast tree that represents the current network topology so that each packet that is sent knows what path it has to take to go through to all group members. If there are changes in router connectivity (changes in topology), all devices have to rebuild their multicast trees to represent the current topology. Unfortunately, due to mobility and frequent disconnection, the wireless network topology changes rapidly. This not only requires the devices to constantly monitor the changes in topology, it also forces each wireless device to frequently rebuild its tree every time the topology changes. If the topology changes very frequently, the mobile device ends up wasting its resources and time building and rebuilding the tree, and it is not able to fully participate in a collaboration session.

5.3.3. Session Management Model

A session management model defines the manner in which people can join together in collaboration systems [108]. Session management models can be categorised into *implicit* and *explicit* session management. In an explicit session management model, the participants in the collaboration are required to take some action and time to join the session. The implicit form of session management, on the other hand, requires less initial overhead.

Most collaborative applications built to date have employed explicit session management [44]. The two common approaches to explicit session management are *initiator-based* and *joiner-based*. In an initiator-based session management, the initiating participant invites other users to the collaboration session. The initiator site notifies users of the existence of the collaboration session and provides a means for collaborating with the others in the session. In contrast, in joiner based session management, users who want to join a session must find the session by discovering currently active sessions or by knowing a priori that the session is or will be taking place. The characteristics of collaboration that uses explicit session management are usually planned, long-term and the session is properly named so the users recognise the session that they want to join.

The implicit session management, however, is usually serendipitous, spontaneous, transient and unnamed. For example, to collaboratively edit a file, users would simply edit the same file. The system would detect the potential for collaboration inherent knowing that multiple users are working on the same object. There is no need for naming sessions or browsing lists of sessions to join the session. The implicit session management obviously avoids the overhead of the explicit session creation, naming, and browsing phase. In contrast to the explicit forms of session management (initiator-based and joiner-based), there are three forms of implicit session management models [79].

- *Artifact based* models assume that people wish to join together in sessions when they use the same artifact or document (e.g. the session management described in [44]).
- *Activity based* models assume that people wish to join together in sessions when they are involved in the same activity, e.g., using the same system (e.g., Piazza[69], Hummingbird [64], Meme Tags [19], and Hocman [48, 49]).
- *Place based* models assume that people wish to join together in sessions when they are at the same gathering point (e.g., Teamrooms [120]).

Although the implicit session management model requires less overhead than the explicit counterpart when a user wants to join in a session, it imposes additional resource consumption in detecting and discovering the peer or the other collaboration participant (group awareness). Hummingbird [64] uses a special device to monitor the presence of other Hummingbirds in the close proximity. GroupWear [18] uses an active badge system that lets user share and compare their answers to a set of multiple-choice questions. The Meme Tags System [19] also provides mechanisms to monitor other users' presence, but at a shorter range than the Hummingbird device. In cruise mode upon discovering a new peer, Hocman [48, 49] will perform an automatic background download of a predefined index-page. In explore mode, Hocman provides group awareness. Whenever the users are in the vicinity of each other, they will be appended in each other's list of accessible peers. In Pirates! [51], in addition to the WLAN adapters, each handheld device is fitted with custom-made proximity sensors, used to determine the players' location in physical space. Furthermore, some use a dedicated server as the session manager that coordinates the joining [151]. Finally, in the artifact based model, two different sessions may have the same document or artifact name causing a user to join the wrong collaboration session. A similar mistake can also happen with the place-based model, where two or more users are at the same gathering point but they do not intend to collaborate with each other.

Therefore, this research assumes the use of an explicit session management model where the user who wants to join a session will contact a current participant. This thesis, however, does not discuss the detail of how users discover sessions and other participants. An existing discovery protocol can be adopted for this process.

YCab [22], SOCT2 [126], and CoWord [134] are three examples of real-time collaborative editors that implement explicit group membership and session management algorithms.

YCAB

YCab handles packet loss by slowing down the rate of operation broadcast**.** Each site implements a method to adjust the rate of transmission to accommodate varying bandwidth and latency, although ideally, the client should adjust the send delay automatically according to the network condition. By adjusting the send delay, YCab could self-adapt to the network conditions in order to reduce packet loss.

With regards to late join in YCab, one of the members in the session (the coordinator of the session) is given the responsibility of bringing the new client up to speed. To join a session, a new site, say site S_i, broadcasts the join request to all existing members and has to wait for their replies. Once all replies are received, S_i will assign itself a rank (*id*) equivalent to 1 + the highest rank in the received replies. S_i then broadcasts the join session message with its assigned rank and then all existing members will process the join session message and add S_i accordingly. This procedure, however, requires all members to be connected during the join session process. If one or more existing members are disconnected during the joining, the new client will not receive all necessary replies, hence, will derive a wrong rank (id). Furthermore, if the new client does not receive replies from anyone within a specific period of time, it will create a new session even though that is not necessarily what it wants. YCab also requires a session coordinator during a state recovery process to bring up any client up to speed with the current session state, therefore, introduces a single point of failure.

Although YCab, by adjusting the send delay, could self-adapt to the network conditions in order to reduce packet loss without manual configuration, it does not discuss how to recover lost messages and how to ensure that the messages will eventually be received by all intended recipients. Relying on the session coordinator to execute state recovery will add to the burden of the session coordinator and increase the severity of the single point of failure.

SOCT2

In SOCT2, Suleiman et al. [126] introduces the applicability of the algorithm to mobile sites by presenting disconnection and reconnection procedures. If a mobile site is about to disconnect, it will notify the other sites that it is disconnected, and then the user of the disconnected site will continue to work on the local replica. The reconnection of a site involves two sub-procedures:

- Updating other sites of its modification during the disconnection period. This is done by simply broadcasting the operations done during the disconnection period to other sites.
- Getting updates from other sites. This can be done either by collecting operations done by other sites during the disconnection period or by nominating one site to

> send across its local replica and its history. The latter is selected due to its simplicity. The reconnected site *S* designates another live site S_v to send its document state to. Before sending it, S_v has to be sure that it has already received and executed all operations generated by other sites. This is done by receiving an acknowledgement from all other sites that they have received the reconnection message from *S*. All other operations that arrive to S_v after the acknowledgement will not be executed until the end of this reconnection phase.

They however only discuss the event of voluntary disconnection with notification. What commonly occurs in mobile network environments is an involuntary disconnection without notification. The disconnected site does not have a chance to send notifications to other sites that it is about to be disconnected. Moreover, the disconnections occur frequently, thus it is difficult to determine which operations are done during a disconnection period. During the reconnection phase, not only is sending of reconnection and acknowledgement messages expensive, it is also difficult for *S* and S_v to determine which sites are currently connected, thus *S* might end up sending reconnection messages to currently disconnected sites and S_v might wait for acknowledgement from the disconnected sites for an indefinite period. However, this initial approach provides a good basis for handling this limitation of mobile network environments, which will be further discussed in section 5.4.1.

CoWord

Sun et al. [134] proposed a quiescence based approach to accommodate late comer in CoWord and CoPowerPoint. The quiescence-based approach depends on all sites to reach a quiescence state, where all sites have received and executed all generated operations, and hence all sites are in a consistent state. The quiescence state is achieved by broadcasting synchronization messages to push all operations to arrive and be executed at destinations, and temporarily blocking the users from generating new operations. Under the quiescent condition, all existing CoWord/CoPowerPoint instances are guaranteed to have consistent and identical document replicas. Consequently, initializing a late-comer is done simply by transferring the copy of the current document to the late-comer, and initializing the state vectors and history buffer to empty.

This approach, however, can only happen if the it is assumed that (1) each operation will always arrive at destination once it is sent, (2) all sites are connected when the late-comer is joining, (3) the synchronisation message arrives at all sites, and (4) there is a way to confidently determine that all sites are no longer sending any operation after the synchronisation message is sent. In mobile replicated architecture, however, none of the above can be assumed, hence waiting for quiescence state before a late comer can be accommodated can mean all sites are held up indefinitely and/or some sites are not necessarily in their quiescence state consistent with other sites. This thesis propose a technique to accommodate late comer by taking into account the above limitations of mobile environments.

5.4. Algorithm

Based on the above discussion, this section proposes a membership management algorithm to handle membership events in real-time mobile collaboration systems. The proposed algorithm is built on top of the consistency and conflict management algorithms presented in the previous chapters.

The following are requirements that need to be met by the proposed algorithm in order to function effectively in mobile real-time collaboration environments.

1. The algorithm must not require a group leader or dedicated server to avoid a single point of failure and to promote ad hoc collaboration.
2. The algorithm must not rely on beacon broadcasts for determining the connectivity among the sites, i.e. each device does not need to detect any changes in network topology, thus focusing its resources primarily on participating in the session.
3. The algorithm must not constantly monitor and maintain the current view of who the other members are (refer to section 5.3.1)
4. The algorithm must ensure that all sites eventually receive all operations.
5. The algorithm must ensure that document consistency across all sites is maintained (i.e. the algorithm is built on top of the previously proposed consistency management algorithm)

As mentioned in section 5.2, there are three membership cases to be handled: (1) *Late Join*, (2) *Missing Operations*, and (3) *Quitting Site* cases. The proposed algorithm is constructed by addressing each membership case separately. A solution for each case is first proposed, and then combined into one complete algorithm that is able to handle all the different membership events. Due to the nature of the solution, this section begins by discussing and solving the *late join* case (Section 5.4.1 discusses the single late join and Section 5.4.2 discusses the concurrent late joins), followed by the *missing operation* case and the *quitting site* case.

5.4.1. Late Join

Due to concurrency and various membership events, each site will be in a different state at the time S_i decides to join the session. Each site will maintain its own current state of the document replica, which is not necessarily the same as the state of the other sites' replicas. In addition to that, each site will have its own current site state (i.e. state vector) which is not necessarily the same as other sites. This means that in order for a new site S_i to participate in the session, S_i has to:

- acquire an updated document from the contacted site (S_j), and
- acquire the complete operations history of S_j so that this new site is able to process any future incoming operations (i.e. correctly transform remote operations such that user intentions are preserved to ensure document consistency), and

- acquire the latest state of the contacted site S_j so that the new site S_i can resume the collaboration just as S_j resumes the collaboration after the joining process.

In other words, this new site has to be an exact replica of one of the current participants (i.e. S_j). If S_i gets the document from S_j, S_i has to acquire the operation history and other information from S_j. If S_i gets the operation history from a site other than S_j, site S_i might not be able to correctly transform future remote operations, thus the replica held by S_i will not be consistent with other replicas.

Therefore, whenever a new site S_i joins a collaboration session via S_j, the following procedure has to be executed:

1. S_j constructs a *state* message (σ_{S_j}) that consists of:
   - The latest version of the replica held by S_j (R_{S_j}), and
   - the operations history of S_j (H_{S_j}), and
   - the list of current participants (P_{S_j}), and
   - the current site state of S_j that consists of the logical clock (LC_{S_j}) and state vector of S_j (V_{S_j})
2. *Sj* sends the state message to *Si*.
3. Upon receiving the state message, S_i adjusts its state with the information provided by the state message.

4. An element in the state vector associated with S_i is added at the end of the state vector of S_i and S_j.

Consider the scenario depicted in Figure 5-1, where there are currently 7 sites in the session (the sites that are within one grey coloured area are connected to each other). If site 8 decides to join the session via site 7, then site 8 will do the above procedure and become the exact replica of site 8, and only site 7 and site 8 know that site 8 has just joined the session (Figure 5-2).

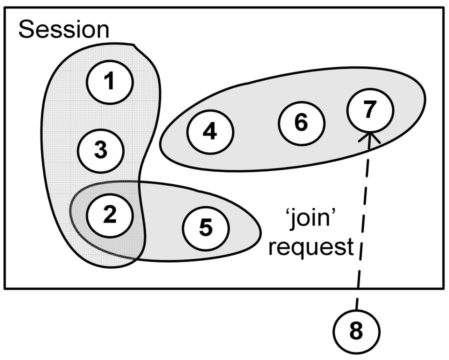


Figure 5-1 A site joins a session

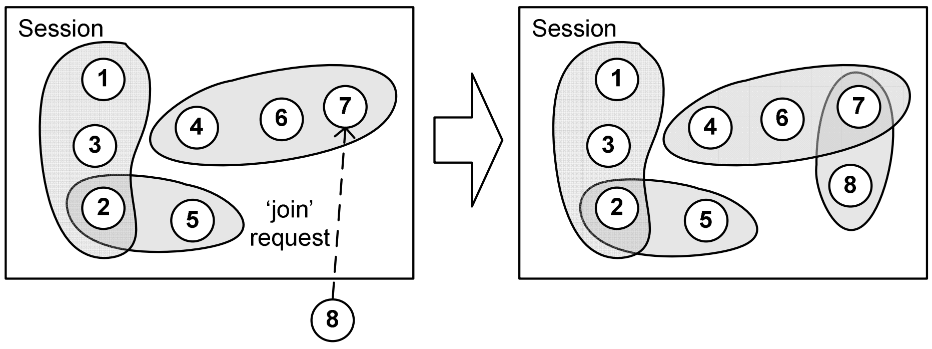


Figure 5-2 After joining a session

The above procedure requires each site to be able to: send a join request to other sites (*send_join_req*), receive a join request from a new site (*rcv_join_req*), and receive the state message from another site (*rcv_state_msg*). Those functions are outlined in Figure 5-3.

```
void send_join_req($S_j$) {
        /*send join request to $S_j$*/
        send_join_req($S_j$);
}

void rcv_join_req($S_i$) {
        /*construct state message to be sent to $S_i$ */
        $\sigma_S = < R_S, H_S, P_S, LC_S, V_S >$;

        /* send state message to $S_i$ */
        send($S_i$, $\sigma_S$);

        /* update state vector */
        /* add an element associated with $S_i$ */
        $V_S = V_S + \{0\}$;

        /* add $S_i$ to the list of participants */
        $P_S = P_S + \{S_i\}$;
}

void rcv_state_msg( $\sigma_{S_j}$ ) {
        /* explode state message */
        $< R_S, H_S, P_S, LC_S, V_S > = \sigma_{S_j}$

        /* update state vector */
        /* add an element associated with $S_i$ */
        $V_S = V_S + \{0\}$;

        /* add $S_j$ to the list of participants */
        $P_S = P_S + \{S_j\}$;
}
```

Figure 5-3 Late join procedure

After completing the late join procedure, the newly joined site S_i becomes an exact replica of the contacted site S_j. Only S_i and S_j know that there is S_i in the session. The other sites do not know this as yet. S_i can explicitly send notifications to all other sites that it has joined the session, or it can do this implicitly. In the future, all operations generated by S_i and S_j will have an extra element in their state vector. This element will implicitly 'tell' other sites that there is a new site in the session. When the other sites know that there is a new site, then they add a new element to their state vectors and all future generated operations.

5.4.2. Concurrent Late Join

The above procedure solves the late join problem when only one site decides to join at a time. The new element of the state vector associated with the new site is appended at the end of the vector ($V_S[N]$). However, if there are two or more sites wanting to join at the same time, the addition of the new element of the state vector is not as easy as appending it to the end of the vector.

Consider the scenario in Figure 5-4 where site 8 and site 9 want to join the session concurrently through different participants. Following the above procedure, site 7 replicates itself to site 8 and site 3 to site 9. Before the join process, site 7 and site 3 have their state vector as $V_7 = \{V_7[0], V_7[1], \ldots, V_7[6]\}$ and $V_3 = \{V_3[0], V_3[1], \ldots, V_3[6]\}$ respectively. After the join process, site 7 will add an element in its state vector such that $V_7[7]$ represents site 8. However, concurrently, site 3 will add an element in its state vector such that $V_3[7]$ represents site 9. When, sometime in the future, site 7 receives a message from site 3 and deduces that there is a new site, site 9, then site 7 will add an element in its state vector such that $V_7[8]$ represents site 9. The same thing happens when site 3 receives a message from site 7, such that $V_3[8]$ represents site 8. This causes conflict, and the precedence and concurrency could not be determined properly, potentially causing document inconsistency. The core of the problem here is that the site naively adds an element to the end of the state vector to represent a new site, where the other site might be doing the same thing concurrently for another new site. Therefore, a technique called *state map* is proposed to ensure that the contacted site can add an element to the state mechanism without causing conflict with other sites.

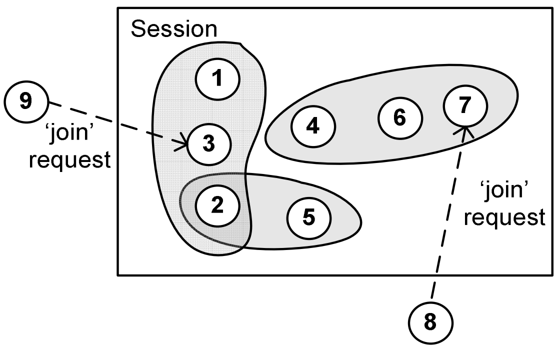


Figure 5-4 Concurrent late join

A state map, similar to a state vector, is a data structure that records the number of executed operations. If a state vector is in the form of an array (vector), where each element is indexed by an integer number, each element of a state map is mapped by keys (not necessarily an integer number). Using a state vector, each element of the state vector has to be associated with the respective site using an integer index. The first site in the session is associated with the first element in the state vector; the second site has the second element and so on. A new site will be assigned an index one greater than the current largest index in the session. However if there is more than one site joining at the same time, it is difficult to determine which new site is assigned to which element in the state vector. Using a state map, each site is assigned a place in the map without having to worry about the position in the map. A site needs only to choose a unique key to represent itself in the map. The key does not have to be an integer and it can be as simple as the site's IP address or the sites hostname or even the hash value (e.g. Md5sum) of either one.

In Figure 5-4, when site 7 grants the 'join' request of site 8, site 7 will add 8 into its list of participants, and site 7 will also add an entry in its state map with the key of site 8 (might be the IP address of site 8) and zero as its initial value. At the same time, site 9 joins via site 3, thus site 3 will add an entry in its state map with the key of site 9. When the information of these two newly joined sites is disseminated to all sites, the state map will uniquely identify each new site. For example, the state map of site 7 before the join process is $M_7 = \{key(S_1) \Rightarrow val(S_1), key(S_2) \Rightarrow val(S_2), \ldots, key(S_7) \Rightarrow val(S_7)\}$, where key(s1) is the unique key of site 1 and val(s1) is the number of operations generated by site 1 that have been executed by site 7. After the join process, its state map becomes $M_7 = \{key(S_1) \Rightarrow val(S_1), key(S_2) \Rightarrow val(S_2), \ldots, key(S_7) \Rightarrow val(S_7), key(S_8) \Rightarrow val(S_8)\}$. Concurrently, after site 9 joins through site 3, $M_3 = \{key(S_1) \Rightarrow val(S_1), key(S_2) \Rightarrow val(S_2), \ldots, key(S_7) \Rightarrow val(S_7), key(S_9) \Rightarrow val(S_9)\}$. Notice that the new element of each state map is appended at the end of the state map, but they have different keys such that they do not need to be placed in a particular location in the state map. Eventually, both sites will end up with $M =$

{ $key(S_1) \Rightarrow val(S_1)$, ..., $key(S_8) \Rightarrow val(S_8)$, $key(S_9) \Rightarrow val(S_9)$}. Figure 5-5 outlines the updated procedure to accommodate concurrent late joins.

```
void send_join_req($S_j$) {
        /*send join request to $S_j$*/
        send ($S_j$, JOIN);
}

void rcv_join_req($S_i$) {
        /*construct state message to be sent to $S_i$ */
        $\sigma_S = < R_S, H_S, P_S, LC_S, M_S >$;
        /* send state message to $S_i$ */
        send($S_i$, $\sigma_S$);

        /* update state map */
        /* add an element associated with $S_i$ */
        $M_S = M_S + \{key(S_i) \Rightarrow 0\}$;
        /* add $S_i$ to the list of participants */
        $P_S = P_S + \{S_i\}$;
}

void rcv_state_msg( $\sigma_{S_j}$ ) {
        /* explode state message */
        $< R_S, H_S, P_S, LC_S, M_S > = \sigma_{S_j}$

        /* update state vector */
        /* add an element associated with $S_i$ */
        $M_S = M_S + \{key(S_i) \Rightarrow 0\}$;

        /* add $S_j$ to the list of participants */
        $P_S = P_S + \{S_j\}$;
}
```

Figure 5-5 Late join procedure to handle concurrent late join

The other sites might not establish that there are new sites in the session until they receive an operation that originates from either the new site or from another site that already knows about the new sites. For example, site 7 generates and broadcasts an operation op_i after site 8 joins the session. Operation op_i will bear a state map with an extra element associated with site 8 ($M_{op_i}[key(S_8)]$). When op_i arrives at site 4, site 4 will be able to determine that there is an additional element in the state map of op_i, $M_{op_i}[key(S_8)]$. Hence, site 4 knows that there is a new site, and adds the new element into its state map ($M_{S_4}[key(S_8)] = 0$), and adds site 8 into its participants list. From this point onwards, all operations that originate from site 4 will bear a new element in their state map associated with site 8. This procedure has to be executed whenever a site receives a remote operation as outlined in Figure 5-6.

Therefore, to allow the algorithm to handle a concurrent late join, the state map technique is used, replacing the state vector. Consequently, every site maintains a state map instead of a state vector and each operation will carry a state map which is equal to the state map of the originator site when the operation was generated. With the use of state maps, the procedure to determine precedence and concurrency is modified as in Figure 5-7.

```
void $rcv\_remote\_op(< op, LC_{op}, M_{op}, S_{op} >)$ {
    /* check for any new site */
    for (all $k \in keysOf(M_{op})$) {
        /* check if this site has element $k$ */
        if ($M_S[k] = \emptyset$) {
            /* add a new element with key $k$ and value 0 */
            $M_S = M_S + \{k \Rightarrow 0\}$;

            /* add the new site to participants list */
            $P_S = P_S + \{S_k\}$;
        }
    }

    /* check if $S_{op}$ has not known $S$ */
    if ($M_{op}[key(S)] = \emptyset$) {
        /* treat op as if it has $M_{op}[key(S)]=0$ */
        $M_{op} = M_{op} + \{key\{S\} \Rightarrow 0\}$;
    }
    :
    :
    :
}
```

Figure 5-6 Checking for a new site

```
boolean $does\_precede(op_i, op_j)$ {
    if ($M_{op_j}[key(S_{op_i})] > M_{op_i}[key(S_{op_i})]$) {
        /* $op_i$ precedes $op_j$ */
        return $true$;
    }

    /* otherwise $op_i$ does not precede $op_j$ */
    return $false$;
}
```

Figure 5-7 Determining operations precedence using state map

It is worth pointing out that the history trimming presented in Chapter 3, as part of the document consistency management algorithm, uses the state vector and the state vector table to determine which operations can be removed from the operation history. With the introduction of the state map in place of the state vector, the history trimming procedure must be modified for the state map technique. Similar to the state vector technique, each site maintains a *state map table (MT)* that contains information about the state maps of all other sites. $MT_i[j]$ $(1 \leq j \leq N)$ is the state map of site S_j as known by site S_i, and $MT_i[j][key(S_k)]$ is the number of operations generated from site S_k that have been executed by site S_j as known by site S_i. Whenever a remote operation *op* from site S_j is executed at site S_i (note that $M_{op} = M_{S_j}$ at the time *op* was generated), $MT_i[j]$ is updated to be equal to M_{op} to ensure $MT_i[j]$ is as up to date as possible. Let op_a be an operation generated from site S_k. Sites that have already executed op_a will have $M_S[key(S_k)] \geq M_{op_a}[key(S_k)]$, thus all operations op_i that op_a precedes will have $M_{op_i}[key(S_k)] \geq M_{op_a}[key(S_k)]$. If site S_i receives an operation op_x from site S_m, site S_i will know that site S_m has already executed *op* if $M_{op_x}[key(S_k)] \geq M_{op}[key(S_k)]$.

Each site also maintains a Minimum State Map (*MSM*). The MSM_i reflects the knowledge of site S_i about the number of operations that have been executed at every site ($MSM_i[key(S_j)]$ = the number of operations generated by site S_j that have been executed by every site as known by S_i). Initially $MSM_i[key(S_j)] = 0$, $\forall j \in \{0,...,N\text{-}1\}$. After executing an operation and updating other elements of the MT_i, site S_i updates $MSM_i[key(S_j)]$ as follows:

$MSM_i[key(S_j)] = \min(MT_i[k][key(S_j)]), \forall k \in \{0,...,N-1\}$. If the value $MSM_i[key(S_j)] = m$, then the first *m* operations generated at site S_j must have been executed at all sites.

Therefore, if an operation *op* is generated from site S_k, *op* can be deleted from the history of S_i if $M_{op}[key(S_k)] \le MSM_i[key(S_k)]$ or, in other words, $M_{op}[key(S_k)] \le MT_i[j][key(S_k)], \ \forall j \in \{0,...,N-1\}$. The following figure (Figure 5-8) outlines the updated history trimming procedure:

```
void trim_history() {
    $n$ = size of $H_S$;

    for $i$ = 1 to $n$ do
        $<op_i, LC_{op_i}, M_{op_i}, S_{op_i}> = H_S[i]$;
        deleted = false;

        if $M_{op}[key(S_k)] \le MSM_i[key(S_k)]$ then
            $H_S = H_S - H_S[i]$;
            deleted = true;
        endif;

        if deleted = false then exit;
    endfor;
}
```

Figure 5-8 The updated history trimming procedure

5.4.3. Missing Operations

Site S_i is disconnected from site S_j if S_i is unreachable from S_j, whether it is because of the unavailable bandwidth, S_i is not within the transmission range of S_j or S_i has crashed.

When S_i is disconnected from S_j, any messages generated and broadcast by S_j will not arrive at S_i. Therefore, S_i will lose any messages or operations sent by S_j during the disconnection period.

When site S_i reconnects to site S_j, site S_i will start receiving operations from site S_j again. When S_i receives an operation *op* from S_j after being disconnected for a while, S_i might not be able to deliver or execute *op* straight away because there are some operations that precede *op* that were sent while S_i was disconnected. Therefore S_i needs to identify the operations that it has missed during its disconnection period and it has to request those operations be sent to S_i. A remote operation *op* that arrives at S_i carries a state map that tells S_i about the state of the originator site when *op* was generated. By comparing state map of S_i (M_{S_i}) and the state map of received operation (M_{op}), S_i will be able to detect whether there are some missing operations that it needs to request. $M_{S_i}[key(S_j)]$ signifies the number of operations originated from S_j that have already been executed by S_i, and $M_{op}[key(S_j)]$ signifies the number of operations originated from S_j that have already been executed by S_j just before *op* was generated. By comparing $M_{S_i}[key(S_j)]$ and $M_{op}[key(S_j)]$, site S_i is able to deduce that it has been missing some operations from S_j. This situation also applies when site S_j receives an operation op from site S_i. By comparing $M_{S_j}[key(S_i)]$ and $M_{op}[key(S_i)]$, site S_j knows what operations that it has not received from S_i .

Consider the example in the Figure 5-9, where S_i is disconnected right after receiving op_1 from S_j. During disconnection, S_j generates op_2 and op_3 that are not received by S_i. When S_i is reconnected, op_4 arrives at S_i, with $M_{op_4} = \{S_i \Rightarrow 0, S_j \Rightarrow 3\}$. By looking at this state map, S_i knows that S_j has generated 3 operations before op_4 ($M_{op_4}[S_j] = 3$). Since S_i has already executed 1 operation from site 2 ($M_{S_i}[S_j] = 1$), S_i requests S_j to send two operations generated before op_4, by sending a tuple <S_1, 1, 2>. When S_j receives this request, S_j will send operations that satisfy the condition $\forall op_i : S_{op_i} = S_j \wedge (1 \le M_{op_i}[S_j] \le 2)$.

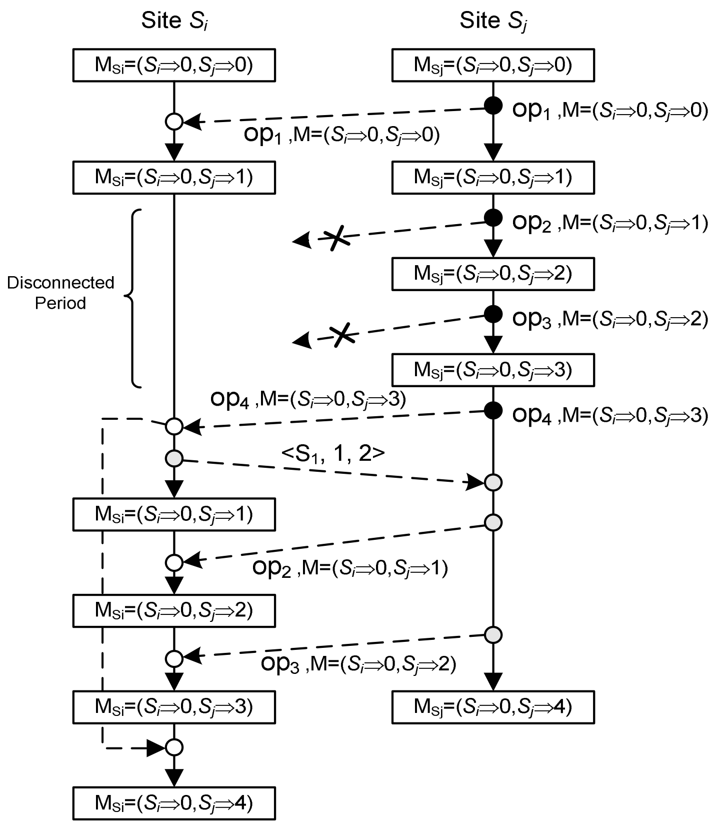

Site S_i
Site S_j
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 0)$
$M_{Sj}=(S_i \Rightarrow 0, S_j \Rightarrow 0)$
op_1 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 0$)
op_1 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 0$)
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 1)$
$M_{Sj}=(S_i \Rightarrow 0, S_j \Rightarrow 1)$
op_2 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 1$)
Disconnected Period
$M_{Sj}=(S_i \Rightarrow 0, S_j \Rightarrow 2)$
op_3 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 2$)
$M_{Sj}=(S_i \Rightarrow 0, S_j \Rightarrow 3)$
op_4 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 3$)
op_4 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 3$)
<S_1, 1, 2>
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 1)$
op_2 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 1$)
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 2)$
op_3 ,M=($S_i \Rightarrow 0, S_j \Rightarrow 2$)
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 3)$
$M_{Sj}=(S_i \Rightarrow 0, S_j \Rightarrow 4)$
$M_{Si}=(S_i \Rightarrow 0, S_j \Rightarrow 4)$

Figure 5-9 Disconnection scenario

In order to detect any missing operation, a site needs to compare its state map with the state map of the incoming operation whenever a remote operation is received. Therefore, the *rcv_remote_op* is modified as in Figure 5-10.

```
void rcv_remote_op(<op,$LC_{op}$, $M_{op}$, $S_{op}$>) {
        /* compare state maps */
        if $M_S[S_{op}] < M_{op}[S_{op}]$ then
                /* request op from  $S_{op}$ */
                send_op_req($S_{op}$, <S, $M_S[S_{op}]$, $M_{op}[S_{op}]$-1>);
        endif;

        /* causal reception */
        wait until $M_S[k] \geq M_{op}[k]$, ($\forall k$: $k \in$ keysOf($M_S$));

        /* execute remote operation by calling
         * the procedure defined in the next phase */
        exec_remote_op(<op, $LC_{op}$, $M_{op}$, $S_{op}$>);
}
```

Figure 5-10 Remote operation reception procedure

When S_i sends a missing operations request to S_j, S_j has to process the request and eventually sends the operations that site S_i asks for. Site S_j then needs to traverse through its operations history to find the operations that S_i requested for and send the matching operations. However, the operation in the history of S_j may have been transformed, thus it is not in its original form. To maintain document consistency, the operations that are going to be sent have to be in their original form as if they are being sent when they were freshly generated by the originator. Therefore, the operation history needs to store operations in their original form in addition to their transformed variants. In the case of a simple text editor, where the operations are simply *insert* and *delete*, storing the original form of the operation means simply storing the original position of the inserted or deleted character. As for other types of applications, the history might need to store the complete original form of the operation.

Figure 5-11 outlines the procedure *receive_op_request* executed by every site *S* to process an operation request message. When the requested operations arrive at site S_i, S_i will treat them as ordinary remote operations by executing the *rcv_remote_op* procedure, resuming the collaboration as usual.

```
void rcv_op_req(<$S_i$, $x$, $y$>) {
    /* find operations in $H_S$ that satisfy
     * $\forall op_i : (S_{op_i} = S) \wedge (x \le M_{op_i}[s] \le y)$ */
    $n = sizeOf(H_S)$
    for $i = 1$ to $n$ do
        <$op_i$, $LC_{op_i}$, $M_{op_i}$, $S_{op_i}$> = $H_S[i]$;
        if $(S_{op_i} = S) AND (x \le M_{op_i}[s] \le y)$ then
            /* send the operations to $S_i$ */
            send($S_i$, <$op_i$, $LC_{op_i}$, $M_{op_i}$, $S_{op_i}$>);
        endif;
    endfor;
}
```

Figure 5-11 Operation request reception procedure

5.4.4. Quitting a Session

A user at a particular site quits a collaboration session when s/he no longer wants to participate in the collaboration. When a site decides to quit the collaboration session, there are three possible scenarios.

The first scenario, the ideal one, is that the quitting site sends a ‘quit’ notification to all other participants, the notification arrives safely to all participants, and they remove the quitting site from their participant lists. The second scenario is similar to the first scenario. However, due to factors such as sudden disconnection and packet loss, the ‘quit’ notification does not arrive to all intended participants, therefore not all sites remove the quitting site from their participant list. The third scenario is that the quitting site does not have a chance to send the ‘quit’ notification at all. The quitting site may be disconnected suddenly or a site that is currently disconnected decides to leave the session permanently. The following discussion describes a proposed algorithm to handle the first and second scenarios. For the third scenario, unfortunately, there is no way to distinguish whether it leaves the session permanently or it is temporarily disconnected. Therefore, until there is an explicit ‘quit’ notification, a site is still considered to be a member of an active session. Future work is needed to address the handling of the third scenario.

The following algorithm is proposed such that all sites will eventually remove the site that has quit from the collaboration session under the first and the second scenarios. The algorithm is described as follows.

1. When site S_i quits a collaboration session, it will broadcast a quit notification to all other participants (Figure 5-12).

```
void quit_session() {
        /* send ‘quit’ notification to all sites */

        for all $S_i$ in $P_S$  do
                send($S_i$, “quit”);
        endfor;
}
```

Figure 5-12 Sending the 'quit' notification

2. When a site receives the ‘quit’ notification, it will remove site S_i from its participant lists and it will remove the element in its state map that is associated with S_i. From this point onward, all operations generated from this site will bear a state map without the element associated with S_i (Figure 5-13).

```
void rcv_quit_notification($S_i$) {
        /* Assume $S_i$ is the quitting site */

        /* update state vector */
        /* add an element associated with $S_i$ */
        $M_S = M_S - \{key(S_i)\}$;

        /* Remove $S_i$  from the list of participants */
        $P_S = P_S - \{S_i\}$;
}
```

Figure 5-13 Receiving the 'quit' notification

3. For the site that does not receive the 'quit' notification from S_i, it will eventually remove S_i from its participant list after it receives an operation from the site that has already received the 'quit' notification from S_i and has removed S_i from its participant lists and its state map. By comparing the state map of the incoming remote operation and the state map of the site, it can implicitly conclude that site S_i has quit the session. Figure 5-14 outlines the additional procedure to the *rcv_remote_op* to check for any site removal.

```
void rcv_remote_op(<op,$LC_{op}$, $M_{op}$, $S_{op}$>) {
        /* compare state maps to check for quitting site */
        if $M_{op}$[k] ≥ $M_S[k]$, ($\forall k$: $k \in keysOf(M_{op})$) AND then
                for (all $k \in keysOf(M_s)$) {
                        if ($M_{op}[k] = \emptyset$) {
                                /* site with key k has
                                 * quit the collaboration*/
                                $M_S = M_S - \{k\}$;

                                /* Remove the site from participants list */
                                $P_S = P_S - \{S_k\}$;
                        }
                }
        }
        :
        :
        :
}
```

Figure 5-14 Implicit 'quit' notification from another site

5.4.5. Conclusion

The above sections discuss the impact of each membership event and propose solutions to each event such that the collaboration session can still continue regardless of the changes in the membership. The disconnected site is able to resume collaboration when reconnected and a late joining site is able to participate in the session as if it has been there since the beginning.

In the late join case, a site must invoke a join request procedure to join the session via one of the current participants. Then the contacted site will send its latest state to the new site. After the new site receives the state and applies it, it can start participating in the session (Figure 5-5). In the disconnection case, a site must ensure that it gets all the operations that it missed during the disconnection and all other sites must ensure they get all the operations that they missed from the disconnected site (Figure 5-10).

For the late join case (Figure 5-5), *send_join_req*, *rcv_join_req* and *rcv_state_msg* procedures were proposed to ensure the new site is brought up to date. Some additions to the *rcv_remote_op* procedure are also proposed to allow sites to recognise the presence of the new site. For the missing operation case, the *rcv_remote_op* procedure is modified to allow the detection of the missing operations and the *rcv_op_req* procedure is used to process the operation request. The proposed procedures are independent of each other except for the *rcv_remote_op* which needs to be modified such that it accommodates the additions made for the late join case and the missing operation case. Figure 5-15 outlines the combined *rcv_remote_op* algorithm to handle those membership problems. Firstly, it checks for any site that has quit the collaboration and if there is a new site by examining the remote operation's state map. Then it compares the state map of the remote operation and the state map of the recipient site to determine any missing operations and send any operation request if necessary. Finally, it executes the operation when it is causally ready.

```
void $rcv\_remote\_op(< op, LC_{op}, M_{op}, S_{op} >)$ {
    /* compare state maps to check for quitting site */
    if $M_{op}[k] \geq M_S[k]$, $(\forall k: k \in keysOf(M_{op}))$ AND then
        for (all $k \in keysOf(M_s)$ {
            if $(M_{op}[k] = \emptyset)$ {
                /* site with key $k$ has quit */
                $M_S = M_S - \{k\}$;
                /* Remove the site from participants list */
                $P_S = P_S - \{S_k\}$;
            }
        }
    }

    /* check for any new site */
    for (all $k \in keysOf(M_{op})$ {
        if $(M_S[k] = \emptyset)$ {
            /* add a new element with key $k$ and value 0 */
            $M_S = M_S + \{k \Rightarrow 0\}$;
            /* add the new site to participants list */
            $P_S = P_S + \{S_k\}$;
        }
    }

    /* check if $S_{op}$ has not known $S$ */
    if $(M_{op}[key(S)] = \emptyset)$ {
        /* treat op as if it has $M_{op}[key(S)]=0$ */
        $M_{op} = M_{op} + \{key\{S\} \Rightarrow 0\}$;
    }

    /* compare state maps for missing operations*/
    if $M_S[S_{op}] < M_{op}[S_{op}]$ then
        $send\_op\_req(S_{op}, <S, M_S[S_{op}], M_{op}[S_{op}]-1>)$;
    endif;

    /* causal reception and execute remote operation */
    wait until $M_S[k] \geq M_{op}[k]$, $(\forall k: k \in keysOf(M_S))$;
    $exec\_remote\_op(<op, LC_{op}, M_{op}, S_{op}>)$;
}
```

Figure 5-15 The complete remote operation reception procedure

5.5. Implementation

The proposed algorithm covers the ways to handle the various membership problems. However, the algorithm provides only the core procedures and the implementation of the algorithm itself may vary depending on the application domain and the amount of available resources. Some issues in the implementation and their impact on performance are therefore discussed in this section.

5.5.1. Storing the original form of operations

Each time a site receives a remote operation (*op*), it has to transform the operation such that it preserves its original intention. This causes operations, both local and remote, to be stored in the history not in their original forms (*op*), but in their transformed variants (*op*'). The transformation and execution of a remote operation might cause some operations in the history to be transformed such that it maintains the consistency.

Consider the following scenario involving two sites, S_i and S_j. Site S_i locally generates $op_i = insert(3,'X')$ and stores op_i as $op_i = insert(3,'X')$ in its history H_{S_i}. Site S_j concurrently generates $op_j = delete(2)$ and stores op_j as $op_j = delete(2)$ in H_{S_j}. When op_j arrives at S_i as a remote operation, S_i has to organise the history and execute op_j such that the intention is preserved. If $op_j \rightarrow op_i$, op_i is transformed such that it becomes $op_i = insert(2,'X')$ when it is stored in H_{S_i}. Therefore, op_i that was locally generated by S_i is stored in its transformed form. This fact can cause a problem when a missing operation is being requested. If S_i misses some operations from S_j and sends an operation request to S_j, S_j has to send the requested operations in their original form as if they are just generated from S_j. If S_j has executed some local and remote operations, it is highly possible that the requested operations have been transformed, thus S_j is not able to send the operations in their original forms.

To solve this problem, there are two possible solutions that can be used: (1) undo the transformation to get the original operation, or (2) keep the original form of each stored operation. While the first solution is elegant and seemingly does not need any additional memory requirement, it is not feasible to implement in practice. To undo the transformation, the site has to record the state of the site when the operation was generated. It has to be able to shift the operation to its original position in the history (when it was first generated) and it has to transform the operation to its original form. It needs extra memory space to store the original position of the operation in the history, but also requires extra processing power to do the transformation. The second solution seems to be more expensive in terms of resource. However, if a site keeps the original form of each stored operation, the site does not need to transform an operation to get its original form. By adding extra information in the history, the site does not need to use extra processing power to do the transformation.

Depending on the type of application, storing the original form of operation might mean storing the whole operation command and the parameters or simply storing the original values of the parameters (if the operation transformation does not change the operation command).

In the case of missing operations, the site that requires the operations sends requests to the originator site. Thus, each site needs only to keep the original form of the locally generated operations. It does not need to worry about maintaining the original form of remote operations. This does not add much to the storage space consumption. However, depending on the implementation, sites may be able to request operations from other sites other than the originator site (i.e. the originator site is disconnected for indefinite period). In this case, each site needs to maintain the original form of all operations, locally generated and remote operations. The next section, section 5.5.2 discusses this issue.

5.5.2. Requesting operations from a site other than the originator

When S_i misses some operation, it sends an operation request to the originator site S_j to resend the missing operations. If S_i and S_j are not connected (i.e. either one of the sites is disconnected or unreachable), the requested operation cannot be sent. The request might not arrive at S_j, thus S_j does not know if there is an operation request, or the operations that are sent might not arrive at S_i. In either case, S_i will not receive the requested operations at all. If they are disconnected for an indefinite period, S_i might not receive the missing operations at all and S_i will not be able to continue processing other future remote operations. This obviously reduces the availability of the data and the ability of sites to collaborate.

This problem can be solved by allowing S_i to send requests to any other sites, other than S_j, that have the requested operations. When S_i receives an operation from another site, say S_k, S_i is able to determine whether S_k has already executed the operations that are requested. If S_k has executed those operations, S_i knows that S_k has the operations and S_i could send the request to S_k instead of S_j.

There are various way of implementing this scheme. S_i could send the operation request to S_j first and wait for S_j to reply. If S_i has not received the operations by a certain timeout period, then S_i can send the request to other sites (e.g. S_k). Another possible strategy is that S_i could send the request arbitrarily to any site as long as S_i knows that the site has the requested operations.

This solution, however, provides greater flexibility and availability at the cost of resources. Sites can request operations from any other connected sites, but it requires all sites to keep the original form of all operations, the locally generated and the remote operations. Hence, each site needs extra storage space to store the original form of all operations. If the connectivity is relatively good, then this solution will only add to the resource consumption without giving a significant benefit. However, if the connectivity is relatively poor, then this solution will be a good alternative to handle missing operations.

5.5.3. Remote Operation Queue

Each time a remote operation is received by a particular site, that site will execute procedure *rcv_remote_op*. The remote operation will go through a checking process before it is executed. One important thing to note is that to maintain consistency, the remote operation will only be executed if it is causally ready (i.e. there are no other operations, which precede it, that have not been executed by the site). Until it is causally ready, it waits in a remote operation queue.

The remote operation queue holds all remote operations that have been received but are not yet ready to be executed. It is highly probable that when another remote operation arrives at a site, it has already had some operations in its remote operation queue. This will affect the way the missing operations are determined. The basic idea is to detect missing operations by comparing the state map of the received operation with the state map of the recipient site. However, the state map of a site only shows the operations it has executed, not the operations it has received that are still in the remote operation queue. Therefore, if a site wants to check for missing operations, it also has to check its remote operation queue to determine whether it has received the operation that it suspects to be missing.

Consider the following scenario where there are three sites collaborating: S_i, S_j and S_k. Let $op_{i,n}$, $op_{j,n}$, and $op_{k,n}$ be the n-th operation generated by S_i, S_j and S_k respectively. S_k has its state map $M_{S_k} = \{S_i \Rightarrow 5, S_j \Rightarrow 4, S_k \Rightarrow 6\}$ and S_i generates an operation $op_{i,10}$ where $M_{op_{i,7}} = \{S_i \Rightarrow 9, S_j \Rightarrow 9, S_k \Rightarrow 5\}$. This means that by the time $op_{i,10}$ was generated, S_i has already executed 6 operations generated by S_i, 9 operations generated by S_j and 5 operations generated by S_k. When $op_{i,7}$ arrives at S_k, S_k notices that S_i has already generated 9 operations, while S_k has only received 5, thus S_k knows that it has missed some operations from S_i.

Using the proposed algorithm, S_k will send a request to S_i to ask for $op_{i,6}$, $op_{i,7}$, $op_{i,8}$, and $op_{i,9}$. S_i might have already received some of those requested operations previously, but because of the causal precedence, it cannot execute them as yet and they are stored in a remote operation queue. Assume $op_{i,7}$ and $op_{i,8}$ have already been received and are currently stored in a remote operation queue. This means S_k will only need to request $op_{i,6}$ and $op_{i,9}$. Therefore, the site's state map is not enough to determine the operations to be requested. The site also needs to search its remote operation queue to see whether some of the requested operations have been received and are currently stored in the remote operation queue.

5.5.4. Duplicate Operation Requests

If a site has missed some operations, it will send an operation request to ask for those operations to be resent. Duplicate operation requests might happen if the site sends the same operation request to the originator site while it is waiting for the response of the first request.

Let say site S_i and S_j are currently participating in a collaboration session. Site S_i generates an operation op_i and broadcasts it to all other participants. Upon receiving op_i, S_j realises that it has missed some operations and it sends the request to S_i requesting operations that it has missed up until operation op_i excluding op_i. When S_i receives the request, it will respond by sending the requested operations to S_j. If, however, S_i generates another operation op_j before responding to the request and sends op_j to S_j, S_j will detect that it missed some operations (which is the same set of operations that it has requested from S_i) and it will send the request to S_i. Therefore, S_j will end up sending two requests for the same set of missing operations.

This issue can be handled by introducing the use of a timeout between the subsequent requests. After S_j sends the first operation request, S_j waits for a certain timeout period before sending another request for the same operations. After the timeout, if S_j has not received some or all operations that it requested, S_j will send another operation request.

5.6. Performance Evaluation

The algorithm has been evaluated by simulation on varying numbers of sites, numbers of generated operations, network delays (the time it takes for an operation to arrive at the destination) and disconnection rates (the probability the operation not arriving at the destination). The simulation aims to evaluate the resource usage of the introduced functions (handling missing operation and late joining) by measuring the portion of processing time used for handling membership events.

The simulation is done using Java and a PC which simulates a number of sites, generating and receiving operations with various simulated network delays and disconnection rates. A simulation in a real mobile network environment is not necessary since the simulation does not aim to represent the actual time taken to run those functions nor the actual disconnection rates since these vary depending on the platforms and network configurations, the different implementations, users' behaviour, and users' mobility.

However, the simulation does show the resource usage of the additional functions relative to the existing consistency management algorithm without the membership functions and the simulation gives the trend of the expected results on various disconnection rates.

Firstly, the number of lost operations, the number of operation requests, and the total number of messages required to completely send all operations to all destinations are counted under various disconnection rates in order to determine the ratio of each of them against the total number of operations that have to be broadcast to all destinations. The number of lost operations is the number of operations that are sent by a site and lost during transmission, whereas the number of operation requests is the number of requests sent by a site because it misses some operations. For example, if there are 4 sites collaborating and 20 local operations generated at each site, one site has to send each of the 20 operations to 3 participants, which means one site has to send 60 operations in total. If, for example, 20 out of these 60 sent operations were lost, the lost operations ratio would be 33%.

Figure 5-16 shows the ratio of lost operations over the total number of operations sent. As expected, as the disconnection rate increases, the number of lost operations increases *exponentially*. This is because some operation requests and resent operations might get lost during transmission causing the receiver site to send additional operation requests and thus the number of lost operations increases exponentially. When a sender site sends operations some of them might be lost during transmission. The receiver site will then send operation requests to the originator when the operations are lost during transmission and the originator will send the requested operations to the receiver site.

Some of these operations might also be lost during transmission, thus additional operation requests are sent to the originator. This goes on until all sent operations arrived at the destinations, causing the number of lost operations to increase exponentially.

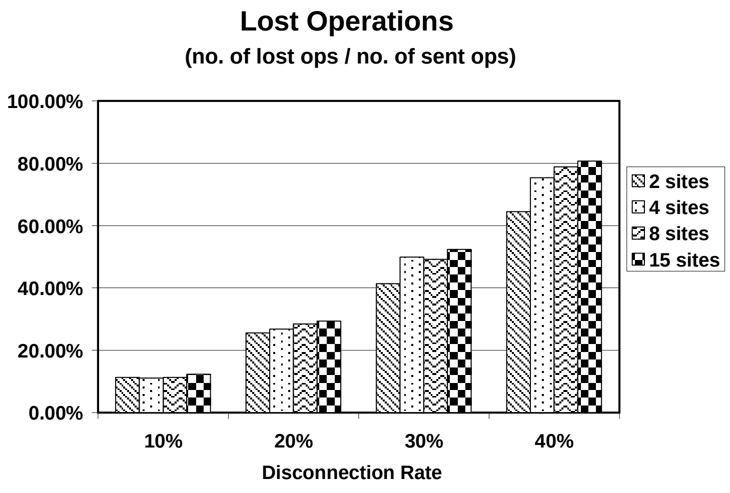


Figure 5-16 Lost operations

Figure 5-17 shows that, again as expected, the number of operation requests increases as the disconnection rate increases. For example, when two sites are participating and the disconnection rate is 30%, the number of operation requests is up to 50% of the number of sent operations. Figure 5-18 shows the correlation between the number of sent messages and the disconnection rate. The number of sent messages is the average number of messages being transmitted by a site in order to completely receive all operations. This number includes the number of operations sent in the first place, the number of operations being re-sent (due to disconnections), and the number of operation requests. When the disconnection rate is 30%, the number of total messages transmitted reaches up to 200% of the number of operations needed to be transmitted.

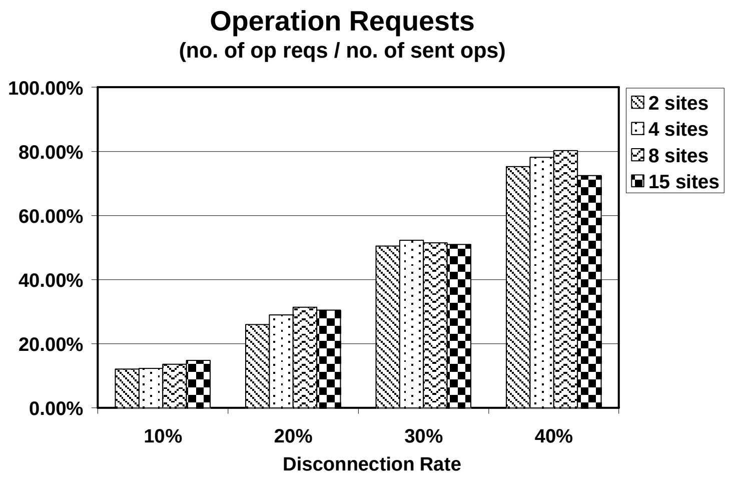


Figure 5-17 Operation requests

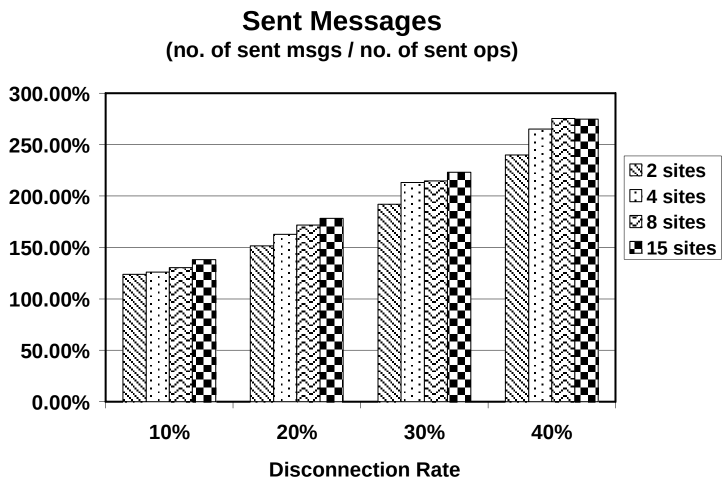


Figure 5-18 Sent messages

Secondly, in order to add membership functionality, the proposed algorithm requires additional overhead in terms of processing time. Before actually executing the remote operation, a site needs extra processing time that comprises: (1) time to check whether or not there is any new site, (2) time to check whether or not the incoming operation has been received or executed, (3) time to check if there are any missing operations, (4) time to construct an operation request for any missing operations if necessary, and (5) time to put the operation request to the outgoing message queue ready to be sent. Figure 5-19 shows the proportion of total time taken to handle the membership management processing overhead.

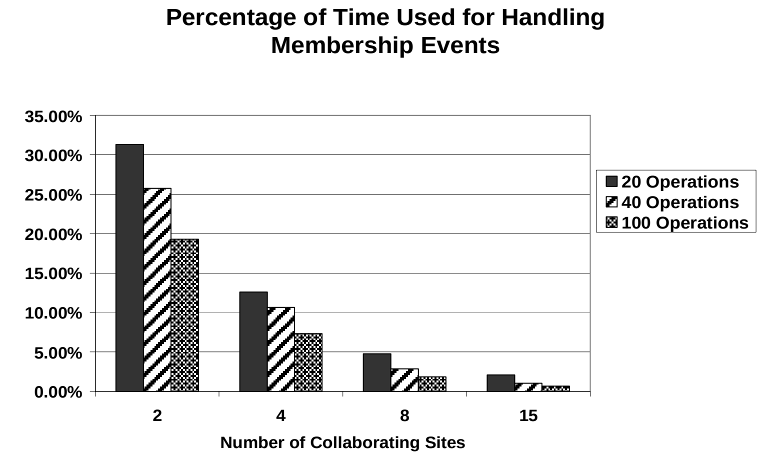
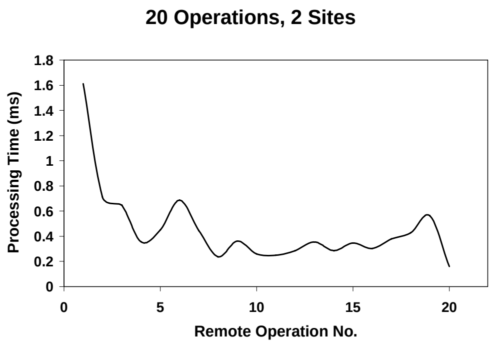


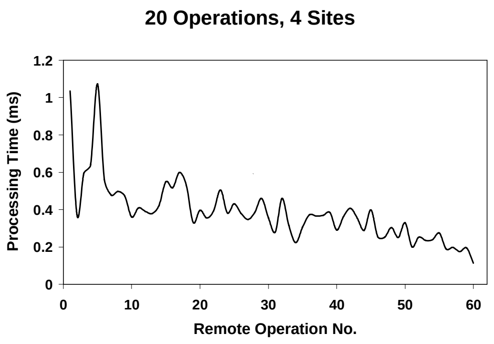
Figure 5-19 Processing time required to handle membership events

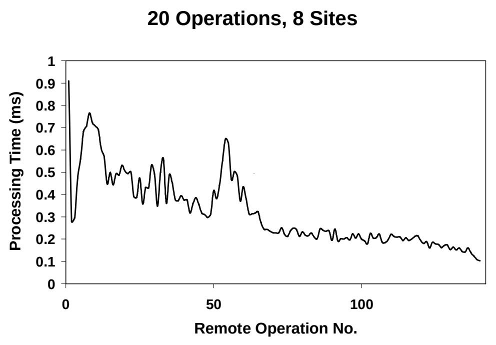
When there are two sites collaborating, around 25-35% of the time is used for handling membership while the remaining 65-75% is used to actually execute operations. When there are more than 8 sites collaborating, the site requires less than 5% of the time to handle membership events. As the number of sites increases, the portion of time used for performing membership functions declines exponentially. This is caused by the fact that the more sites, the more time required to execute the operation (since executing an operation involves tracing the history to find concurrent operations and to do the necessary operational transformations).

The experiment shows that the time used by the proposed algorithm to perform the membership functions is constant regardless of the number of participating sites or the number of generated operations. In the simulation using a PC with 1.8GHz of CPU clock and 512MB of RAM, the time to execute a remote operation ranges from 1.42ms for two sites to around 45ms for 15 sites, whereas the time used for these membership functions is relatively constant around 0.23ms to 0.5ms.

Figure 5-20 and Figure 5-21 show that the time to do this processing is relatively small at around 1 to 2 ms for each remote operation, and the processing time does not increase over time. Therefore, the proposed algorithm allows each site to handle the various membership events with very little processing power.







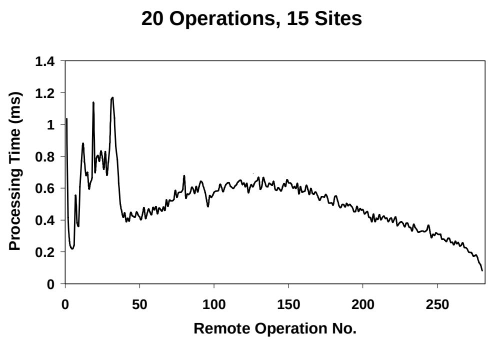
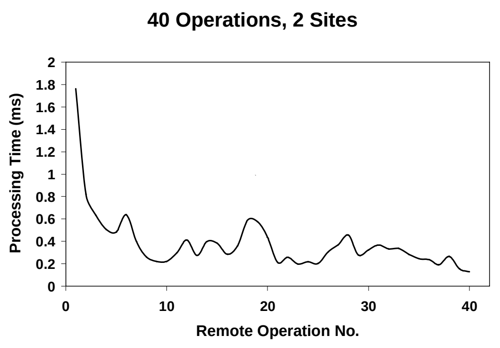
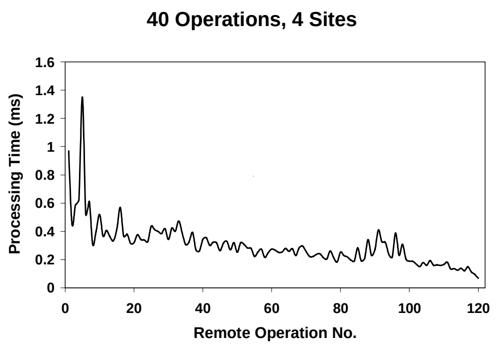
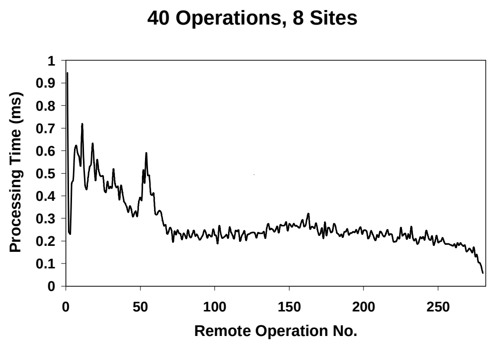


Figure 5-20 Processing time for 20 operations







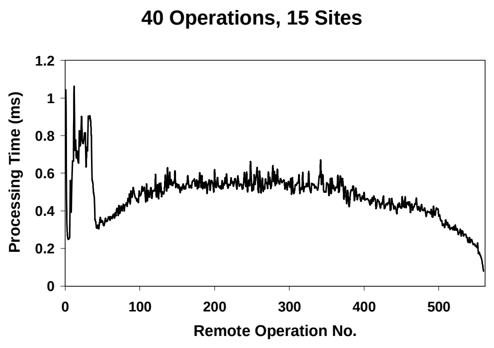


Figure 5-21 Processing time for 40 operations

5.7. Conclusion

This chapter has discussed the need to handle membership events in a fully distributed real time mobile collaboration system. Frequent disconnections and late joins cause fluctuations in collaboration session membership. If this is not handled correctly, it may result in inconsistent document replicas among the collaborating sites.

An algorithm that handles these two major membership events in mobile networks has been proposed and it is integrated with the proposed consistency management algorithm (presented in Chapter 3) so as to ensure the consistency among document replicas. The proposed algorithm detects if there are any missing operations and requests the originator site should resend to allow a new joining site to blend in the session smoothly and be able to participate in the session. The algorithm incorporates a novel technique, called a state map, as a replacement of the state vector technique to allow concurrent late joins to be handled correctly. Furthermore, the algorithm does not require new participants to explicitly inform other participants that they have joined the session. The experiments show that the proposed algorithm provides each collaborating site with the capability of handling various membership events while maintaining document consistency with a relatively small overhead (5%-25% of the total processing time).

Future work may include an investigation of how to optimize the algorithm in terms of minimizing the number of sent messages in the event of disconnections. Some possible strategies may include sending the missing operations in one message and dividing the document into partitions so as to reduce the overall number of messages needed to be transferred. Group membership events might also be analysed to identify patterns of behaviour that can be incorporated into the algorithm to help the algorithm anticipate membership events and to improve the overall performance.

Chapter 6

Document Partitioning

6.1. Introduction

Methods of dealing with consistency management, conflict management and membership events in mobile environments have been proposed in previous chapters. Like other existing algorithms that support mobile real-time collaborative editing [34, 46, 115, 126, 132, 142], they require each device (site) to maintain a local replica, the state of the local replica (site state map) and store a history of operations to be used to correctly process and execute concurrent remote operations.

Mobile devices have very limited storage space, thus if the document is large, they may not be able to hold the complete document replica. In replicated real time collaboration each device has to store the document replica with its state and the operations history to fully function. The size of the operation history may grow indefinitely during the collaboration session. History trimming can be used to minimise memory resources.

However, depending on the concurrency level of the generated operations, the history size may not be able to be trimmed to zero size. Furthermore, mobile devices have limited interface (screen) size such that even though the document replica could be stored completely in the device storage, a user might not be able to see the whole document at once. Therefore, a user might choose to work only on a particular part (section) of the document.

For example, consider a group of architects working on a draft of the design of a health clinic. The draft may be too large to be completely stored or replicated in a mobile device, and some of them may choose to work on some specific parts of the clinic, leaving other parts to other team members. When *Alice* is working on the pathology room, she might not need to know the immediate updates on the surgery room. When *Bob* is working on the reception area, he might not need to know the immediate updates on the consultation rooms. Another example would be a group of authors working on a book. An author may choose to work on only some chapters, while another may choose to work on other chapters. Note that one chapter may be updated by more than one user at a time.

By allowing a document to be partitioned into several sections, users can work only on desired sections. This not only provides flexibility for users, it also reduces resource consumption. When *Alice* is working only on a specific section of the document and is not interested in updates on other sections, she does not need to receive updates on other sections, thus reducing bandwidth consumption (since the sites working on other sections do not need to send updates to *Alice*) and reduces processing consumption (*Alice*'s site does not need to process any unnecessary incoming updates).

This chapter presents an algorithm to support document partitioning in a replicated architecture that maintains consistency and provides flexible membership management. The collaborative document is divided into application-defined sections and users can choose to work on one or more selected sections. A generated operation may affect one or more sections and it is broadcast to all sites that work on the affected section(s). The algorithm ensures that the sections are consistent across all sites, and users receive only operations that they are interested in, i.e. operations that affect sections that they are working on. The algorithm also utilises a document index, maintained by all sites, to ensure that sections are laid out correctly in the complete document. The algorithm allows users to create new sections whenever necessary and provides mechanism for them to switch from one section to another section as needed.

The rest of the paper is structured as follows Section 6.2 explains the rationale and the problem of document partitioning. Section 6.3 discusses related work in document segmentation and adaptation into mobile environments. Section 6.4 outlines the model of the document partitioning problem. Section 6.5 presents the proposed algorithm. Section 6.6 presents the results of the performance analysis. Finally, section 6.7 concludes the paper and outlines some possible future work.

6.2. Document Partitioning Problem

A collaboration session starts when two or more users decide to produce a document collaboratively. In a replicated mobile environment, a user initiates the session and other users will join the session. When a user joins a session, s/he will receive the up-to-date document, document state and other necessary data to ensure that s/he will be able to participate in the collaboration session as outlined in Chapter 5. Throughout the session, the document consistency algorithm presented in Chapter 3 is used to ensure that the consistency of the document replicas is maintained.

The document consistency algorithm, however, requires additional processing power and storage space. In terms of storage space requirement, the consistency algorithm requires each site to maintain an operation history. The operation history stores all executed operations (local and remote) and its size increases indefinitely as the collaboration session goes on. In order to minimise the storage requirement, document partitioning allows users to work only on some sections, therefore they do not need to receive operations on other sections, and thus reducing the size of the history. It can be argued that the history trimming technique [34, 132] alone can be used to minimise the history size by garbage collecting the operations that will no longer be used for transformation purpose. However, due to network delays, operations concurrency and frequent disconnections, the history may not be able to be cleaned entirely. The length of the delay, the level of operations concurrency and the frequency of disconnections correlates positively with the size of the history [34]. By combining document partitioning and history trimming, the history size may be further reduced.

In addition to reducing the storage requirement, document partitioning may also reduce the power consumption. Consistency management algorithms consume processing power to process incoming remote operations. GOT [137] works by undoing the already executed operations, executing the remote operation and re-executing the undone operations; GOTO [131] and SOCT2 [126] attempt to use backward transformation to ensure the remote operation are transformed at its original state; SOCT3 [142] separates the history into two sequences, transforming and executing the remote operation, placing the remote operation in the history and reorganising the history; the algorithm proposed in Chapter 3 uses a total ordering algorithm to ensure the operations are ordered consistently across all sites. Regardless of the way the algorithm works, all of them consume processing power to process an incoming remote operation. By allowing users to work only on some sections, updates are sent only to interested users. Therefore, users do not need to receive and consequently process unnecessary operations thus reducing processing power consumption.

For example, three authors - Alice, Bob and Cameron - are currently participating in a collaboration session. As the document increases in size, *Alice* may choose to work only on the first and second chapters. Another team member, *Bob*, chooses to work on the second and the fourth chapter. Meanwhile, *Cameron*, another team member, is working on all chapters (chapter one to four). Modifications on chapter four by *Bob* might not interest

Alice, neither might *Bob* be interested in modifications of chapter one by *Alice*. To reduce unnecessary resource consumption (network bandwidth, storage space and processing power), updates on chapter one by *Alice* need to be sent only to *Cameron*, updates on chapter four by *Bob* need to be sent only to *Cameron*, updates on chapter two need to be sent to all members, and updates on chapter three by *Cameron* need not be broadcast to anyone. Document partitioning allows each user to be active on his/her selected parts and not to receive unnecessary updates. Therefore, there is a need for a document partitioning strategy that allows users to work on desired parts and at the same time minimises resource consumption.

Document partitioning, however, presents some challenges in real time collaborative editing in mobile replicated architecture, as follows.

1. ***Section Boundaries***. Boundaries of sections must be clearly defined and agreed by all users. In an object based document, for example, if an object belongs to a particular section at a site, that same object must belong to the same section at all other sites. Furthermore, the objects in one section must comprise a meaningful piece of work or artefact, such as a chapter of a text document, a page of a class diagram, and a worksheet of a spreadsheet document.
2. ***Section Membership***. If a site chooses to work on a particular section, that site must receive all updates generated by all other sites on the section. This implies that each user has to know all users who are currently working on the same section.

Consequently, a site should not receive operations that do not affect sections that the site is currently active on.

3. ***Section Creation***. During a session, a document section may increase in size and there may be a need to either split the section into two sections or to add a new section into the document. To ensure the collaboration continues correctly and smoothly, when a new section is created, every user must agree on its boundaries and, if applicable, its initial components (objects).
4. ***Joining a Section***. During a session, a user on site *S1* may choose to join a section that s/he is not currently active on. In order for *S1* to be able to join a section, say section *A*, *S1* must first contact a site that is currently active on section *A* to receive the up to date content of section *A* with all necessary data. Eventually, after joining section *A*, all sites that are currently active on section *A* must know that site *S1* has joined, thus will send further updates on section *A* to site *S1*.
5. ***Leaving a Section.*** After working on a section, say section *A*, a user on site *S1* can choose to stop working on that section. In this case, other users that are working on section *A* must eventually know that *S1* is no longer active on section *A* and therefore will not send any updates on section *A* to site *S1*,

The proposed document partitioning algorithm addresses the challenges above in order to provide flexibility for users to work on their desired sections while still maintaining the consistency of the document replicas.

6.3. Related work

Much work has been done in porting documents into mobile devices due to the limitations in mobile network environments [81, 82, 95, 106]. Adaptation and document decomposition are two common approaches to port the document content into lower capacity and bandwidth limited network environments.

Duplex [106] uses two concepts: *document decomposition* and *kernel existence*. The document decomposition is similar to the proposed strategy presented in this chapter. However, Duplex assumes the document segments are totally independent. The kernel holds the central repository of the document parts and the kernel is replicated for responsiveness and availability. Duplex, however, works only in asynchronous (non-real-time) collaboration where users need to download the document parts from the kernel if they want to edit certain segments and then commit the changes back to the kernel once they are done with the changes.

Puppeteer [82], a system for adapting applications and content for the mobile environment, took advantage of the exported APIs of component-based applications to customize the behaviour of applications based on their environment *without* modifying the source code of the original application. Puppeteer transcodes the document components/segments into a partial fidelity version of the document components to allow faster download of the document from its native store. However, Puppeteer allows only a read-only adaptation. Odyssey [102], similar to Puppeteer, also adapts application data to the current state of the network connection. However, unlike Puppeteer, which uses public APIs of applications to manipulate that application's data files, Odyssey requires applications to be customized to support its implementation scheme.

Alliance [38] was designed as a cooperative application that allows several users, distributed on a network, to work on shared structured documents. Documents are automatically divided into variable sized fragments, which are the sharing units. A fragment is a continuous part of a document. For maintaining document consistency in the LAN version, only one copy of each document is stored on disk, whatever the number of users working on it and the sites from which the users work; but several files are used for storing a single document, one file for each fragment. These files reside on a volume that is accessible from all workstations involved in the application. This is achieved by using Network File System (NFS) [4] which presents all remote files as if they were local. Each user plays a certain role on each document fragment. However, to maintain document consistency, each document fragment can only be modified by one person (who has the *writer* role) at any given time; thus it does not support concurrency. When a fragment is updated by a user acting as a writer, this fragment is not automatically sent to all sites working on the document. Only a short message is sent: it informs the remote sites that an updated version of the fragment is available. With the replication policy presented above, communication between sites is needed only to transmit these short messages, to transfer updated copies to the sites which ask for it, and to get remote user lists.

CoFi [81] extends Puppeteer by implementing adaptation-aware editing and progressive update propagation mechanisms that enable document authoring and collaborative work over bandwidth-limited links. Adaptation-aware editing enables editing adapted documents by differentiating between modifications made by the user and those that result from adaptation. Progressive update propagation reduces the time and the resources required to propagate components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of those modifications. CoFi, however, works in semi-centralised architecture and it does not support synchronous (real-time) collaboration. CoFi requires the clients to propagate the updates to a remote proxy, and consequently fetch updated document content from remote proxy as well. The remote proxy will then synchronise the document with the native store so that another remote proxy can fetch the updated document contents.

Some work has also been done to allow mobile clients to browse web pages while efficiently using the limited bandwidth. One such example is the work by Yau et al. [152] which proposed a mechanism that allows a web client to explore the more content-bearing portion of a web document before deciding to download the whole document. The user is presented with the main document content before the supplementary information is presented (downloaded). The mechanism aims to prevent the web client from downloading unnecessary and/or irrelevant web content, thus reducing unnecessary bandwidth consumption. This mechanism, however, is only applicable to the web environment which by nature is very different from a real-time collaboration environment.

Veiga et al. [141] introduced a notion of semantic-chunks, where documents are segmented into portions of files that have content-derived (instead of offset-derived) boundaries. Application-based semantic-chunk borders may be defined as sections, paragraphs, sentences in text documents, cell areas in spreadsheets, objects and geometry in CAD tools, functions and declaration zones in programming source code editing and so on. The proposed semantic-chunks middleware provides transparency to the users (i.e. users are not aware of the chunks) and allows the existing office application to be fragmented without significantly modifying the off-the-shelf applications. Semantic-chunks middleware, however, adopts a very simple consistency enforcement strategy where each update is comprised of a set of modified semantic-chunks and is propagated either implicitly, whenever two peers meet with neighbouring devices, or explicitly, whenever two or more peers meet and the file owner broadcasts a new update to explicitly overwrite all other replicas. This strategy, however, works only when the concurrent editing on the same chunk is rare, therefore an update by a user or the file owner can simply overwrite the chunk of other peers. Furthermore, although they mentioned the use of version-maps, they do not discuss how version-maps are used with regards to maintaining consistency.

TreeOPT [67], as mentioned in section 3.2.1 and 3.3.3, divides documents into multiple granularity levels which allows independent document modification on separate nodes. TreeOPT however is somewhat a multi-level linear representation of the document and each node, although independent at its level, is not completely independent as they belong to the same higher-level node (for example two words in a paragraph are dependent on each other at the paragraph level). Therefore, treeOPT does not allow the document to be partitioned.

While much work has been done in the area of adaptation and document decomposition, none of it directly supports document partitioning in synchronous (real-time) mobile collaborative editing in a fully replicated architecture.

6.4. The Model

A real time mobile groupware system consists of a set of participant sites ($S = \{S_1, S_2 \dots S_N\}$, N = the number of current participants) connected by a wireless network. A site corresponds to a mobile device and there is one device per user. Each site holds a document replica (R_i = replica of site S_i). A document consists of document objects ($R_i = \{O_1, O_2, \dots, O_k\}$, k = the number of objects in the document). For example, an ER diagram consists of several entities and relationships as the objects; and a home designer draft consists of furniture, lightings, and decorations as the objects.

The document is divided into sections C_j. A replica is complete if it contains all sections ($R_i = C_1 \cup C_2 \cup \dots \cup C_t$, where t is the number of sections in the document). A site may choose to work on one or more sections at one time. The ability of each site to choose whether or not to work on a particular section determines the status of each site on each section as follows.

- Site S_1 is *active* on section C_1 if a user at site S_1 is able to update section C_1 and receives updates on section C_1 from other sites.
- On the contrary, site S_1 is *passive* on section C_1, if a user at site S_1 decides not to make any further update on section C_1, *and* not to receive any updates on section C_1 from other sites.

Each section consists of several objects ($C_j = \{O_1, O_2, \ldots, O_x\}$) . An object may belong to more than one section, especially if the object links two objects on two separate sections. For an example, in a class diagram, an association line that links two classes on two separate sections will belong to the two sections so that users can see the association line even though s/he is working on only one of the sections.

Whenever a user updates the document replica, a local operation *op* is generated to realise the user's intention. In an unpartitioned document, the generated operation is broadcast to all other sites since all sites are working on the whole document (complete replicas). In a partitioned document, however, the generated operation needs to be broadcast only to sites that are active on the same section where the operation has been generated (sites that are *active* on the same section). Therefore, each section must maintain a membership list, SC_{S_i,C_x} = the list of sites that are active on section C_x as known by site S_i. This list tells site S_i which sites each generated operation must be sent to. Even though an operation usually affects only one section, some operations may affect two or more sections. Suppose *Alice* is working on two sections, C_1 and C_2, and *Bob* is working on another two sections C_2 and C_3. If *Alice* creates an object that links an object in C_1 and another object in C_2, this operation has to be sent to users that work on C_1 and/or C_2 causing *Bob* to receive this operation even though *Bob* is not working on C_1. Consequently, each section must maintain its own history of operations that each local operation and remote operation can be processed according to the section it belongs to, and stored in the section's operations history (H_{S_i,C_x} = operations history of section C_x in site S_i).

To maintain causality, each operation bears a state map M_{op_i} to signify the state of the document when the operation was generated. In an unpartitioned document, the state map signifies the state of the whole document (how many operations have been executed from each sites). The state map signifies when it was generated and the state map technique is useful to determine whether an operation has been generated after, before, or concurrent to another operation. Note that the state map technique works in a similar way to the state vector technique. Instead of using an integer index, a state map uses a unique key to indicate which site each element of the state map is associated with (refer to section 5.4.2). Traditionally, when *op* is generated by S_{op}, the operation state map is a map of size N, $M_{op} = \{key(S_1) \Rightarrow val(S_1), key(S_2) \Rightarrow val(S_2), \ldots, key(S_N) \Rightarrow val(S_N)\}$, where $key(S_i)$ is the key associated with site S_i and $val(S_1)$ is the number of operations generated by site S_i that have been executed by S_{op} at the time *op* was executed. The state map is then used to determine whether an incoming remote operation can be executed or it has to wait for other operations that precede it.

However, if this state map technique is applied to a partitioned document, a remote operation has to wait for other preceding operations, even though the preceding operations might not be directly related with the remote operation or the preceding operations are done at another part of the document. Therefore, in a partitioned document the state map should signify the state of the section where the operation is generated, instead of the state of the whole document, to remove dependency (causality) between operations on different sections. If the state of the whole document is used, whenever a user receives a remote operation, it has to wait for, receive and execute all operations that causally precede it. Therefore, if one or more of the preceding operations belong to another section, they will not be sent to the user and thus the user will not have to wait for those operations indefinitely. By using a section state map, each section can operate independently and each section has its own set of operations and therefore its own operation history. The section state map is represented by M_{S_i,C_x} with each of the element of M_{S_i,C_x} represents the number of operations, generated by another site targeting C_x, that have been executed by site S_i on its own section C_x.

If an operation belongs to more than one section, it will bear multiple state maps, one from each section that it affects. This is necessary to ensure that when an operation arrives at another site, the site can process the operation correctly even though the site has only one of the affected sections. Therefore, every operation will be time-stamped with a state map list containing the state maps of the affected sections, $ML_{op} = \{ML_{op}[1], ML_{op}[2], \ldots ML_{op}[k]\}$, where k is the number of the affected sections. Each entry of the ML_{op} is a state vector of an affected section, $ML_{op}[i] = [x, M_{op,C_x}]$, where x is the id of the affected section (C_x) and M_{op,C_x} is the state map of the section C_x in site S_{op} when *op* was generated. As an example, suppose site S_1 generates *op* and *op* affects two sections: C_1 and C_3. If the state maps of the sections are $M_{S_1,C_1} = \{S_1 \Rightarrow 3, S_2 \Rightarrow 4 , S_3 \Rightarrow 4\}$ and $M_{S_1,C_3} = \{S_1 \Rightarrow 6, S_2 \Rightarrow 4, S_3 \Rightarrow 9\}$ at the time of the local operation generation, *op* will bear the state vector list $ML_{op} = \{[1, \{S_1 \Rightarrow 3, S_2 \Rightarrow 4 , S_3 \Rightarrow 4\}], [3, \{S_1 \Rightarrow 6, S_2 \Rightarrow 4, S_3 \Rightarrow 9\}]\}$.

Each section maintains its own operation history; therefore each section is responsible for doing its own garbage collection to remove operations that are no longer needed. In order to do this, each section must maintain a record on which sites have executed which operations on that section, therefore each section must maintain a state map table (MT_{S_j,C_x}) to be used for history trimming. The garbage collection algorithm using the state map technique is presented in detail in section 5.4.2.

During the course of the collaboration session, there might be a need to create a new section. For example, an author decides to create a new chapter, an architect decides to create a new room or a software designer decides to create new classes in a new section. Each time a section is created, all users must be notified. This is to ensure that if more than one user is concurrently creating new sections, possible conflicts are resolved, and the new sections and their relationships with other sections must be correct and consistent in all document replicas.

A document index and/or layout may be necessary to construct the complete document when a user has all the sections. In a text document with chapters as the sections, a document index is used to provide the order of the chapters. In a 2D document, a document layout is used to construct the complete draft when all sections are available. For simplicity, this chapter assumes a single dimensional document index, where sections (chapters) are ordered linearly with one preceding the other. Adding a new chapter means creating a new section and adding the section index into the document index at the appropriate position. A document index at site S_i is represented by DI_{S_i} = $\{DI_{S_i}[1], DI_{S_i}[2], \ldots, DI_{S_i}[m]\}$ where m is the current total number of the sections. Each element of DI_{S_i}, $DI_{S_i}[j]$, is a document section, C_x, and x is not necessarily equal to j since the sections are dynamically created and inserted in various positions.

6.5. Proposed Algorithm

During a collaboration process, a site typically goes through the following phases: local operation generation, operation broadcast, and remote operation reception and execution. In a collaboration that allows document partitioning, a site may go through some other phases such as adding an active section, removing an active section and creating a new section. The following are notations to be used in the proposed document partitioning algorithm:

- S_i = site with id i.
- C_x = section with id x.
- R_i = document replica at site S_i.

- N = the number of current participants.
- m = the number of current sections in the document
- SC_{S_i,C_x} = the list of sites that are active on section C_x as known by site S_i.
- H_{S_i,C_x} = operations history of section C_x in site S_i.
- M_{S_i,C_x} = the state map of section C_x in site S_i, with each of the elements of M_{S_i,C_x} represents the number of operations, generated by another site targeting C_x, that have been executed by site S_i on its own section C_x.
- MT_{S_i,C_x} = the state map table of section C_x in site S_i. $MT_i[j][key(S_k)]$ is the number of operations generated from site S_k that have been executed by site S_j as known by site S_i.
- ML_{op} = the state map list of operation op.
- ML_{op} = $\{ML_{op}[1], ML_{op}[2], \ldots ML_{op}[k]\}$, where k is the number of the affected sections.
- $ML_{op}[i]$ = $[x, M_{op,C_x}]$, where x is the id of the affected section (C_x) and M_{op,C_x} is the state map of the section C_x in site S_{op} when op was generated.
- DI_{S_i} = document index as known by site S_i.
- $DI_{S_i} = \{DI_{S_i}[1], DI_{S_i}[2], \ldots, DI_{S_i}[m]\}$ where m is the current total number of sections.
- $DI_{S_i}[j]$ is a document section, C_x and x is not necessarily equal to j.

6.5.1. Local Operation Generation

Every time a local operation is generated, it is immediately executed locally, and broadcast to other sites. The operation will bear one or more timestamps (state maps) depending on how many sections it affects. For each affected section, the operation will bear the section's state map. The operation is then broadcast to all sites working on the affected sections and is stored in the operation history of the affected sections. Figure 6-1 presents the procedure *gen_local_op(op)* that is invoked when a local operation is generated.

```
void gen_local_op(op) {
        //Assume op is generated at site $S_i$
        $ML_{op} = \{\};$
        $S_{dest} = \{\};$ // $S_{dest}$ is the set of sites to send the op to
        for each affected sections $C_x$ {
                //Adding the section state map
                //to the operation's state map list
                $ML_{op} = ML_{op} + [x, M_{S_i,C_x}];$

                //Update the section state map
                $M_{S_i,C_x}[key(S_i)] = M_{S_i,C_x}[key(S_i)] + 1;$

                //Adding the operation into the
                //section's operation history
                $H_{S_i,C_x} = H_{S_i,C_x} + \{op\};$

                //Gathering the sites to send the op to
                $S_{dest} = S_{dest} + SC_{S_i,C_x};$
        }

        send $<op, LC_{op}, ML_{op}, S_{op}>$ to all sites in $S_{dest}$;
}
```

Figure 6-1 Local operation generation

When the generated local operation affects more than one section, the originator site may or may not be active on all the affected sections. If a user creates an object connecting two objects from different sections, s/he has to have all the affected sections on his/her device to ensure s/he knows the current state of each section before creating the connecting object. For example, when *Alice* wants to create an object *X* that links two objects *Y* and *Z* that belong to different sections, say C_1 and C_2, she has to have both sections to ensure that *Y* and *Z* still exist.

However, if a user modifies an object that connects two objects from different sections, s/he does not necessarily need to have all the affected sections. For example, if *Alice* wants to delete object *X*, she does not necessarily need to be active on both sections. If she is active on both sections, she can easily determine the participants that should receive the operation from the section participant lists. However, if she is not active on C_1, she will not have the participant list of the section C_1.

There are two options to address this issue. The first option is to allow users to modify an object that connects two sections *only if* they have both sections active on their device. Therefore the connecting object appears as read-only object if a user does not have both sections. The second option is to allow users to modify an object that connects two sections regardless. If they modify an object that connects two sections and they don't have one of the sections, then the device needs to build up a list of recipients which is the union of the participants that are active on both sections. For the section that the user is active on, the list of participants is up to date. However, for the section that the user is not active on, s/he has to retrieve the list of participants from another user that is currently active on that section. The user can broadcast a request of the list of participant and another site can reply with the appropriate list. Then from the combined list of participants, the user sends the operation to all affected sites.

The first option reduces complexity and is less expensive as it does not need to request and receive the participant list from another site. While it reduces bandwidth consumption, it restricts users from updating connecting objects in such case. However, by restricting such updates, it ensures that the user does not make an uninformed change on the object. Since the user does not have one of the connected sections, it is probably wise to not modify the object since the other section may have been updated without the user being aware of it.

Therefore, it would be up to the user to select which option to use. If the user wants to reduce resource consumption or the user does not want to risk making an uninformed update, then the first option should be used. If, however, resource consumption is not an issue or the user wants the flexibility to update all objects despite the risk of an uninformed update, then the second option could be used.

6.5.2. Remote Operation Reception

A remote operation may affect one or more sections, therefore a remote operation will eventually be executed at all affected sections. To maintain causality, in each affected section the remote operation has to wait for all preceding operations to be executed at the section before it is processed and executed (the operation has to be *causally-ready*). By comparing the section's state map with the operation state map, the section is able to determine whether the operation is causally-ready. Since the operation might bear more than one state map (one state map for each affected section), the state map used for comparison is obviously the state map of the respective section where the operation is to be executed.

Once an operation is causally ready, as in the consistency management algorithm, the remote operation is then transformed, executed and stored in the history so as to maintain the consistency of the document replica. The main difference is that the history being used is not the operation history of the whole document, but rather the operation history of each affected section.

Figure 6-2 outlines two procedures: procedure *rcv_remote_op* is invoked when a remote operation is received and procedure *exec_remote_op* is invoked to process the remote operation at each section. Note that procedure *exec_remote_op* in Figure 6-2 does not include the consistency management algorithm in detail and readers can refer to section 3.4 for the detail of the consistency management algorithm. The procedure *exec_remote_op* could also be used with other consistency management algorithms, such as SOCT3 [126], and GOTO [131].

```
void rcv_remote_op(<op, $LC_{op}$, $ML_{op}$, $S_{op}$>) {
        //$S_{op}$ is the site that generates op
        //Execute op in each affected section
        for each $ML_{op}[i]$ in $ML_{op}$ {
                //$C_x$ is the affected section
                //$M_{op,C_x}$ is the state map of the
                //affected section as carried by op
                // Explode $ML_{op}[i]$ to derive x and $M_{op,C_x}$
                let [x, $M_{op,C_x}$ ] = $ML_{op}[i]$;

                //Wait until op is causally ready
                //Assume the recipient site is $S_i$.
                wait until $M_{S_i,C_x}[key(S_k)] \geq M_{op,C_x}[key(S_k)]$, $0 \leq k < N$;

                //Execute exec_remote_op procedure to
                //execute op at each affected sections
                exec_remote_op(op, $C_x$, $M_{op,C_x}$ );
        }
}

void exec_remote_op(op, $C_x$, $M_{op,C_x}$) {
        //Transform, and execute operation
        //as per consistency management algorithm

        //Add op into the section's operations history
        $H_{S_i,C_x} = H_{S_i,C_x} + op$;

        //Rearrange the history as necessary.
        :
        :
        //Trim the history as necessary.
        :
        :
}
```

Figure 6-2 Remote operation reception and execution

6.5.3. Joining a Section

When a user at a site, S_i, decides to work on a section, C_x, the site has to make sure the section in its local replica is up to date before starting work on the section. The site can request an updated section and associated operation history data from another site. Site S_i basically requests to *join* in the section collaboration. However, firstly, the site has to notify all other sites that are currently working on C_x. Each site should respond to S_i indicating whether or not it is active on C_x and send its respective M_{S_i,C_x} to let S_i know the state of its section C_x. Site S_i will then decide which site it will contact to obtain the most up-to-date version of C_x along with its operation history and section state. The decision can be automated by choosing the site that has executed the most number of operations on section C_x. Site S_i could also consider other factors such as the corresponding peer-to-peer network delay to make decision. At a given time some sites may be disconnected, thus site S_i may not receive all necessary replies. Therefore, site S_i should be allowed to decide which site to contact without having to wait for all sites to reply. A timeout may be applied here to let S_i wait for a certain period before making a decision.

Suppose site S_k is the one contacted by S_i, S_i basically gets a complete copy of section C_x of S_k in order for it to be able to resume collaboration. Once S_i successfully *joins* in the section, site S_i can either explicitly send a notification to other participants, or let the other participants know about this by receiving an operation generated by S_i on section C_x. Eventually, the other sites that are active on section C_x will add S_i to their membership list so they will include S_i when they send further updates.

Figure 6-3 outlines the procedures involved when a site joins a section: procedure *join_section* is invoked by a site to join a section and procedure *rcv_join_request* is invoked by another site contacted by the newly joining site.

```
void join_section ($C_x$) {
        //Send notification to all sites that $S_i$ wants to join $C_x$.
        send <active,$C_x$> to all sites;

        //Site $S_i$ can wait for all replies, or can decide immediately
        //which site to contact once there are some replies
        Wait for some or all replies until timeout;

        //Let $S_j$ is the site chosen to be contacted
        send join($C_x$) to $S_j$;

        //Wait for reply from $S_j$, and then
        //process all data received from $S_j$
        receive <$C_x$, $H_{S_j,C_x}$, $SC_{S_j,C_x}$, $M_{S_j,C_x}$ > from $S_j$;

        $C_x$ is the new section content of $S_i$;
        $H_{S_i,C_x} = H_{S_i,C_x}$;
        $SC_{S_i,C_x} = SC_{S_j,C_x} + S_i$;
        $M_{S_i,C_x} = M_{S_j,C_x}$;
}

void rcv_join_request (join($C_x$)) {
        //Assume site $S_j$ receives this join request from $S_i$
        send <$C_x$, $H_{S_j,C_x}$, $SC_{S_j,C_x}$, $M_{S_j,C_x}$ > to $S_i$;

        //Add $S_i$ to its section membership
        $SC_{S_j,C_x} = SC_{S_j,C_x} + S_i$;
}
```

Figure 6-3 Joining a section

6.5.4. Leaving a Section

When a user on a site decides to be passive on a section that s/he has been working on, the site will notify all sites that are working on that section so that they will not need to send further updates to the newly passive site. The other sites will simply remove the leaving site from their section membership list.

Suppose site S_i is leaving C_x, S_i should notify all sites in SC_{S_i,C_x} that it is leaving C_x. Some sites may however be disconnected when site S_i is sending the notification; therefore those sites may not receive the notification. To ensure that those sites will eventually receive the notification, the notification should be in a form of a proper operation with an appropriate timestamp (state map). Therefore, when the disconnected sites reconnect and receive future operations, they may recognise that they have missed the operation which is the leaving notification from site S_i.

Figure 6-4 outlines the procedures involved when a site leaves a section. Procedure *leave_section* is invoked by a site to leave section C_x. Since the request to leave a section is broadcast as an operation, procedure *rcv_remote_op* is invoked by other sites that are currently active on section C_x after receiving the leave request. Procedure *rcv_remote_op* is the same as the one presented in Figure 6-2. Procedure *exec_remote_op* is modified so that it recognises the request to leave and execute the operation accordingly.

```
void leave_section ($C_x$) {
        //Send notification to all sites that $S_i$ wants to leave $C_x$
        $op = leave(C_x)$;
        $ML_{op} = ML_{op} + [x, M_{S_i,C_x}]$;

        //Update the section state vector
        $M_{S_i,C_x}[key(S_i)] = M_{S_i,C_x}[key(S_i)] + 1$;

        //Adding the operation into the
        //section's operation history
        $H_{S_i,C_x} = H_{S_i,C_x} + \{op\}$;

        //send the notification
        broadcast(<$op$, $LC_{op}$, $ML_{op}$, $S_{op}$>) to all sites in $SC_{S_i,C_x}$;
}

void exec_remote_op($op$, $C_x$, $M_{op,C_x}$) {
        //Transform, and execute operation
        //as per consistency management algorithm
        if ($op = leave(C_x)$) {
                //Remove $S_i$ from its section membership
                $SC_{S_j,C_x} = SC_{S_j,C_x} - [S_i]$;
        }

        //Add $op$ into the section's operations history
        $H_{S_j,C_x} = H_{S_j,C_x} + op$;

        //Rearrange the history as necessary and/or
        //Trim the history as necessary.
        :
        :
}
```

Figure 6-4 Leaving a section

6.5.5. Creating a New Section

Creating a new section simply involves two things: creating the new section C_{new} and adding C_{new} into the document index DI_{S_i}. The challenge is how to add the new section into the document index when two or more sections are being created concurrently by different sites.

This problem can be treated in the same way as two or more users modifying a document concurrently (consistency management algorithm) with the document index being treated as the document being modified and the section ids as the objects being inserted. When two or more users are inserting new sections at different positions in the document index, the new section positions are properly transformed so that the new sections are placed correctly and consistently across all document indexes. On the other hand, when new sections are created at the same position, then a conflict resolution strategy has to be invoked to allow users to agree on the correct order of the new sections.

6.6. Performance Analysis

The performance of the proposed algorithm has been evaluated both theoretically and empirically to determine whether or not the document partitioning algorithm is worth implementing and under what scenario the document is best left intact instead of partitioned into several sections. This section discusses the theoretical resource requirement of the document partitioning technique, followed by presentation of the empirical performance analysis.

6.6.1. Theoretical Performance Analysis

Let *T* be the number of objects in the document, *P* be to average number of generated operations at each site, *m* be the number of sections in the document, *X* be the average number of sections in each site, and *B* be the number of sites that work on each section. For simplicity, it is assumed that each section is worked on by the same number of sites; therefore $B = {X.N}/{m}$. Table 6-1 presents the resource consumption of the proposed document partitioning technique.

	Unpartitioned Document	**Partitioned Document**
No. of objects per site	T	$T \cdot X/m$
No. of received ops	$(P \cdot N) - P$	$(P \cdot N \cdot X/m) - P$
No. of op broadcast	$(P \cdot N) - P$	$(P \cdot N \cdot X/m) - P$
History size (without trimming)	$P \cdot N$	$P \cdot N \cdot X/m$
State map table elements	$1 \cdot N \cdot N = N^2$	$X \cdot B \cdot B = X \cdot \frac{X \cdot N}{m} \cdot \frac{X \cdot N}{m}$ $= N^2 \cdot X^3/m^2$

Table 6-1 Resource consumption

It is obvious that with document partitioning, the number of objects, the number of operations transmitted (received and broadcast), and the history size are reduced by the factor of X/m. The lower X is, the less resources consumed. The state map table elements, however, are larger with document partitioning (by the order of X^3/m^2) since each section maintains its own state map table for garbage collecting purpose. Therefore, the number of state map table elements are kept minimum if $X^3/m^2 \leq 1$, or in other words, the total size of the state map table with document partitioning can be reduced if $X \leq m^{2/3}$. As an example, if a document is divided into 10 sections (m = 10), each site works only on 4 sections or less ($X \leq 4$) in order to keep the size of the state map table the same as or less than the one in the unpartitioned counterpart.

6.6.2. Empirical Performance Analysis

The proposed algorithm has been tested in a simulation environment and its performance parameters have been recorded and analysed. The aim of the empirical performance analysis is to compare the partitioned document with the unpartitioned document and to determine whether or not document partitioning reduces resource consumption while providing users with the flexibility of selecting document parts.

The independent variables are the number of sites (3, 5, and 10 sites), the number of operations generated by each site (30, 50 and 100), the number of document partitions (1, 3, 5, 10) and the number of partitions actually held by each site (from 1 to the actual number of document partitions). The performance parameters (dependent variables) are (1) the maximum history size (with and without history trimming), (2) the average processing time per operation, (3) the size of the state map table, and (4) the total number of messages transferred.

Firstly, Figure 6-5, Figure 6-6, and Figure 6-7 show the maximum history size of each participant when there are 3, 5 and 10 participants respectively in the collaboration. Each site generates either 30, 50 or 100 operations and the chart shows the maximum history size either with or without history trimming. The axis is a pair of two independent variables: the number of document partitions and the number of partitions per site. For example, (5, 3) means that there are 5 partitions in the document and each site holds 3 partitions. Some partitions can be held by more than one site, but not all sites have all the partitions. The axis of (1, 1) means that there is only one partition and all sites have that 1 partition, in other words, (1, 1) is the *unpartitioned document*.

It can be seen from the graphs that dividing the document into partitions reduces the maximum history size in each site (to as little as 15%), provided that each site does not hold all partitions regardless of whether or not the history trimming is implemented. If each site holds all partitions, then document partitioning is simply an unnecessary overhead since the users are collaborating as if there is no partition. However, with history trimming, the maximum history size can be smaller than the unpartitioned document, since there are fewer sites that work on each partition; therefore the history is trimmed more regularly.

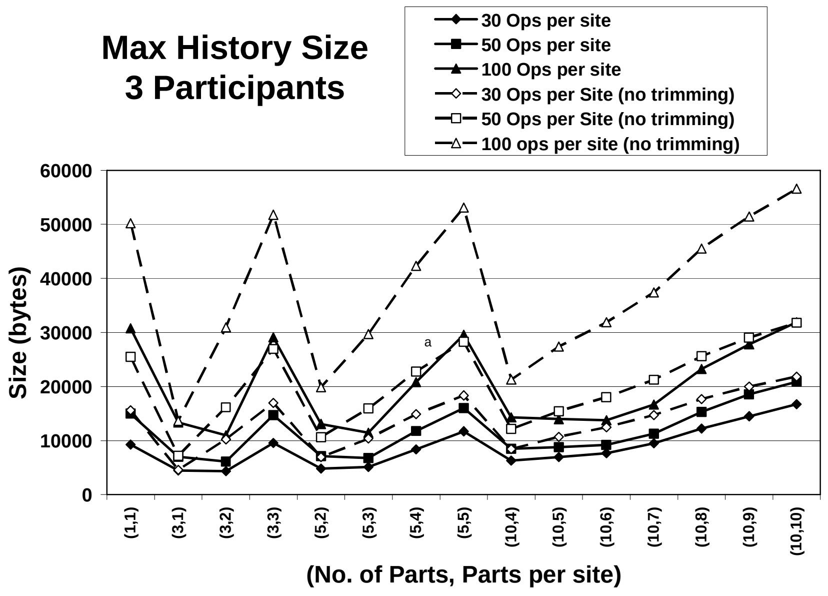


Figure 6-5 Maximum history size - 3 participants

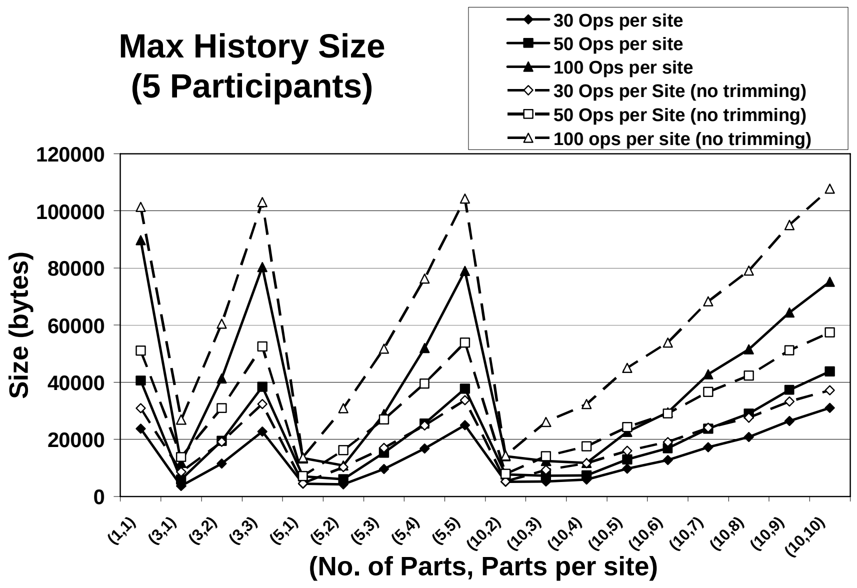


Figure 6-6 Maximum history size - 5 participants

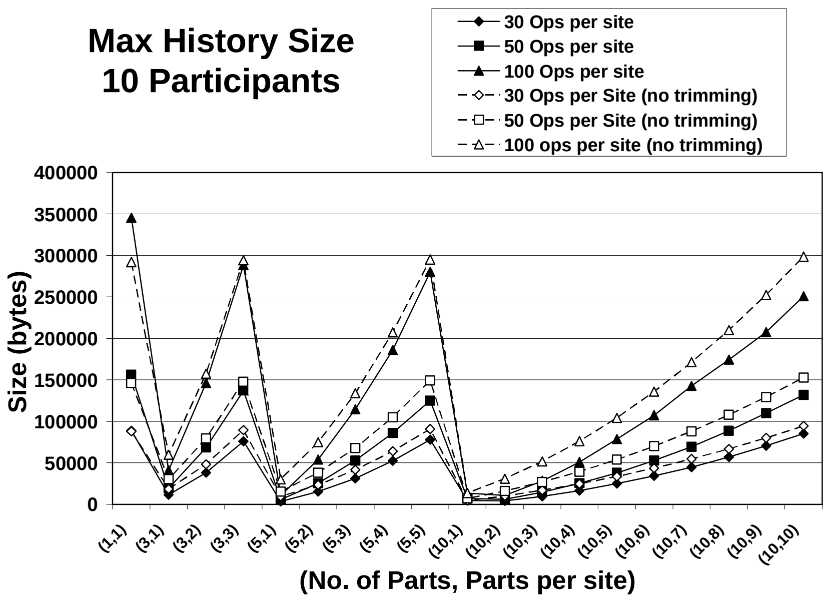


Figure 6-7: **Maximum history size – 10 participants**

Secondly, Figure 6-8 displays the average processing time per operation, and it is obvious that the time to process an operation in a partitioned document is significantly less than the unpartitioned document. This is because without document partitioning, the history size is greater and each remote operation processing requires tracing through the whole history which, in an unpartitioned document, includes all operations whereas in partitioned document, it only includes the operations involved in that particular partition.

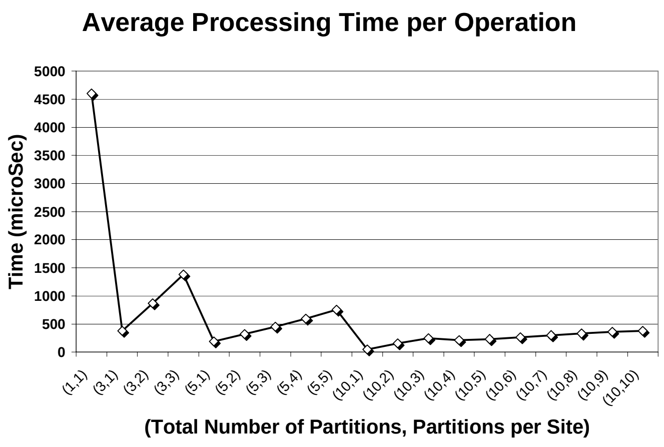


Figure 6-8: Average processing time

Thirdly, Figure 6-9 shows the size of the state map table for various numbers of document partitions. The state map table size increases as the number or partitions held by each site increases. However, as expected from the theoretical calculation, the state map table size is less than or equal to the unpartitioned document if $X \le m^{2/3}$, where X is the number of partitions held by each site and m is the actual number of partitions. Although the size of the state map table with document partitioning is increasingly larger as the number of partitions held by each site increases, the storage/memory space saving in the history size is more than the additional consumption by the state map table. Therefore, document partitioning reduces the overall storage/memory space consumption.

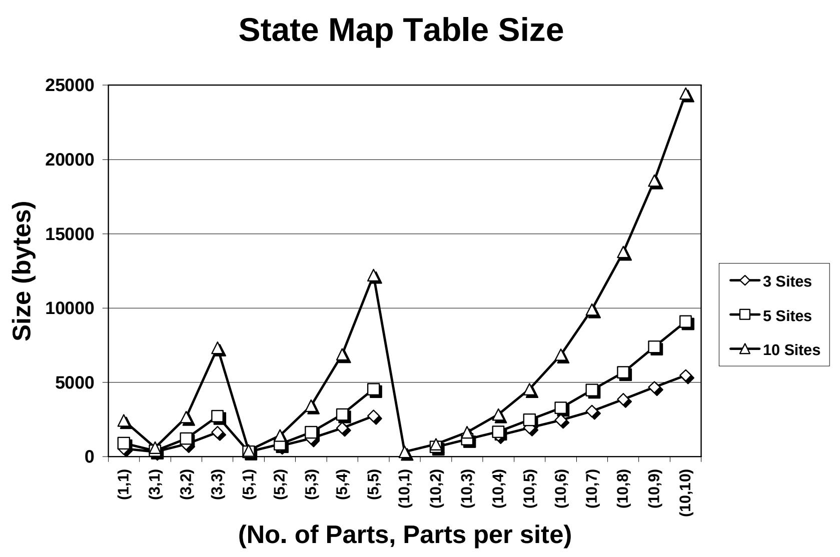


Figure 6-9: State map table size

Finally, bandwidth consumption is also reduced with document partitioning as expected and is shown by Figure 6-10. It is obvious, and expected theoretically, that the number of messages transferred will be reduced with document partitioning. It is also obvious that as the number of partitions held by each site increases, the number of messages transferred also increases. However, it will not exceed the number of messages without document partitioning.

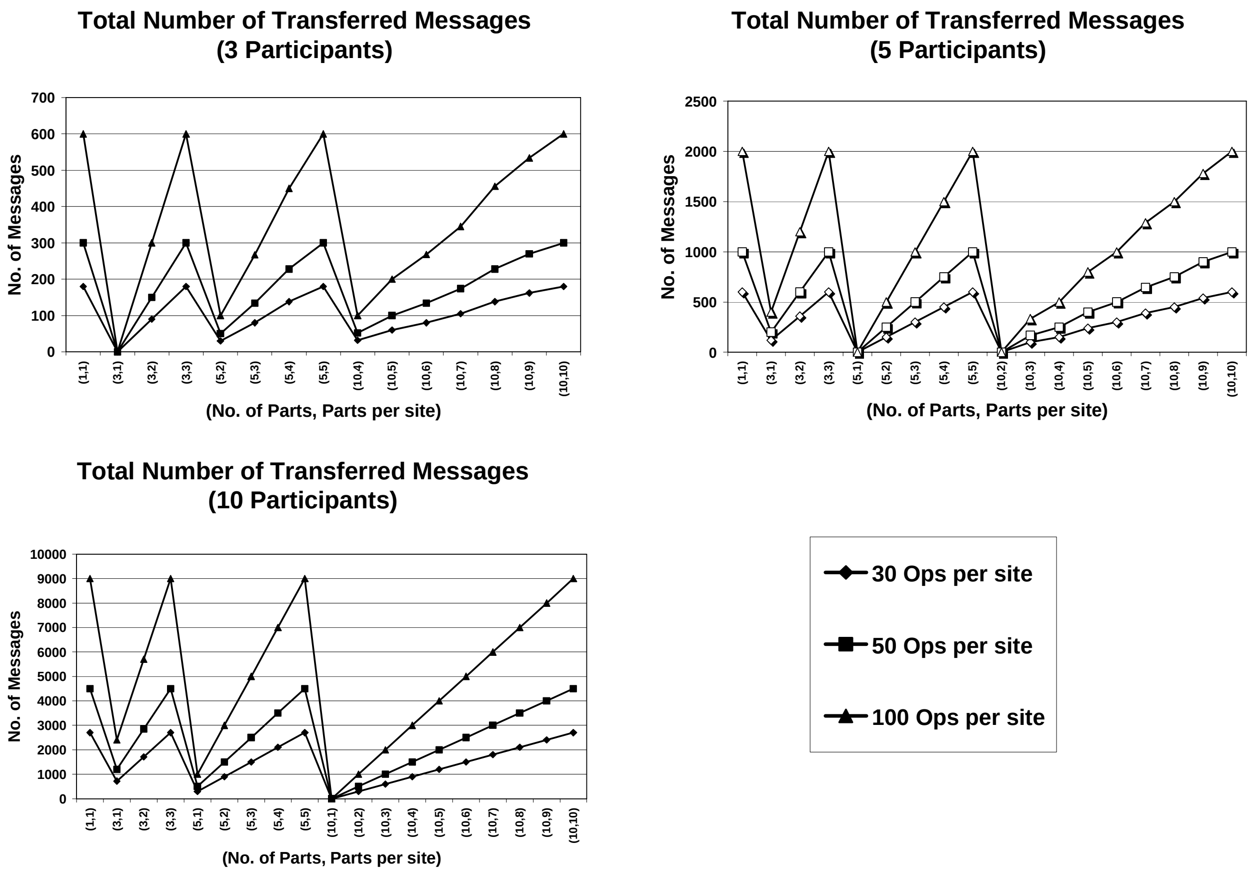


Figure 6-10: Transferred messages

6.7. Conclusion

This chapter has presented the need for a document partitioning algorithm in real time mobile collaboration systems to provide flexibility for users to work only on a particular part of the document. There are some reasons a user may choose to be active only on some sections: the limitation of his/her device interface, s/he is not interested in other sections, and/or to minimise resource consumption. The document partitioning, however, presents some challenges with regards to the section boundaries, section membership, section creation, and sites joining and leaving a section. This chapter has also presented an algorithm that provides a mechanism for document partitioning while addressing the above challenges. By dividing a document into pre-defined sections and adding an independent membership mechanism to each section, a user can choose to work on his/her desired sections without having to worry about what happens to other sections.

Furthermore, the performance of the proposed algorithm has been evaluated, both theoretically and empirically. It has been theoretically shown that by partitioning the document into several sections and allowing users to work only on selected sections, the storage space requirement is reduced since the number of objects per site and the size of the operation history decreases. The bandwidth usage is also reduced as the number of operations being broadcast is reduced due to the fact that not all sites need to receive all generated operations. The state map table elements, however, are larger with document partitioning (by the order of X^3/m^2) since each section maintains its own state map table for garbage collecting purpose. The total size of the state map table with document partitioning is however lower than its unpartitioned counterpart if $X \leq m^{2/3}$. The empirical performance analysis has also shown that by partitioning the document, the proposed strategy reduces resource consumption as follows: (1) it reduces the maximum history size in each site (to as little as 15%), (2) the time to process an operation in a partitioned document is significantly less than the unpartitioned document, and (3) the greater the number of partitions, the fewer messages are transferred.

Future work may involve extending the algorithm to allow dynamic unpredefined section creation, such as splitting a section into several sections and merging two or more sections.

Chapter 7

Application

7.1. Introduction

The previous chapters have proposed various algorithms to support real-time collaborative editing in mobile replicated architecture. This chapter describes an application prototype that implements the proposed framework. This chapter is organised as follows: section 7.2 describes the system architecture of a collaborative editor, section 7.3 presents an example of a collaborative editing scenario, and section 7.4 concludes the chapter.

7.2. System Architecture

This section describes the system architecture of the proposed collaborative framework. Based on the discussions in previous chapters, and as illustrated in Figure 7-1, each collaboration site consists of the following collaboration framework components:

1. *Connection Manager*, responsible for sending and receiving messages to and from other collaboration peers.
2. *Collaboration Engine,* responsible for managing the whole collaboration. It receives the messages from the connection manager and it forwards each message to the appropriate component. If it is a join request or an operation request or a message related to a membership event, it is forwarded to the membership manager. Every message related to document partitioning is forwarded to the document partition manager. Any conflict related message is forwarded to be processed by the conflict manager. The collaboration engine itself contains the consistency management module that processes ordinary operations to ensure it is processed and executed appropriately to the shared document.
3. *Membership Manager,* responsible for managing the session and the membership events. It maintains the list of the collaboration participants. It also ensures all operations that the site has missed during disconnection period will be requested and received when it is reconnected.
4. *Document Manager,* responsible for managing the document and its partitions. It maintains the document index as the representation of the complete document. It also facilitates the joining and the leaving of a particular document partitioning as desired by the user.
5. *Conflict Manager,* responsible for managing and handling the conflict, and enabling users to resolve the conflict.

6. *Document Replica,* the data type and the raw data of the document replica.
7. *Editor,* contains the tools for editing the shared document.
8. *Presentation,* responsible for presenting the document and the editor to the user interface.
9. *Input Controller,* receives input from the user and translates them into the editor.

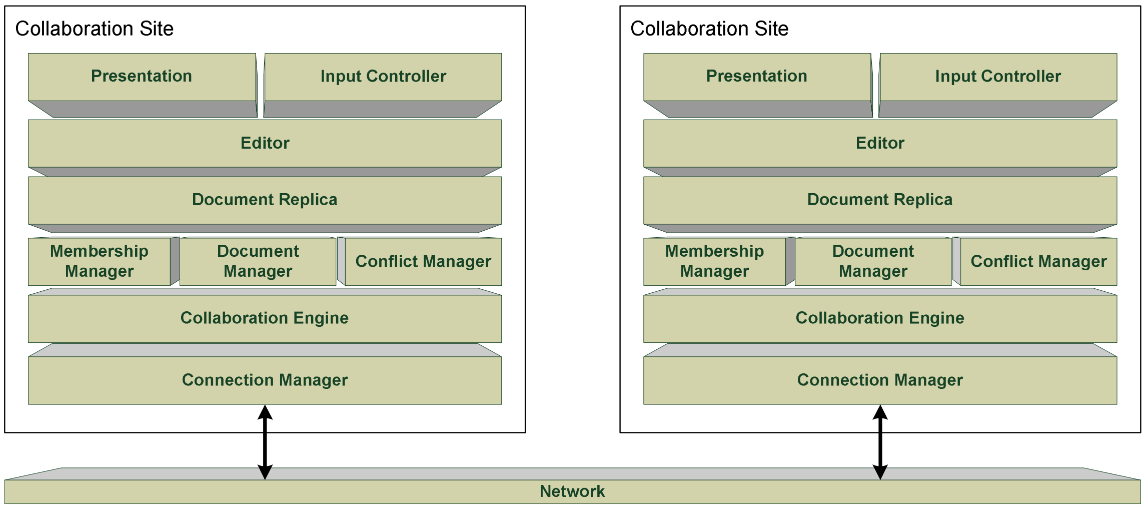


Figure 7-1 System architecture

The following are the class diagrams of the collaborative editing application prototype. Firstly, Figure 7-2 shows the classes that are related to the document. Secondly, Figure 7-3 shows the connection manager related classes. Thirdly, Figure 7-4 shows the collaboration engine class that manage the key components of the collaboration. Fourthly, Figure 7-5 shows the classes involved in handling and resolving conflicts. Fifthly, Figure

7-6 shows classes involved in dealing with membership events. Finally, Figure 7-7 shows the complete class diagram of the collaborative editor.

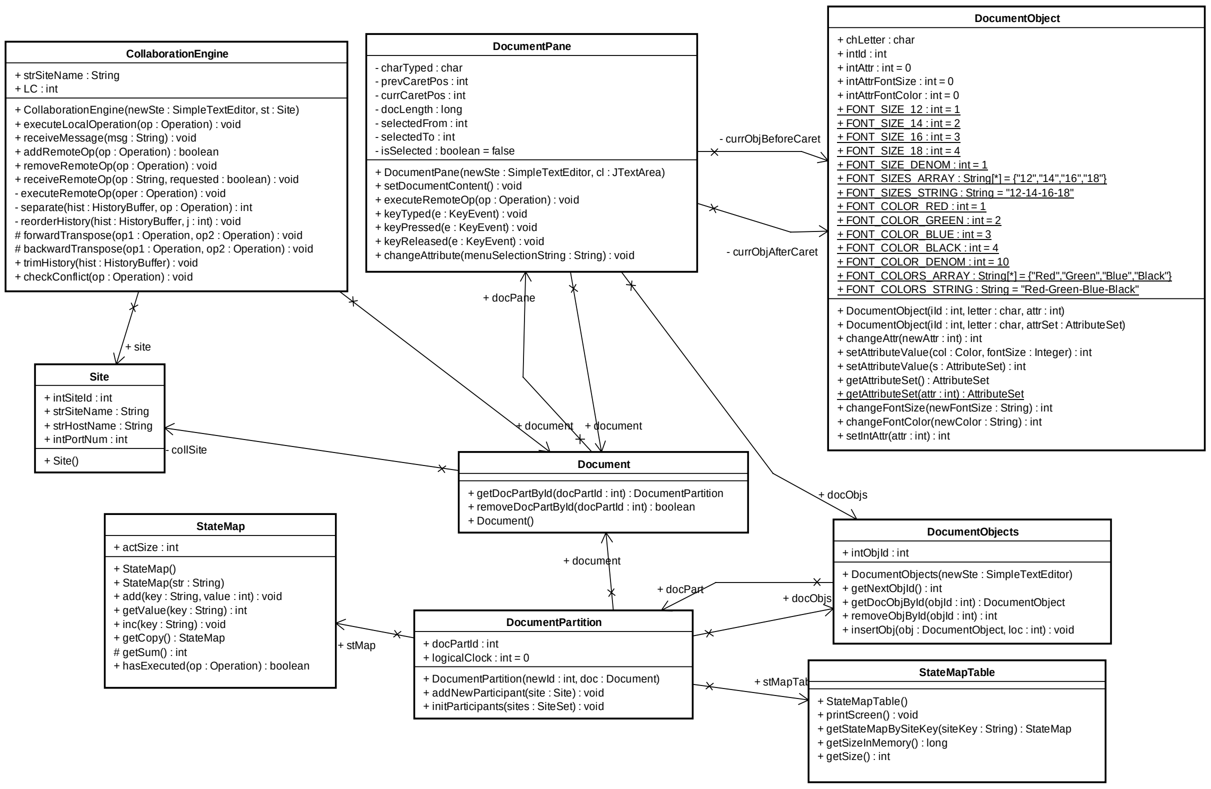

CollaborationEngine
+ strSiteName : String
+ LC : int
+ CollaborationEngine(newSte : SimpleTextEditor, st : Site)
+ executeLocalOperation(op : Operation) : void
+ receiveMessage(msg : String) : void
+ addRemoteOp(op : Operation) : boolean
+ removeRemoteOp(op : Operation) : void
+ receiveRemoteOp(op : String, requested : boolean) : void
- executeRemoteOp(oper : Operation) : void
- separate(hist : HistoryBuffer, op : Operation) : int
- reorderHistory(hist : HistoryBuffer, j : int) : void
# forwardTranspose(op1 : Operation, op2 : Operation) : void
# backwardTranspose(op1 : Operation, op2 : Operation) : void
+ trimHistory(hist : HistoryBuffer) : void
+ checkConflict(op : Operation) : void
DocumentPane
- charTyped : char
- prevCaretPos : int
- currCaretPos : int
- docLength : long
- selectedFrom : int
- selectedTo : int
- isSelected : boolean = false
+ DocumentPane(newSte : SimpleTextEditor, cl : JTextArea)
+ setDocumentContent() : void
+ executeRemoteOp(op : Operation) : void
+ keyTyped(e : KeyEvent) : void
+ keyPressed(e : KeyEvent) : void
+ keyReleased(e : KeyEvent) : void
+ changeAttribute(menuSelectionString : String) : void
DocumentObject
+ chLetter : char
+ intId : int
+ intAttr : int = 0
+ intAttrFontSize : int = 0
+ intAttrFontColor : int = 0
+ FONT_SIZE_12 : int = 1
+ FONT_SIZE_14 : int = 2
+ FONT_SIZE_16 : int = 3
+ FONT_SIZE_18 : int = 4
+ FONT_SIZE_DENOM : int = 1
+ FONT_SIZES_ARRAY : String[*] = {"12","14","16","18"}
+ FONT_SIZES_STRING : String = "12-14-16-18"
+ FONT_COLOR_RED : int = 1
+ FONT_COLOR_GREEN : int = 2
+ FONT_COLOR_BLUE : int = 3
+ FONT_COLOR_BLACK : int = 4
+ FONT_COLOR_DENOM : int = 10
+ FONT_COLORS_ARRAY : String[*] = {"Red","Green","Blue","Black"}
+ FONT_COLORS_STRING : String = "Red-Green-Blue-Black"
+ DocumentObject(iId : int, letter : char, attr : int)
+ DocumentObject(iId : int, letter : char, attrSet : AttributeSet)
+ changeAttr(newAttr : int) : int
+ setAttributeValue(col : Color, fontSize : Integer) : int
+ setAttributeValue(s : AttributeSet) : int
+ getAttributeSet() : AttributeSet
+ getAttributeSet(attr : int) : AttributeSet
+ changeFontSize(newFontSize : String) : int
+ changeFontColor(newColor : String) : int
+ setIntAttr(attr : int) : int
- currObjBeforeCaret
- currObjAfterCaret
+ docPane
+ site
+ document
+ document
Site
+ intSiteId : int
+ strSiteName : String
+ strHostName : String
+ intPortNum : int
+ Site()
- collSite
Document
+ getDocPartById(docPartId : int) : DocumentPartition
+ removeDocPartById(docPartId : int) : boolean
+ Document()
+ docObjs
StateMap
+ actSize : int
+ StateMap()
+ StateMap(str : String)
+ add(key : String, value : int) : void
+ getValue(key : String) : int
+ inc(key : String) : void
+ getCopy() : StateMap
# getSum() : int
+ hasExecuted(op : Operation) : boolean
+ document
+ docPart
+ docObjs
DocumentObjects
+ intObjId : int
+ DocumentObjects(newSte : SimpleTextEditor)
+ getNextObjId() : int
+ getDocObjById(objId : int) : DocumentObject
+ removeObjById(objId : int) : int
+ insertObj(obj : DocumentObject, loc : int) : void
+ stMap
DocumentPartition
+ docPartId : int
+ logicalClock : int = 0
+ DocumentPartition(newId : int, doc : Document)
+ addNewParticipant(site : Site) : void
+ initParticipants(sites : SiteSet) : void
+ stMapTab
StateMapTable
+ StateMapTable()
+ printScreen() : void
+ getStateMapBySiteKey(siteKey : String) : StateMap
+ getSizeInMemory() : long
+ getSize() : int

Figure 7-2 Document

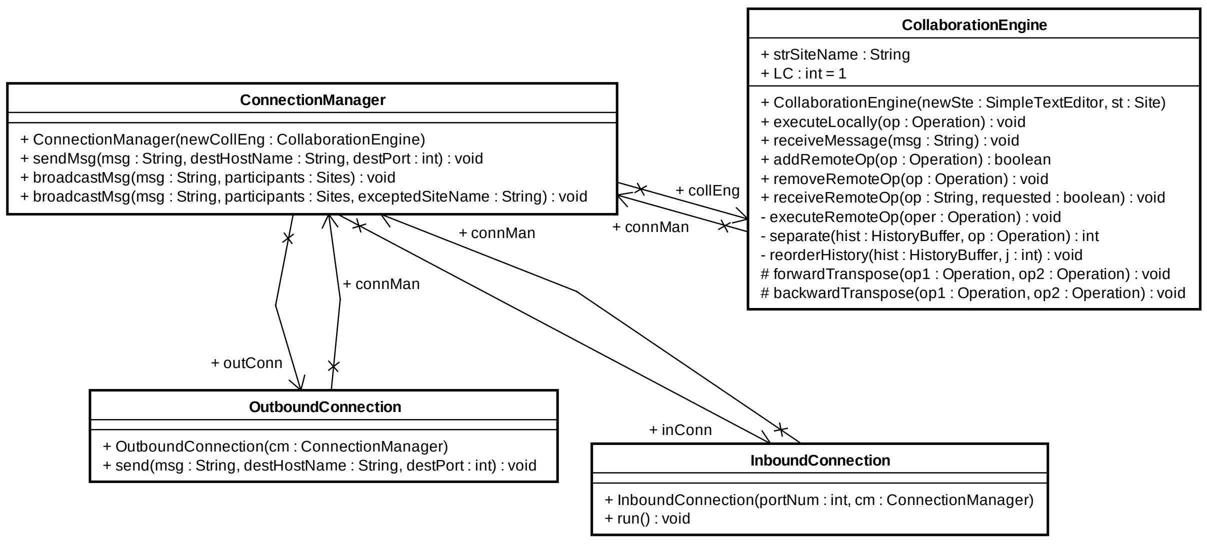

CollaborationEngine
+ strSiteName : String
+ LC : int = 1
+ CollaborationEngine(newSte : SimpleTextEditor, st : Site)
+ executeLocally(op : Operation) : void
+ receiveMessage(msg : String) : void
+ addRemoteOp(op : Operation) : boolean
+ removeRemoteOp(op : Operation) : void
+ receiveRemoteOp(op : String, requested : boolean) : void
- executeRemoteOp(oper : Operation) : void
- separate(hist : HistoryBuffer, op : Operation) : int
- reorderHistory(hist : HistoryBuffer, j : int) : void
# forwardTranspose(op1 : Operation, op2 : Operation) : void
# backwardTranspose(op1 : Operation, op2 : Operation) : void
ConnectionManager
+ ConnectionManager(newCollEng : CollaborationEngine)
+ sendMsg(msg : String, destHostName : String, destPort : int) : void
+ broadcastMsg(msg : String, participants : Sites) : void
+ broadcastMsg(msg : String, participants : Sites, exceptedSiteName : String) : void
+ collEng
+ connMan
+ connMan
+ connMan
+ outConn
+ inConn
OutboundConnection
+ OutboundConnection(cm : ConnectionManager)
+ send(msg : String, destHostName : String, destPort : int) : void
InboundConnection
+ InboundConnection(portNum : int, cm : ConnectionManager)
+ run() : void

Figure 7-3 Connection manager

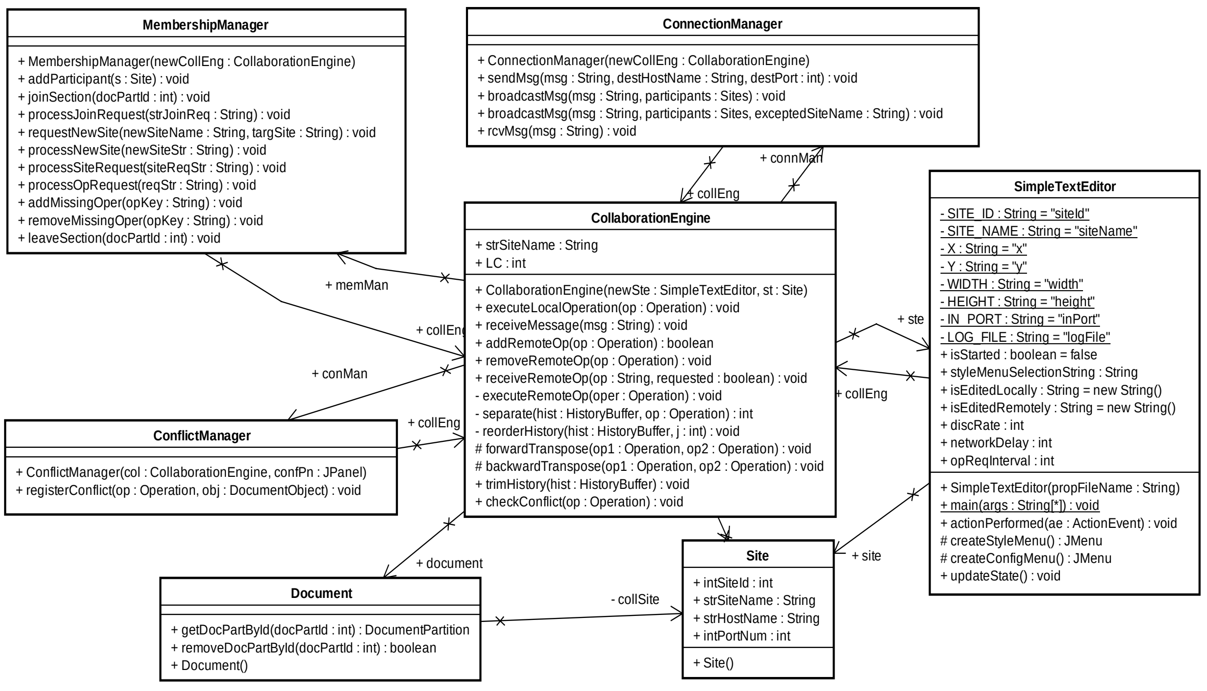

MembershipManager
+ MembershipManager(newCollEng : CollaborationEngine)
+ addParticipant(s : Site) : void
+ joinSection(docPartId : int) : void
+ processJoinRequest(strJoinReq : String) : void
+ requestNewSite(newSiteName : String, targSite : String) : void
+ processNewSite(newSiteStr : String) : void
+ processSiteRequest(siteReqStr : String) : void
+ processOpRequest(reqStr : String) : void
+ addMissingOper(opKey : String) : void
+ removeMissingOper(opKey : String) : void
+ leaveSection(docPartId : int) : void
ConnectionManager
+ ConnectionManager(newCollEng : CollaborationEngine)
+ sendMsg(msg : String, destHostName : String, destPort : int) : void
+ broadcastMsg(msg : String, participants : Sites) : void
+ broadcastMsg(msg : String, participants : Sites, exceptedSiteName : String) : void
+ rcvMsg(msg : String) : void
+ connMan
+ collEng
CollaborationEngine
+ strSiteName : String
+ LC : int
+ CollaborationEngine(newSte : SimpleTextEditor, st : Site)
+ executeLocalOperation(op : Operation) : void
+ receiveMessage(msg : String) : void
+ addRemoteOp(op : Operation) : boolean
+ removeRemoteOp(op : Operation) : void
+ receiveRemoteOp(op : String, requested : boolean) : void
- executeRemoteOp(oper : Operation) : void
- separate(hist : HistoryBuffer, op : Operation) : int
- reorderHistory(hist : HistoryBuffer, j : int) : void
# forwardTranspose(op1 : Operation, op2 : Operation) : void
# backwardTranspose(op1 : Operation, op2 : Operation) : void
+ trimHistory(hist : HistoryBuffer) : void
+ checkConflict(op : Operation) : void
+ memMan
+ collEng
+ conMan
+ collEng
+ ste
+ collEng
SimpleTextEditor
- SITE_ID : String = "siteId"
- SITE_NAME : String = "siteName"
- X : String = "x"
- Y : String = "y"
- WIDTH : String = "width"
- HEIGHT : String = "height"
- IN_PORT : String = "inPort"
- LOG_FILE : String = "logFile"
+ isStarted : boolean = false
+ styleMenuSelectionString : String
+ isEditedLocally : String = new String()
+ isEditedRemotely : String = new String()
+ discRate : int
+ networkDelay : int
+ opReqInterval : int
+ SimpleTextEditor(propFileName : String)
+ main(args : String[*]) : void
+ actionPerformed(ae : ActionEvent) : void
# createStyleMenu() : JMenu
# createConfigMenu() : JMenu
+ updateState() : void
ConflictManager
+ ConflictManager(col : CollaborationEngine, confPn : JPanel)
+ registerConflict(op : Operation, obj : DocumentObject) : void
+ document
+ site
- collSite
Document
+ getDocPartById(docPartId : int) : DocumentPartition
+ removeDocPartById(docPartId : int) : boolean
+ Document()
Site
+ intSiteId : int
+ strSiteName : String
+ strHostName : String
+ intPortNum : int
+ Site()

Figure 7-4 Collaboration engine

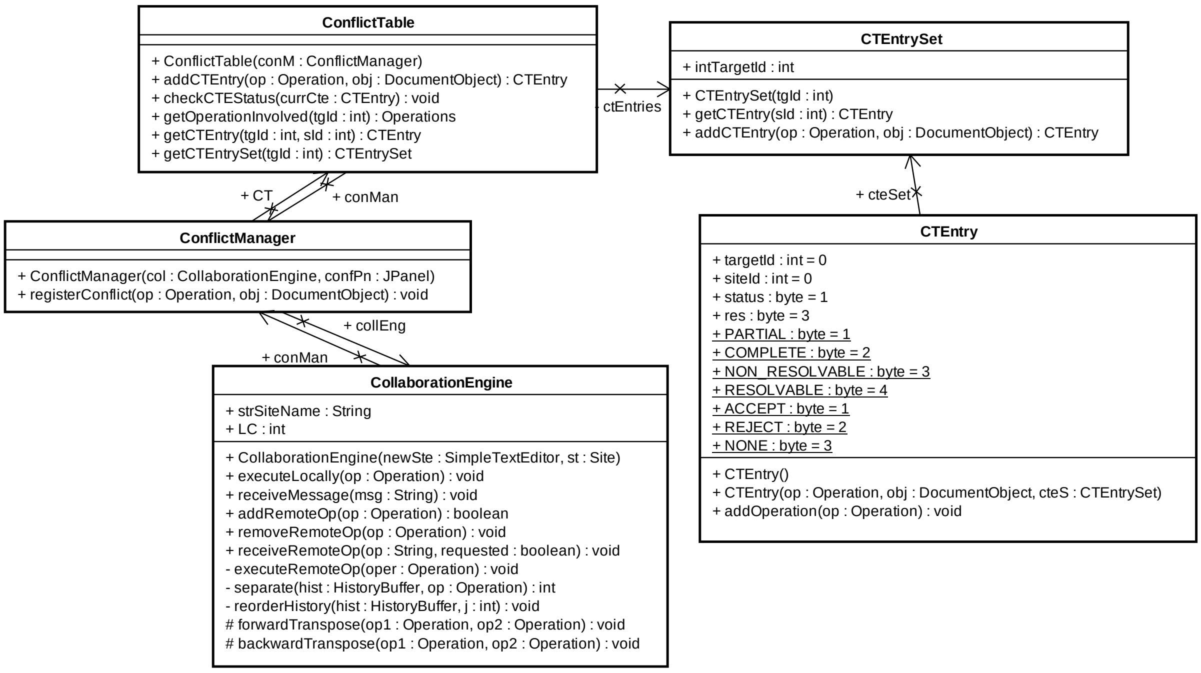

ConflictTable
+ ConflictTable(conM : ConflictManager)
+ addCTEntry(op : Operation, obj : DocumentObject) : CTEntry
+ checkCTEStatus(currCte : CTEntry) : void
+ getOperationInvolved(tgId : int) : Operations
+ getCTEntry(tgId : int, sId : int) : CTEntry
+ getCTEntrySet(tgId : int) : CTEntrySet
- ctEntries
CTEntrySet
+ intTargetId : int
+ CTEntrySet(tgId : int)
+ getCTEntry(sId : int) : CTEntry
+ addCTEntry(op : Operation, obj : DocumentObject) : CTEntry
+ CT
+ conMan
+ cteSet
ConflictManager
+ ConflictManager(col : CollaborationEngine, confPn : JPanel)
+ registerConflict(op : Operation, obj : DocumentObject) : void
CTEntry
+ targetId : int = 0
+ siteId : int = 0
+ status : byte = 1
+ res : byte = 3
+ PARTIAL : byte = 1
+ COMPLETE : byte = 2
+ NON_RESOLVABLE : byte = 3
+ RESOLVABLE : byte = 4
+ ACCEPT : byte = 1
+ REJECT : byte = 2
+ NONE : byte = 3
+ CTEntry()
+ CTEntry(op : Operation, obj : DocumentObject, cteS : CTEntrySet)
+ addOperation(op : Operation) : void
+ collEng
+ conMan
CollaborationEngine
+ strSiteName : String
+ LC : int
+ CollaborationEngine(newSte : SimpleTextEditor, st : Site)
+ executeLocally(op : Operation) : void
+ receiveMessage(msg : String) : void
+ addRemoteOp(op : Operation) : boolean
+ removeRemoteOp(op : Operation) : void
+ receiveRemoteOp(op : String, requested : boolean) : void
- executeRemoteOp(oper : Operation) : void
- separate(hist : HistoryBuffer, op : Operation) : int
- reorderHistory(hist : HistoryBuffer, j : int) : void
# forwardTranspose(op1 : Operation, op2 : Operation) : void
# backwardTranspose(op1 : Operation, op2 : Operation) : void

Figure 7-5 Conflict manager

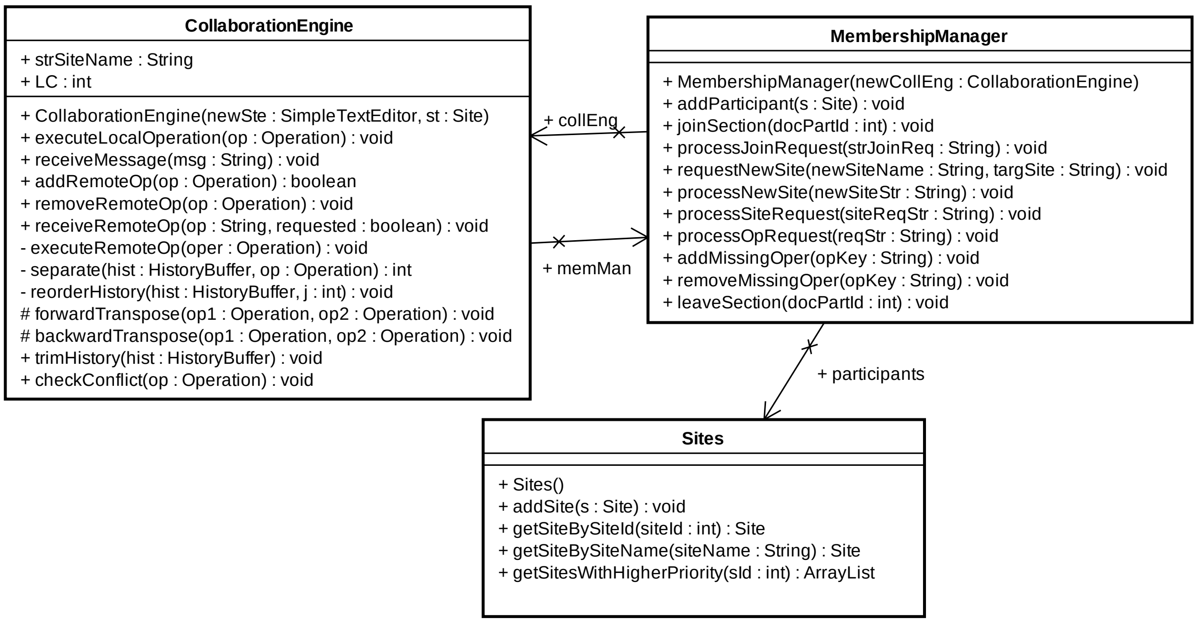

CollaborationEngine
+ strSiteName : String
+ LC : int
+ CollaborationEngine(newSte : SimpleTextEditor, st : Site)
+ executeLocalOperation(op : Operation) : void
+ receiveMessage(msg : String) : void
+ addRemoteOp(op : Operation) : boolean
+ removeRemoteOp(op : Operation) : void
+ receiveRemoteOp(op : String, requested : boolean) : void
- executeRemoteOp(oper : Operation) : void
- separate(hist : HistoryBuffer, op : Operation) : int
- reorderHistory(hist : HistoryBuffer, j : int) : void
# forwardTranspose(op1 : Operation, op2 : Operation) : void
# backwardTranspose(op1 : Operation, op2 : Operation) : void
+ trimHistory(hist : HistoryBuffer) : void
+ checkConflict(op : Operation) : void
+ collEng
+ memMan
MembershipManager
+ MembershipManager(newCollEng : CollaborationEngine)
+ addParticipant(s : Site) : void
+ joinSection(docPartId : int) : void
+ processJoinRequest(strJoinReq : String) : void
+ requestNewSite(newSiteName : String, targSite : String) : void
+ processNewSite(newSiteStr : String) : void
+ processSiteRequest(siteReqStr : String) : void
+ processOpRequest(reqStr : String) : void
+ addMissingOper(opKey : String) : void
+ removeMissingOper(opKey : String) : void
+ leaveSection(docPartId : int) : void
+ participants
Sites
+ Sites()
+ addSite(s : Site) : void
+ getSiteBySiteId(siteId : int) : Site
+ getSiteBySiteName(siteName : String) : Site
+ getSitesWithHigherPriority(sId : int) : ArrayList

Figure 7-6 Membership manager

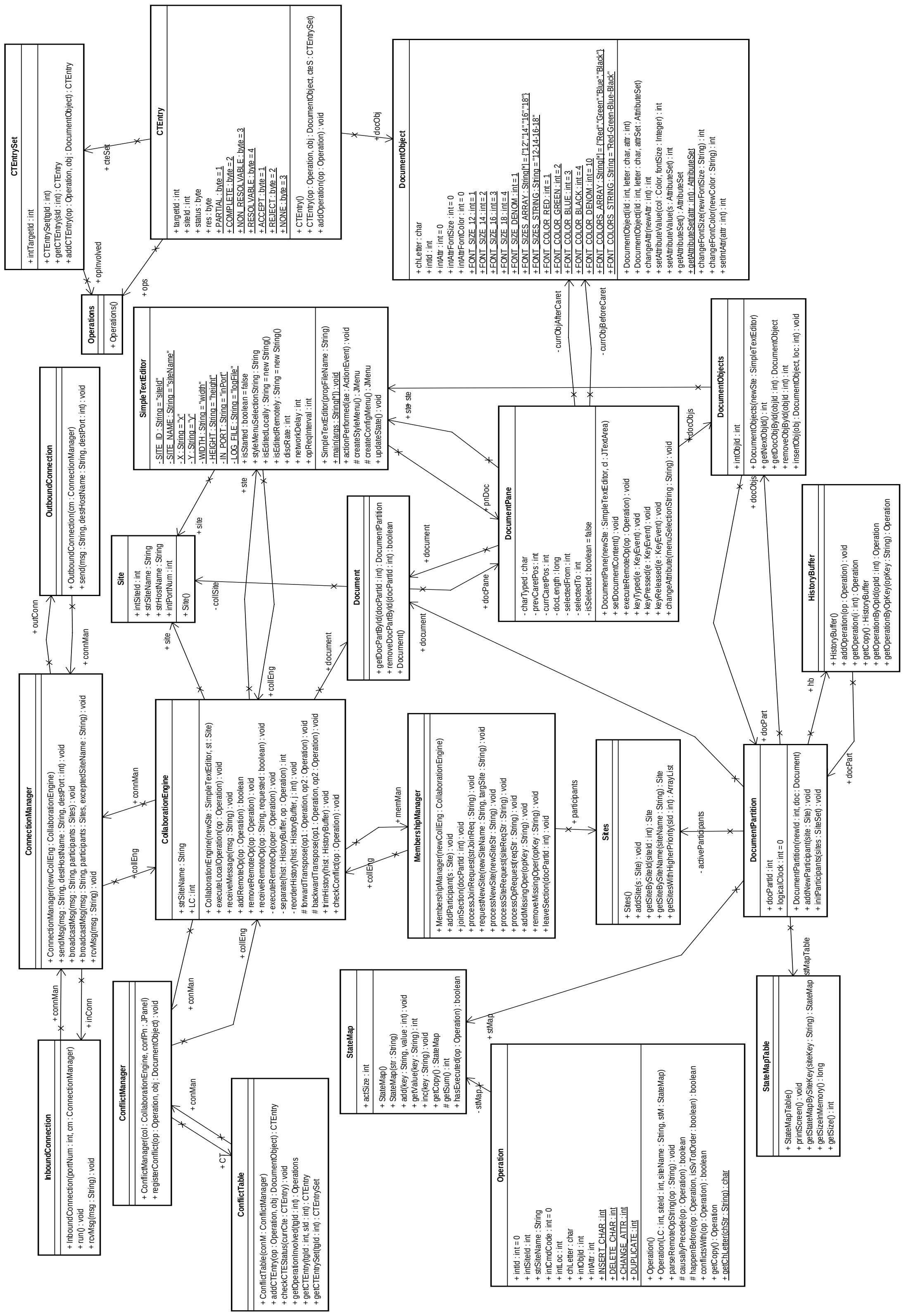


Figure 7-7 Class diagram

7.3. A Collaborative Editing Scenario

This section describes a sample scenario of collaborative editing in mobile replicated architecture and points out how each of the algorithms proposed in this thesis is used to support the collaboration.

Suppose Alice and Bob are two authors of a paper to be submitted to an upcoming conference. They want to be able to contribute to the paper both individually and collaboratively. At the first meeting, they decide that there will be four chapters in the paper: (1) introduction, (2) literature review, (3) proposed algorithm and (4) conclusion. Alice initiates the collaborative paper in her laptop with a new blank copy of the shared document. Alice then writes the paper title and creates the paper structure, each with empty content. Bob then joins the collaboration and receives the copy of the shared document from Alice and stores it as a local replica in his device. They also decide that Alice should write the draft of the introduction section and literature review section while Bob will write the proposed algorithm and conclusion sections. With the document partitioning mechanism presented in Chapter 6, Alice and Bob only focus and work on their selected sections (Figure 7-8). They will then review the draft collaboratively without having to meet at a particular place. During this period, they are "disconnected" and they can work on their local replica independently.

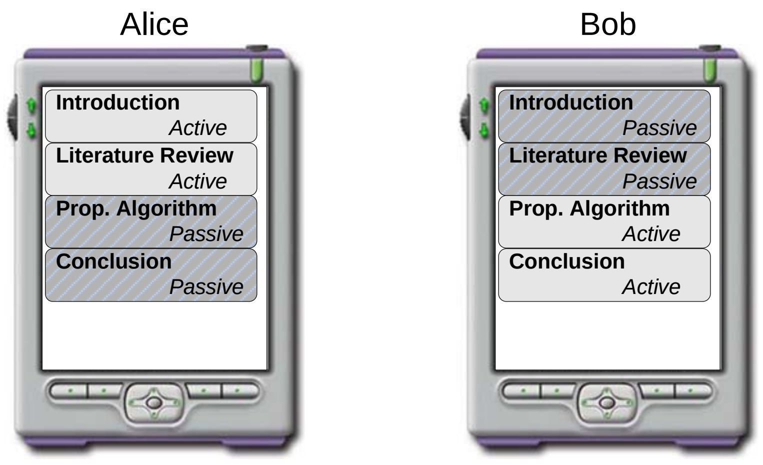


Figure 7-8 Initial states of Alice's and Bob's devices

A week later, Alice and Bob, both at different places, meet again over the wireless network. When they are connected, Alice and Bob can start collaborating on the shared document. Alice needs to get the updated document on the two sections that Bob has been working on, therefore Alice ‘joins’ into those two sections. During the ‘join’ process, Alice receives the updated sections from Bob, Alice becomes *active* in those two sections, and therefore Alice can review and work on those sections. In the same way, Bob ‘joins’ into the two sections Alice has been working on in order to get the latest state of the sections. They then review the flow of the paper after the initial content of all sections are put together (Figure 7-9). During this connected period, both Alice and Bob can make changes to any part of the document and the changes are synchronously sent to each other so they can view the changes that take place during the meeting. The consistency maintenance algorithm presented in Chapter 3 ensures that the concurrent changes are applied appropriately and consistently on both Alice’s and Bob’s local replicas. As an example, Alice and Bob both being active on the introduction chapter can update this chapter concurrently: they may concurrently insert or delete some words or characters. The proposed consistency maintenance algorithm ensures that the updates made by Alice are received by Bob and are applied correctly at Bob’s replica despite the concurrent updates made by Bob to his replica. The concurrent updates may however lead to an exclusive conflict if both Alice and Bob are updating the same object (word or character) with different intentions. Depending on the application, conflicts may be defined at different levels of the object hierarchy. For example, Alice and Bob update a word in the paper title concurrently to different values, or they change the caption of a figure differently. Exclusive conflicts are detected whenever they occur and each conflict is handled and resolved consistently using the conflict management presented in Chapter 4. The proposed conflict management method facilitates the users by giving them adequate information about the conflict status and about the intention of each participant involved in the conflict. It also ensures that Alice and Bob have made enough updates to the object to fully represent their intentions before the conflict can be resolved. Once the conflict is ready to be resolved, they can resolve the conflict by selecting a version of the object (which represents the intention of one of the users) either by voting or by other resolution strategies.

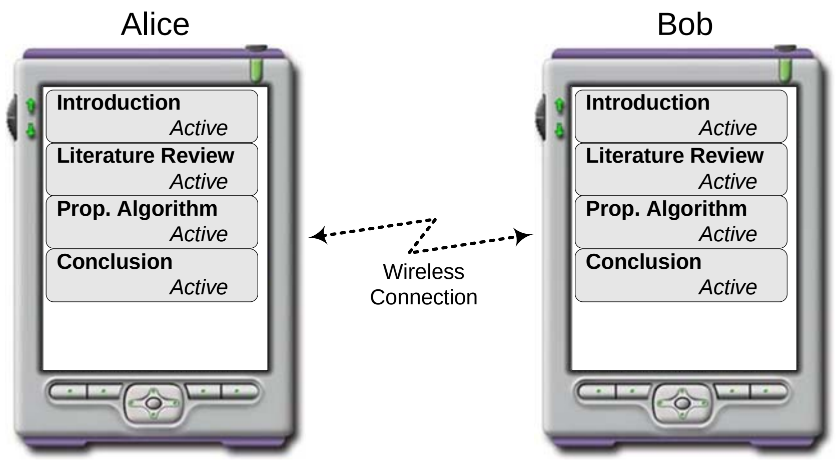
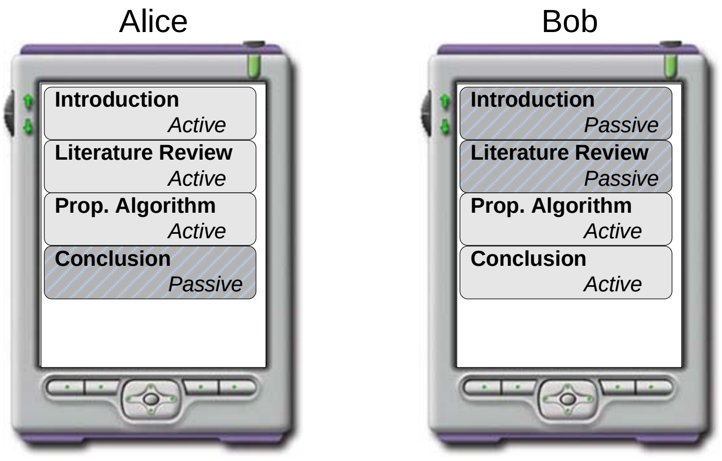


Figure 7-9 Alice and Bob are collaborative over a wireless connection

During the collaboration session, due to reasons such as radio frequency noise or a user going under a tunnel, Alice and Bob may be disconnected from time to time. During this disconnection period, they can still work on their local replicas. Once they are reconnected, using the algorithm proposed in Chapter 5, both devices will send and receive the missed updates, i.e. the operations that have been generated during the disconnection period.

Before the meeting is over, they can choose whether they want to be 'active' on all sections of the document or they only want to be 'active' on some selected sections. Suppose they decide that Alice will be 'active' on the first three chapters, while Bob will be 'active' on the last two chapters as illustrated in Figure 7-10. Bob has also requested Alice to invite Cameron to contribute to the paper especially to evaluate the performance of the algorithm.



Alice:

Introduction
Active
Literature Review
Active
Prop. Algorithm
Active
Conclusion
Passive

Bob:

Introduction
Passive
Literature Review
Passive
Prop. Algorithm
Active
Conclusion
Active

Figure 7-10 Alice's and Bob's sites after the second meeting

After the second meeting, Alice and Bob are working individually on the active sections on their own devices. During this period, as agreed, Alice adds a section about the performance of the algorithm (section 6.5.5) and she invites Cameron to contribute to the paper as Cameron has some expertise in this area. When Cameron agrees, Cameron joins in the collaboration session via Alice's site and, using the late-join algorithm in Chapter 5, Cameron will receive the most updated document sections and their states from Alice. At this point, Cameron has joined in the collaboration and Bob's site is not aware of this (although Bob has agreed that Alice will invite Cameron into the collaboration). Before they departed, Cameron decides that he will focus only on the performance analysis section, therefore leaving the other sections passive.

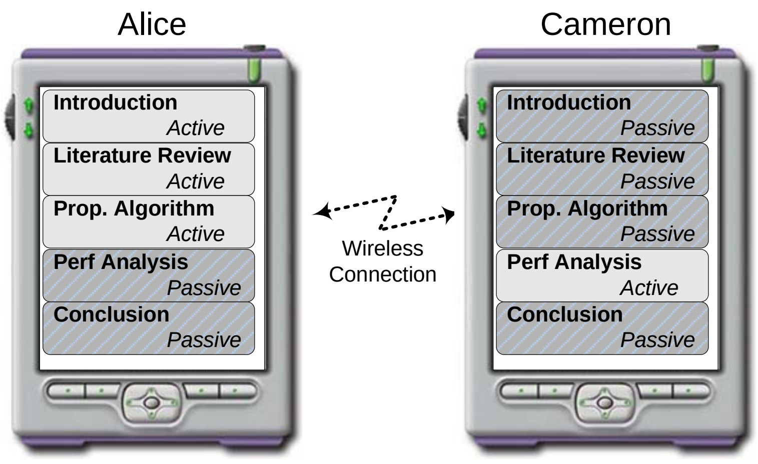


Figure 7-11 Alice's and Cameron's sites after Cameron joins

Sometime later, they all meet together for the third meeting. To start the meeting, they synchronise their documents. During the synchronisation period:

- Alice retrieves the performance analysis section from Cameron and retrieves the conclusion analysis from Bob,
- Alice and Bob merge the proposed algorithm section with the changes made by each of them,
- Bob retrieves the introduction and literature review from Alice,
- Bob and Alice retrieve the performance analysis section from Cameron, and
- Cameron retrieves all other sections from Alice and Bob accordingly.

The procedure of a site retrieving an updated state of a section is described in Chapter 6, particularly section 6.5.3. The process of merging changes of a section by different users involve: (1) the requesting and sending the missing operations described in section 5.4.3,

(2) the consistency maintenance algorithm described in Chapter 3 to ensure all operations are executed consistently and the conflict management mechanism described in Chapter 4 to handle conflicts that arise during the merging process. Figure 7-12 illustrates the three sites being in the same state after the synchronisation process.

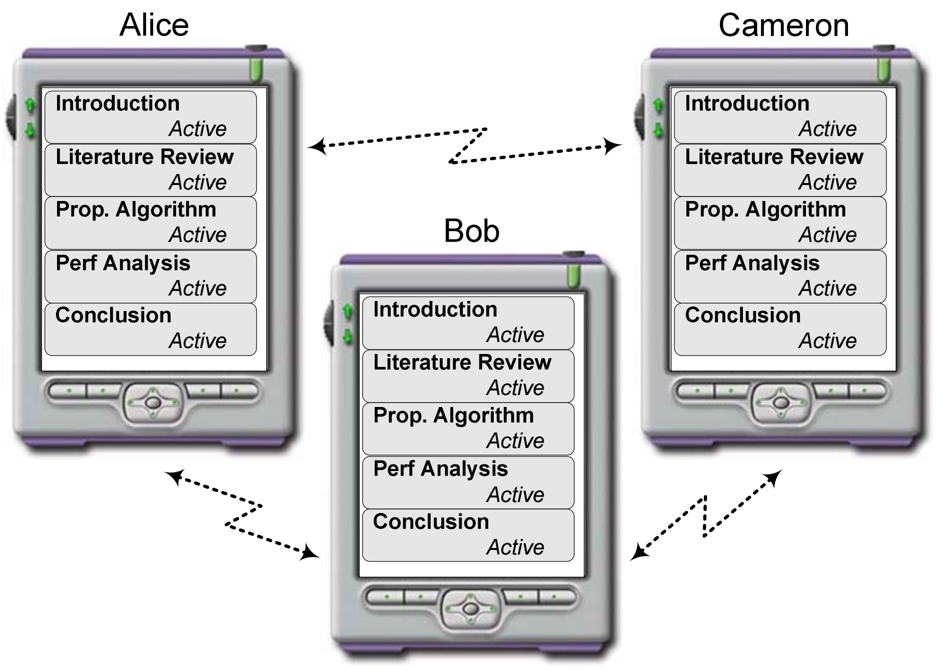


Figure 7-12 The states of the three participants at the third meeting

During the meeting, Alice, Bob and Cameron update the document concurrently. As mentioned above, the consistency management algorithm and the conflict management algorithm ensure the operations are applied consistently at all sites and all conflicts are handled and resolved correctly.

7.4. Conclusion

This chapter has presented the system architecture of the proposed collaboration framework. It combines all proposed algorithms into a cohesive framework to support the major functions of a collaborative editing application. Class diagrams have been presented and they provide basic implementation of a collaborative editor which can be extended to build collaborative editor in a specific application domain. A sample scenario has also been elaborated to show how the framework can support a real-time collaboration. Future work may include an actual implementation of the proposed framework in a specific real-life application with a comprehensive assessment to see the effectiveness and the efficiency of the cohesive framework in.

Chapter 8

Summary and Conclusion

This thesis has discussed and proposed algorithms that support collaborative editing in limited and constrained mobile environments. In particular, this thesis has described the framework for supporting real time collaborative editing in a mobile replicated architecture. A replicated architecture is suited to many mobile collaborative editing situations since it does not require a dedicated server in order for the collaboration session to work. It allows mobile users to collaborate in quickly formed networks when there is only basic communication infrastructure and nothing more than their devices available. Collaboration in a replicated architecture, however, faces challenges that are not present in a centralised architecture. The major contributions of this thesis are addressing the challenges of consistency management, conflict management, the dynamics of membership management, and partitioning work for mobile devices with limited connectivity, memory and processing power.

Firstly, this thesis has dealt with consistency management. In a replicated architecture, each mobile device holds a replica of the document since there is no dedicated server that holds the shared document. To promote concurrency, all sites can update the shared document any time and every update made by one site is immediately broadcast to all other participants. In a centralised architecture, concurrent updates are easily handled since a dedicated server receives all the updates and is able to apply the updates consistently. In a replicated architecture, however, concurrent updates have to be handled individually by each site so as to maintain the consistency of the document replicas held by the participants. Updates may arrive in different order at all sites, and the state of the origin site at which an operation is generated may not be the same as the state of the destination site to which the operation arrives. Simply totally ordering the operations at all site does not necessarily result in consistent replicas. To address and solve this challenge, a document consistency management method was proposed in Chapter 3 to ensure the consistency of the document replicas in the midst of the concurrent operations. The proposed algorithm does not involve locking, hence it promotes concurrency and all sites can update the document any time. The proposed algorithm is based on the operational transformation technique and it ensures the consistency of document replicas regardless of the arrival order of the operations. To improve its efficiency, it incorporates an existing *history trimming* technique and a novel *partial history copying* technique to reduce the storage consumption. The proposed algorithm also incorporates corrections to the existing techniques, particularly in dealing with *swapping operations* and *duplicate operations*, to ensure that consistency is maintained in all scenarios. Furthermore, an empirical study has been conducted to demonstrate the efficiency of the proposed technique and to determine the most efficient implementation of the proposed algorithm given a range of implementation environments.

Secondly, the proposed consistency management algorithm not only ensures document consistency in the midst of concurrent updates, it also deals with conflicting concurrent updates. Conflict can occur when two or more users concurrently update the same object with different intentions. In replicated architecture, however, handling and resolving conflicts is not as easy as in a centralised architecture. In a mobile network, due to fluctuating network delay, each update may not arrive at all sites in the same order. Furthermore, due to packet loss and/or sudden disconnection, it may not even arrive at some sites. Not only do conflicts have to be handled and resolved consistently, conflicts have to be handled and resolved despite not all sites having received all updates. Chapter 4 introduces a means of handling conflicts in a real-time collaboration session and facilitates users in resolving the conflict. Unlike the existing work, the proposed algorithm utilises a novel *user intention lock* to take into account the completeness of each user's intention before the conflict can be appropriately handled. It also uses a conflict table to store conflict information to facilitate users in resolving conflicts by giving them adequate information on the status of the conflict. It allows users to resolve an exclusive conflict without requiring all sites to have received all conflicting operations. While the proposed conflict management algorithm is generic and it can be used with any conflict resolution strategy such as voting or leader's decision, Chapter 4 also proposes a conflict resolution strategy that is suited to mobile replicated architecture since it does not require a group leader and requires fewer message transfers than a voting strategy.

Thirdly, the implementation of collaboration in a mobile replicated architecture has to take into account the hostile characteristics of mobile networks. Fluctuating bandwidth, sudden disconnection, voluntary and involuntary disconnection, and arbitrary leaving and joining characterise a mobile network. In order for the collaborative session to work well in a mobile replicated architecture, it has to handle such membership events. The membership management algorithm introduced in Chapter 5 allows the session to continue smoothly despite changes to site connectivity and membership. The proposed membership management algorithm is built on top of the consistency and conflict management algorithm such that it handles various membership events while still ensuring document consistency and handling conflict accordingly. With the use of the proposed *state map* technique, the algorithm ensures that all sites will receive all operations exchanged during the session and it ensures that the sites that have joined the session late will be brought up to date so they can fully participate in the collaboration session. The empirical study has shown that the algorithm is able to handle the membership events without consuming significant additional resources making it suitable to be used in limited capacity mobile networks.

Fourthly, the size of the collaborative document may be too large for a mobile device. A mobile device has a limited display and memory capability such that it may not be able to present or even hold the complete document in its device. Not all users are interested in all parts of the document and a user may choose to work only on selected parts of the document. Furthermore, with each user holding a complete document replica, all updates by all users must be sent to all sites even though some users may not need to receive updates on some parts of the document. To address this challenge, Chapter 6 introduces a document partitioning algorithm that allows the shared document to be divided into sections. This provides flexibility for users to choose only the sections they are interested in and, consequently reduces the bandwidth and resource consumption since not every site needs to receive all operations. With the use of the membership algorithm proposed in Chapter 5, users can choose to be active on a particular section and they can choose to leave a particular section when they want to. The partitioning of the shared document also allows a mobile device with limited display capacity to participate in the collaboration session by only selecting some sections to the limit of its display or storage capability. The theoretical and empirical performance evaluations have shown that document partitioning reduces resource consumption (storage and/or memory space, and bandwidth). The proposed algorithm is built on top of the previous algorithm such that it takes into account the document consistency, the handling of conflicts and the handling of membership events.

Finally, in Chapter 7, the system architecture and the class diagrams are presented to provide a basic and generic implementation of the framework comprising the proposed algorithms that can be extended to build a collaborative editor in a specific application domain. An example of the collaboration scenario is also presented in Chapter 7 giving an overview of how the proposed framework can support real time collaborative editing.

Although the constructed framework supports important functions/aspects of real time collaboration, it however does not address some issues that are important to real-time mobile collaboration. Firstly, it does not discuss the collaboration session discovery protocol. It assumes that, using an existing technique, a site that wants to join in a collaboration session knows in advance that there is a session currently running. The site joins in the session by first contacting a current session participant. Secondly, it does not discuss the technicalities of mobile network and data communications. It assumes the use of an existing mobile ad-hoc network and it assumes that the mobile sites are able to communicate with each other with the available wireless medium. Thirdly, it does not discuss all kinds of possible collaborative editing applications. It uses a simple notion of an object-based editor and it aims to be adaptable to other object-based applications. The transformation rules proposed in Chapter 3 are only applicable to text editors. Additional or new transformation rules need to be defined for different application domains.

There are also many possible ways of extending this research. Although this research has discussed the applicability of the proposed algorithms in hierarchical and graphical documents to some extent, a future work may involve an in-depth discussion and investigation on how to best support collaboration in such documents. Future work that has been identified in this thesis include the following: (1) investigating and evaluating of TTF and COT consistency maintenance algorithms and the possibility of adopting their concepts in the proposed document consistency management algorithm; (2) exploring alternative conflict resolution strategies focusing on their usability, performance and impact in a mobile environment; (3) optimising the proposed membership management algorithm in order to minimise the number of sent messages in the event of disconnections; and (4) extending the document partitioning algorithm to allow dynamic and unpredefined section creation such as splitting a section into several sections and merging two or more sections.

Finally, this research has been an effort to push the boundary of real time collaboration so that users of small mobile devices working in wireless environments can use applications that have only been possible in large capacity devices connected by high capacity, stable wired networks. In the future, users will expect to and be able to collaborate using their mobile devices in a limited and constrained mobile network environment without an established network infrastructure other than the wireless transmitters of their devices. This research has addressed some of the key issues in developing new generations of such applications that can be adapted to the changing information technology and communication landscape.

Bibliography

[1] *GoToMeeting*, http://www.gotomeeting.com/

[2] *Microsoft NetMeeting*, http://www.microsoft.com/netmeeting/

[3] *Project JXTA*, http://www.jxta.org/

[4] *RFC 1094 (rfc1094) - NFS: Network File System Protocol specification*, http://www.faqs.org/rfcs/rfc1094.html

[5] *ShowMe SharedApp*, http://www.sun.com/products-n-solutions/sw/ShowMe/products/ShowMe_SharedApp.html

[6] *WebArrow*, www.webarrow.com

[7] *WebEx*, http://www.webex.com/

[8] *World of Warcraft*, http://www.worldofwarcraft.com/

[9] H. M. Abdel-Wahab and M. A. Feit, *XTV: a framework for sharing X Window clients in remote synchronous collaboration*, In *Proceedings of IEEE TRICOMM* pp. 159-167, 1991

[10] Y. Amir, D. Dolev, S. Kramer and D. Malki, *Membership Algorithms for Multicast Communication Groups*, In *Workshop on Distributed Algorithms*, pp. 292-312, 1992

[11] Ö. Babaoğlu, R. Davoli, L.-A. Giachini and M. G. Baker, *RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems*, In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, pp. 612-621. IEEE Computer Society, 1995

[12] Ö. Babaoğlu, R. Davoli and A. Montresor, *Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications*, Operating Systems Review, 31(2): 11-22, 1997

[13] J. E. Bardram, T. R. Hansen and M. Soegaard, *AwareMedia: a shared interactive display supporting social, temporal, and spatial awareness in surgery*, In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 109 - 118, Banff, Alberta, Canada. ACM Press, 2006

[14] M. Beaudouin-Lafon and A. Karsenty, *Transparency and awareness in a real-time groupware system*, In *Proceedings of the 5th annual ACM symposium on User interface software and technology*, pp. 171 - 180, Monteray, California, United States. ACM Press, 1992

[15] J. Begole, M. B. Rosson and C. A. Shaffer, *Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems*, ACM Transactions on Computer-Human Interaction (TOCHI), 6(2): 95-132, 1999

[16] J. Begole, C. A. Struble and C. A. Shaffer, *Leveraging Java applets: toward collaboration transparency in Java*, Internet Computing, IEEE, 1(2): 57-64, 1997

[17] Y. W. Bernier, *Latency compensating methods in client/server in-game protocol design and optimization*, In *Proceedings of the 15th Games Developers Conference*, San Jose, CA, 2001

[18] R. Borovoy, F. Martin, M. Resnick and B. Silverman, *GroupWear: nametags that tell about relationships*, In *CHI 98 conference summary on Human factors in computing systems*, pp. 329-330, Los Angeles, California, United States. ACM Press, 1998

[19] R. Borovoy, F. Martin, S. Vemuri, M. Resnick, B. Silverman and C. Hancock, *Meme tags and community mirrors: moving from conferences to collaboration*, In *Proceedings of the 1998 ACM conference on Computer supported cooperative work (CSCW '98)*, pp. 159-168, Seattle, Washington, United States. ACM Press, 1998

[20] T. Brinck, *Tom's CSCW and Groupware Page*, http://www.infres.enst.fr/~vercken/multicast/cscw.html

[21] T. Brinck, *Usability First: Groupware: Introduction*, http://www.usabilityfirst.com/groupware/intro.txl

[22] D. Buszko, W. Lee and A. Helal, *Decentralized ad-hoc groupware API and framework for mobile collaboration*, In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pp. 5-14, Boulder, Colorado, USA. ACM Press, 2001

[23] C. Butcher and B. House, *Recreating the LAN party online: The networking and social infrastructure of Halo 2*, In *Proceedings of the Games Developers Conference (GDC 2005)*, 2005

[24] M. Caporuscio, A. Carzaniga and A. L. Wolf, *An experience in evaluating publish/subscribe services in a wireless network*, In *Third International Workshop on Software and Performance*, pp. 128-133, Rome, Italy, 2002

[25] M. Caporuscio and P. Inverardi, *Yet another framework for supporting mobile and collaborative work*, In *Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 81-86, 2003

[26] A. Carzaniga, D. S. Rosenblum and A. L. Wolf, *Design and evaluation of a wide-area event notification service*, ACM Transactions on Computer Systems (TOCS), 19(3): 332-383, 2001

[27] A. Chandler and J. Finney, *On the effects of loose causal consistency in mobile multiplayer games*, In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pp. 1-11, Hawthorne, NY. ACM Press, 2005

[28] A. Chandler and J. Finney, *Rendezvous: supporting real-time collaborative mobile gaming in high latency environments*, In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology (ACE '05)*, pp. 310-313, Valencia, Spain. ACM Press, 2005

[29] T. D. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost, *On the impossibility of group membership*, In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96)*, pp. 322-330, Philadelphia, Pennsylvania, United States. ACM Press, 1996

[30] K. H. Chang, Y. Gong, T. Dollar, S. Gajiwala, B. Lee and A. W. Wear, *On computer supported collaborative writing tools for distributed environments*, In *Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pp. 222-229, Nashville, Tennessee, United States. ACM Press, 1995

[31] D. Chen and C. Sun, *A distributed algorithm for graphic objects replication in real-time group editors*, In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pp. 121-130, Phoenix, Arizona, United States. ACM Press, 1999

[32] D. Chen and C. Sun, *Optional and responsive locking in collaborative graphics editing systems*, ACM SIGGROUP Bulletin, 20(3): 17-20, 1999

[33] J. Chu-Carroll and S. Carberry, *Response generation in collaborative negotiation*, In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pp. 136-143, Cambridge, Massachusetts. Association for Computational Linguistics, 1995

[34] S. Citro, J. McGovern and C. Ryan, *An efficient consistency management algorithm for real-time mobile collaboration*, In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, pp. 287-294, 2005

[35] M. Cohen, R. Fish, R. Kraut and M. Leland, *Quilt: A Collaborative Tool for Cooperative Writing*, In *Proceedings of ACM SIGOIS Conference*, pp. 30-37, 1988

[36] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems Concepts and Design*, Addison Wesley, 2001.

[37] A. H. Davis, C. Sun and J. Lu, *Generalizing operational transformation to the standard general markup language*, In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pp. 58-67, New Orleans, Louisiana, USA. ACM Press, 2002

[38] D. Decouchant, V. Quint and M. Romero, *Structured and distributed cooperative editing in large scale network*, in R. Rada, ed., *Groupware and Authoring*, Academic Press, London, 1996, pp. 265-296.

[39] P. Dewan, *Architectures for collaborative applications*, in M. Beaudouin-Lafon, ed., *Computer Supported Cooperative Work*, John Wiley & Sons, 1999, pp. 169-193.

[40] D. Dolev, S. Kramer and D. Malki, *Early delivery totally ordered multicast in asynchronous environment*, In *Proceedings of the 23th Annual International Symposium on Fault-Tolerant Computing*, pp. 544-553, Toulouse, June 1993

[41] D. Dolev, D. Malki and R. Strong, *An asynchronous membership protocol that tolerates partitions*, Technical Report CS94-6, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994

[42] D. Dolev, D. Malki and R. Strong, *A framework for partitionable membership service*, Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995

[43] P. Dourish, *Software infrastructures*, in M. Beaudouin-Lafon, ed., *Computer Supported Cooperative Work*, John Wiley & Sons, Chichester, 1999, pp. 195-219.

[44] W. K. Edwards, *Session management for collaborative applications*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work (CSCW '94)*, pp. 323-330, Chapel Hill, North Carolina, United States. ACM Press, 1994

[45] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, D. Balfanz, D. K. Smetters, H. C. Wong and S. Izadi, *Using speakeasy for ad hoc peer-to-peer collaboration*, In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pp. 256-265, New Orleans, Louisiana, USA. ACM Press, 2002

[46] C. A. Ellis and S. J. Gibbs, *Concurrency control in groupware systems*, In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pp. 399-407, Portland, Oregon, United States. ACM Press, 1989

[47] C. A. Ellis, S. J. Gibbs and G. L. Rein, *Design and Use of a Group Editor*, In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pp. 13-28, Napa Valley, California. Elsevier, 1989

[48] M. Esbjörnsson, O. Juhlin and M. Östergen, *Motorcycling and social interaction: design for the enjoyment of brief traffic encounters*, In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work (GROUP '03)*, pp. 85-94, Sanibel Island, Florida, USA. ACM Press, 2003

[49] M. Esbjörnsson and M. Östergren, *Hocman: supporting mobile group collaboration*, In *CHI '02 extended abstracts on Human factors in computing systems*, pp. 838-839, Minneapolis, Minnesota, USA. ACM Press, 2002

[50] P. D. Ezhilchelvan, R. A. Macedo and S. K. Shrivastava, *Newtop: a fault-tolerant group communication protocol*, In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pp. 296-306, Washington, DC, USA. IEEE Computer Society, 1995

[51] J. Falk, P. Ljungstrand, S. Björk and R. Hansson, *Pirates: proximity-triggered interaction in a multi-player game*, In *CHI '01 extended abstracts on Human factors in computing systems*, pp. 119-120, Seattle, Washington. ACM Press, 2001

[52] B. Federico, P. Agostino and S. Matteo, *A collaborative platform for fixed and mobile networks*, Commun. ACM, 45(11): 39-44, 2002

[53] M. J. Fischer, N. A. Lynch and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM (JACM), 32(2): 374-382, 1985

[54] D. Garfinkel, B. Welti and T. Yip., *HP SharedX: A tool for real-time collaboration*, HP Journal: 23-36, April 1994

[55] S. Greenberg and D. Marwood, *Real time groupware as a distributed system: concurrency control and its effect on the interface*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pp. 207-217, Chapel Hill, North Carolina, United States. ACM Press, 1994

[56] S. Greenberg and M. Roseman, *Groupware toolkits for synchronous work*, in M. Beaudouin-Lafon, ed., *Computer Supported Cooperative Work*, John Wiley & Sons, Chichester, 1999, pp. 135-168.

[57] S. Greenberg, M. Roseman and D. Webster, *Issues and experiences designing and implementing two group drawing tools*, In *Proceedings of the 25th Annual Hawaii Intemational Conference on the System Sciences*, pp. 139-250, 1992

[58] J. Grudin, *Computer-Supported Cooperative Work*, IEEE Computer, 27(5): 19-26, 1994

[59] R. Guerraoui and C. Hari, *On the consistency problem in mobile distributed computing*, In *Proceedings of the second ACM international workshop on Principles of mobile computing*, pp. 51-57, Toulouse, France. ACM Press, 2002

[60] C. Gutwin and S. Greenberg, *The effects of workspace awareness support on the usability of real-time distributed groupware*, ACM Transactions on Computer-Human Interaction (TOCHI), 6(3): 243-281, 1999

[61] C. Gutwin, S. Greenberg and M. Roseman, *Supporting awareness of others in groupware*, In *Conference companion on Human factors in computing systems: common ground*, pp. 205, Vancouver, British Columbia, Canada. ACM Press, 1996

[62] H. He, Q. Wu and L. Luo, *Document marking scheme for preserving intention of operation in cooperative editing system*, Journal of Software, 10(2): 160-164, 1999

[63] J. Hill and C. Gutwin, *Awareness support in a groupware widget toolkit*, In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pp. 258-267, Sanibel Island, Florida, USA. ACM Press, 2003

[64] L. E. Holmquist, J. Falk and J. Wigström, *Supporting Group Collaboration with Interpersonal Awareness Devices*, Personal and Ubiquitous Computing, 3: 13-21, 1999

[65] C.-L. Ignat and M. C. Norrie, *Draw-together: graphical editor for collaborative drawing*, In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 269-278, Banff, Alberta, Canada. ACM Press, 2006

[66] C. Ignat, M. Norrie and G. Oster, *Handling Conflicts through Multi-level Editing in Peer-to-peer Environments*, In *International Workshop on Collaborative Editing Systems - CEW 2006*, Banff, Alberta, Canada, 2006

[67] C. L. Ignat and M. C. Norrie, *Customizable Collaborative Editor Relying on treeOPT Algorithm*, In *Proceedings of ECSCW'03*, pp. 315-334, Helsinki, Finland, 2003

[68] C. L. Ignat and M. C. Norrie, *Flexible Definition and Resolution of Conflicts through Multi-level Editing*, In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2006*, Georgia, Atlanta, USA, November 2006

[69] E. A. Isaacs, J. C. Tang and T. Morris, *Piazza: a desktop environment supporting impromptu and planned interactions*, In *Proceedings of the 1996 ACM conference on Computer supported cooperative work (CSCW '96)*, pp. 315-324, Boston, Massachusetts, United States. ACM Press, 1996

[70] ISO/IEC, *FDIS 9126-1 Software Engineering - Product Quality - Part 1: Quality Model*, November 1999.

[71] F. Jahanian, S. Fakhouri and R. Rajkumar, *Processor group membership protocols: specification, design and implementation*, In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pp. 2-11, Princeton, October 1993

[72] R. Kanawati, *LICRA: A replicated-data management algorithm for distributed synchronous groupware applications*, Parallel Computing, 22(13): 1733-1746, 1997

[73] A. Karsenty and M. Beaudouin-Lafon, *An algorithm for distributed groupware applications*, In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pp. 195-202, 1993

[74] A. Karsenty, C. Tronche and M. Beaudouinlafon, *GroupDesign: shared editing in a heterogeneous environment*, Usenix Journal of Computing Systems, 6(2): 167-195, 1993

[75] I. Keidar, J. Sussman, K. Marzullo and D. Dolev, *Moshe: A group membership service for WANs*, ACM Transactions on Computer Systems, 20(3): 191-238, 2002

[76] R. I. Khazan, *Group membership: a novel approach and the first single-round algorithm*, In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing (PODC '04)*, pp. 347-356, St. John's, Newfoundland, Canada. ACM Press, 2004

[77] E. Kirda, P. Fenkam, G. Reif and H. Gall., *A service architecture for mobile teamwork*, In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pp. 513-518, Ischia, Italy, 2002

[78] G. Kortuem, *Proem: a middleware platform for mobile peer-to-peer computing*, ACM SIGMOBILE Mobile Computing and Communications Review, 6(4): 62-64, 2002

[79] S. Kristoffersen and F. Ljungberg, *An empirical study of how people establish interaction: implications for CSCW session management models*, In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '99)*, pp. 1-8, Pittsburgh, Pennsylvania, United States. ACM Press, 1999

[80] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM, 21(7): 558-565, 1978

[81] E. d. Lara, R. Kumar, D. S. Wallach and W. Zwaenepoel, *Collaboration and multimedia authoring on mobile devices*, In *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, pp. 287-301, San Francisco, California. ACM Press, 2003

[82] E. d. Lara, D. Wallach and W. Zwaenepoel, *Puppeteer: component-based adaptation for mobile computing*, In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, pp. 159-170, San Francisco, California, USA, 2001

[83] D. Li and R. Li, *Ensuring content and intention consistency in real-time group editors*, In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 748-755, 2004

[84] D. Li and R. Li, *Preserving operation effects relation in group editors*, In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pp. 457-466, Chicago, Illinois, USA. ACM Press, 2004

[85] R. Li and D. Li, *A landmark-based transformation approach to concurrency control in group editors*, In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pp. 284-293, Sanibel Island, Florida, USA. ACM Press, 2005

[86] R. Li, D. Li and C. Sun, *A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications*, In *Proceedings of the Parallel and Distributed Systems, Tenth International Conference on (ICPADS'04)* pp. 429-436. IEEE Computer Society, 2004

[87] C. Ling, *An adaptive consistency maintenance approach for replicated continuous applications*, In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pp. 795-801 2005

[88] B. Lubachevsky, A. Schwartz and A. Weiss, *An analysis of rollback-based simulation*, ACM Transactions on Modeling and Computer Simulation (TOMACS), 1(2): 154-193, 1991

[89] J. Ma, M. Shizuka, J. Lee and R. Huang, *A P2P groupware system with decentralized topology for supporting synchronous collaborations*, In *Proceedings of the 2003 International conference on Cyberworlds (CW'03)*, pp. 54-61, 2003

[90] F. Mattern, *Virtual time and global states of distributed systems*, In *Proceedings of the International Workshop on Parallel and Distributed Algorithms,*, pp. 215–276. Elsevier Pub., 1989

[91] M. Mauve, *Consistency in replicated continuous interactive media*, In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pp. 181-190, Philadelphia, Pennsylvania, United States. ACM Press, 2000

[92] M. Mauve, J. Vogel, V. Hilt and W. Effelsberg, *Local-lag and timewarp: providing consistency for replicated continuous applications*, IEEE Transactions on Multimedia, 6(1): 47-57, 2004

[93] J. McGovern and C. Ryan, *Adaptive Consistency Management Support for Limited Capacity Devices in Ad-hoc Mobile Networks*, In *Proceedings of 2004 International Symposium on Collaborative Technologies and Systems*, pp. 250-256, San Diego, USA, 2004

[94] L. J. McGuffin and G. M. Olson, *ShrEdit: a shared electronic workspace*, Technical Report 45, The University of Michigan, 1992

[95] I. Mohomed, J. C. Cai, S. Chavoshi and E. d. Lara, *Context-aware interactive content adaptation*, In *Proceedings of the 4th international conference on Mobile systems, applications and services (MobiSys 2006)*, pp. 42-55, Uppsala, Sweden. ACM Press, 2006

[96] T. P. Moran, K. McCall, B. v. Melle, E. R. Pedersen and F. G. H. Halasz, *Some design principles for sharing in Tivoli, a whiteboard meeting support tool*, in S. Greenberg, S. Hayne and R. Rada, eds., *Groupware for Realtime Drawing: A Designer's guide*, McGraw-Hill, 1995, pp. 24-36.

[97] J. Munson and P. Dewan, *A concurrency control framework for collaborative systems*, In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pp. 278-287, Boston, Massachusetts, United States. ACM Press, 1996

[98] J. P. Munson and P. Dewan, *Sync: a Java framework for mobile collaborative applications*, IEEE Computer, 30(6): 59-66, 1997

[99] G. Neiger, *A new look at membership services (extended abstract)*, In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96)*, pp. 331-340, Philadelphia, Pennsylvania, United States. ACM Press, 1996

[100] R. E. Newman-Wolfe, M. L. Webb and M. Montes, *Implicit locking in the ensemble concurrent object-oriented graphics editor*, In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pp. 265-272, Toronto, Ontario, Canada. ACM Press, 1992

[101] D. A. Nichols, P. Curtis, M. Dixon and J. Lamping, *High-latency, low-bandwidth windowing in the Jupiter collaboration system*, In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pp. 111-120, Pittsburgh, Pennsylvania, United States. ACM Press, 1995

[102] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn and K. R. Walker, *Agile application-aware adaptation for mobility*, In *Proceedings of the sixteenth ACM symposium on operating systems principles (SOSP '97)*, pp. 276-287, Saint Malo, France. ACM Press}, 1997

[103] J. F. Nunamaker, *Collaborative Computing: The Next Millennium*, Computer: 66-71, 1999

[104] G. Oster, P. Molli, P. Urso and A. Imine, *Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems*, In *Proceedings of The 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Atlanta, Georgia, 2006

[105] G. Oster, P. Urso, P. Molli and A. Imine, *Data consistency for P2P collaborative editing*, In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 259-268, Banff, Alberta, Canada. ACM Press, 2006

[106] F. Pacull, A. Sandoz and A. Schiper, *Duplex: a distributed collaborative editing environment in large scale*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work (CSCW '94)*, pp. 165-173, Chapel Hill, North Carolina, United States. ACM Press, 1994

[107] L. Pantel and L. C. Wolf, *On the impact of delay on real-time multiplayer games*, In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pp. 23-29, Miami, Florida, USA. ACM Press, 2002

[108] J. F. Patterson, R. D. Hill, S. L. Rohall and S. W. Meeks, *Rendezvous: an architecture for synchronous multi-user applications*, In *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pp. 317-328, Los Angeles, California, United States. ACM Press, 1990

[109] M. O. Pendergast, *GroupGraphics: prototype to product*, in S. Greenberg, S. Hayne and R. Rada, eds., *Groupware for Real-time Drawing: A Designer's guide*, McGraw-Hill, 1995, pp. 209-227.

[110] W. G. Phillips, *Quality Analysis of Distribution Architectures for Synchronous Groupware* In *Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing* Atlanta, Georgia, USA. IEEE Computer Society, 2006

[111] A. Prakash, *Group editors*, in M. Beaudouin-Lafon, ed., *Computer Supported Cooperative Work*, John Wiley & Sons, 1999, pp. 103-133.

[112] A. Prakash and M. J. Knister, *A framework for undoing actions in collaborative systems*, ACM Transactions on Computer-Human Interaction (TOCHI), 1(4): 295-330, 1994

[113] A. Prakash and H. S. Shim, *DistView: support for building efficient collaborative applications using replicated objects*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pp. 153-164, Chapel Hill, North Carolina, United States. ACM Press, 1994

[114] G. K. Raikundalia and H. Zhang, *Newly-discovered group awareness mechanisms for supporting real-time collaborative authoring*, In *Proceedings of the Sixth Australasian conference on User interface - Volume 40*, pp. 127-136, Newcastle, Australia. Australian Computer Society, Inc., 2005

[115] M. Ressel, D. Nitsche-Ruhland and R. Gunzenhäuser, *An integrating, transformation-oriented approach to concurrency control and undo in group editors*, In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pp. 288-297, Boston, Massachusetts, United States. ACM Press, 1996

[116] A. M. Ricciardi and K. P. Birman, *Using process groups to implement failure detection in asynchronous environments*, In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing (PODC '91)*, pp. 341-353, Montreal, Quebec, Canada. ACM Press, 1991

[117] G. C. Roman, Q. Huang and A. Hazemi, *Consistent group membership in ad hoc networks*, In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pp. 381-388, May 2001

[118] M. Roseman and S. Greenberg, *Building real-time groupware with GroupKit, a groupware toolkit*, ACM Transactions on Computer-Human Interaction (TOCHI), 3(1): 66-106, 1996

[119] M. Roseman and S. Greenberg, *GROUPKIT: a groupware toolkit for building real-time conferencing applications*, In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pp. 43-50, Toronto, Ontario, Canada. ACM Press, 1992

[120] J. Roth and C. Unger, *Using Handheld Devices in Synchronous Collaborative Scenarios*, Personal and Ubiquitous Computing, 5(4): 243-252, 2001

[121] A. Schiper and A. Sandoz, *Primary Partition Virtually-Synchronous Communication harder than Consensus*, In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG-8)*, pp. 39-52, Terschelling, The Netherlands. Springer-Verlag, September 1994

[122] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning and D. Tatar, *WYSIWIS revised: early experiences with multiuser interfaces*, ACM Transactions on Information Systems (TOIS), 5(2): 147-167, 1987

[123] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning and L. Suchman, *Beyond the chalkboard: computer support for collaboration and problem solving in meetings*, Communications of the ACM, 30(1): 32-47, 1987

[124] N. A. Streitz, J. Geißler, J. M. Haake and J. Hol, *DOLPHIN: integrated meeting support across local and remote desktop environments and LiveBoards*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, Chapel Hill, North Carolina, United States. ACM Press, 1994

[125] N. A. Streitz, J. Geißler, J. M. Haake and H. Jeroen, *DOLPHIN: integrated meeting support across local and remote desktop environments and LiveBoards*, In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pp. 345-358, Chapel Hill, North Carolina, United States. ACM Press, 1994

[126] M. Suleiman, M. Cart and J. Ferrié, *Serialization of concurrent operations in a distributed collaborative environment*, In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pp. 435-445, Phoenix, Arizona, United States. ACM Press, 1997

[127] C. Sun, *Optional and Responsive Fine-Grain Locking in Internet-Based Collaborative Systems*, IEEE Transactions on Parallel and Distributed Systems, 13(9): 994-1008, 2002

[128] C. Sun, *Undo as concurrent inverse in group editors*, ACM Transactions on Computer-Human Interaction (TOCHI) 9(4): 309-361, 2002

[129] C. Sun and W. Cai, *Capturing causality by compressed vector clock in real-time group editors*, In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pp. 59–66, April 2002

[130] C. Sun and D. Chen, *Consistency maintenance in real-time collaborative graphics editing systems*, ACM Transactions on Computer-Human Interaction (TOCHI), 9(1): 1-41, 2002

[131] C. Sun and C. Ellis, *Operational transformation in real-time group editors: issues, algorithms, and achievements*, In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pp. 59-68, Seattle, Washington, United States. ACM Press, 1998

[132] C. Sun, X. Jia, Y. Zhang, Y. Yang and D. Chen, *Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems*, ACM Transactions on Computer-Human Interaction (TOCHI), 5(1): 63-108, 1998

[133] C. Sun and R. Sosič, *Optimal locking integrated with operational transformation in distributed real-time group editors*, In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pp. 43-52, Atlanta, Georgia, United States. ACM Press, 1999

[134] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen and W. Cai, *Transparent adaptation of single-user applications for multi-user real-time collaboration*, ACM Transactions on Computer-Human Interaction (TOCHI), 13(4): 531 - 582, 2006

[135] C. Sun, Y. Yang, Y. Zhang and D. Chen, *A Consistency Model and Supporting Schemes for Real-time Cooperative Editing Systems*, In *Proceedings of the 19th Australian Computer Science Conference*, pp. 582-591, Melbourne, Australia, 1996

[136] C. Sun, Y. Yang, Y. Zhang and D. Chen, *Distributed Concurrency Control in Real-time Cooperative Editing Systems*, In *Proc. of the 1996 Asian Computing Science Conference*, pp. 84-95, Singapore. Springer-Verlag, 1996

[137] C. Sun, Y. Zhang, X. Jia and Y. Yang, *A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems*, In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pp. 425-434, Phoenix, Arizona, United States. ACM Press, 1997

[138] D. Sun and C. Sun, *Operation context and context-based operational transformation*, In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 279-288, Banff, Alberta, Canada. ACM Press, 2006

[139] K. Tee, S. Greenberg and C. Gutwin, *Providing artifact awareness to a distributed group through screen sharing*, In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pp. 99-108, Banff, Alberta, Canada. ACM Press, 2006

[140] M. H. Tran, Y. Yang and G. K. Raikundalia, *Extended radar view and modification director: awareness mechanisms for synchronous collaborative authoring*, In *Proceedings of the 7th Australasian User interface conference - Volume 50*, pp. 45-52, Hobart, Australia. Australian Computer Society, Inc., 2006

[141] L. Veiga and P. Ferreira, *Semantic-Chunks a middleware for ubiquitous cooperative work*, In *Proceedings of the 4th workshop on Reflective and adaptive middleware systems (ARM '05)*, Grenoble, France. ACM Press, 2005

[142] N. Vidot, M. Cart, J. Ferrié and M. Suleiman, *Copies convergence in a distributed real-time collaborative environment*, In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pp. 171-180, Philadelphia, Pennsylvania, United States. ACM Press, 2000

[143] J. Vogel and M. Mauve, *Consistency control for distributed interactive media*, In *Proceedings of the ninth ACM international conference on Multimedia*, pp. 221-230, Ottawa, Canada. ACM Press, 2001

[144] V. von Biel, *Groupware Grows Up*, MacUser: 207-211, June 1991

[145] P. Wilson, *Computer supported cooperative work : an introduction*, Kluwer Academic Publishers, Oxford, England Norwell, MA, 1991.

[146] X. Wu and N. Gu, *A concurrency control method based on document marking*, Journal of Computer Research and Development, 39(12): 1662-1667, 2002

[147] S. Xia, D. Sun, C. Sun, D. Chen and H. Shen, *Leveraging single-user applications for multi-user collaboration: the coword approach*, In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pp. 162-171, Chicago, Illinois, USA. ACM Press, 2004

[148] L. Xue, M. Orgun and K. Zhang, *A multi-versioning algorithm for intention preservation in distributed real-time group editors*, In *Proceedings of the 26th Australasian computer science conference - Volume 16*, pp. 19-28, Adelaide, Australia. Australian Computer Society, Inc., 2003

[149] L. Xue, K. Zhang and C. Sun, *Conflict Control Locking in Distributed Cooperative Graphics Editors*, In *Proceedings of the First International Conference on Web Information Systems Engineering (WISE 2000)*, pp. 401-408. IEEE Computer Society, 2000

[150] L. Xue, K. Zhang and C. Sun, *An Integrated Post-Locking, Multi-Versioning, and Transformation Scheme for Consistency Maintenance in Real-Time Group Editors*, In *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS)*, pp. 57-64, 2001

[151] G. G. Yang, *A data management System for replicated application*, In *Proceedings of the 4th international conference on Collaborative virtual environments (CVE '02)*, pp. 147-148, Bonn, Germany. ACM Press, 2002

[152] S. M. T. Yau, H. V. Leong, D. McLeod and A. Si, *On mutli-resolution document transmission in mobile Web*, ACM SIGMOD Record, 28(3): 37-42, 1999