Abstract

Synchronous distributed groupware is a class of software applications allowing a geographically distributed group of people to collaborate in real time. There are different types of groupware, e.g., collaborative editing software, distributed meeting support tools, and multiplayer games. However, collaborators in groupware can become disconnected from the session. Disconnections can range in duration from a few seconds (e.g., due to a network outage) to hours or days (e.g., stowing a laptop while flying). Disconnection causes information loss and makes it difficult for users to understand both the state of the workspace and the current activities of other people upon reconnection. Thus, it is important to handle disconnection in groupware. However, handling disconnection is difficult for groupware developers. They need to determine varieties of strategies in order to address different disconnection scenarios. These strategies determine how stored information can be manipulated as the system waits for a disconnected user to rejoin, and how information should be replayed upon reconnection. If disconnection lasts for a long time, developers need to select and combine strategies in order to manage a trade-off between performance requirements (e.g., delivering stored information as quickly as possible upon reconnection) and understandability requirements (e.g., allowing users to watch missed information in an understandable manner). Developers might not know how to implement such strategies in a reusable manner. Because of this lack of knowledge in handling disconnection, developers might build disconnection-aware groupware applications that will not address the range of wide variety of user-level requirements that arise from different disconnection scenarios. Moreover, as there are a few disconnection-aware groupware applications, developers might not know the overhead (e.g., additional message transmission time, memory usage and programming complexity) of handling disconnection in synchronous groupware in practice. In order to mitigate developers’ problems to handle disconnection in synchronous groupware, this thesis provides codified solutions that capture and organize a wide range of strategies for handling disconnection, that manage the performance and understandability trade-off through selecting and combining suitable strategies and that show how to implement the strategies in a reusable manner. In order to determine the overhead of handling disconnection, a toolkit is developed following the designed solutions, and different applications are constructed using the toolkit. User studies and performance analyses are conducted that evaluate the toolkit and demonstrate its quality goals, such as offering simple application programming interface (API), high performance, and supporting different disconnection scenarios and timeframes.
Co-Authorship

This thesis has resulted in a number of publications. Chapters 4, 6, 7, and 8 have been published as a major conference paper at CSCW 2012 [80], co-authored with T.C. Nicholas Graham and Carl Gutwin. There have also been two other papers [81, 82] and a poster [83]; these non-archival publications were co-authored with T.C. Nicholas Graham, and Carl Gutwin. Part of Chapter 4, 6, and 7 was published at Grand 2011 [81] and part of Chapter 5 was published at Grand 2012 [82].

As a background study to this thesis on software architecture two papers have been published. The first paper on software architecture recovery was published at ECSA 2008, co-authored with T.C. Nicholas Graham. The second paper on user-centered development of groupware was published at EICS 2009, co-authored with Christopher Wolfe, T.C. Nicholas Graham, and W. Greg Phillips. While both the papers are highly related and worked as a solid background to this thesis work, we included only parts of them in Chapter 6.
Acknowledgements

I would first like to express my heart-felt and most sincere gratitude to my respected co-supervisors T.C. Nicholas Graham and Carl Gutwin for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without them, this work would have been impossible.

I would like to thank Thomas R. Dean and Mohammad Zulkernine, the other two members of my Ph.D. supervisory committee for their suggestions and inspiration during the thesis work. Thanks are also due to James R. Cordy, Greg Phillips and Robert Biddle, the other members of my thesis examination committee for their helpful comments, insights and suggestions.

I am grateful to the Ontario Graduate Scholarship in Science and Technology (OGSST) program and Queen’s University for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all my lab mates both at EQUIS and HCI labs who helped me in one way or to the other along the way. Some of the lab mates whose names I cannot but mention are: Christopher Wolfe, Tad Stach, Cheryl Savery, Gregor McEwan, Andre Doucette, Scott Bateman, Dian Watson, Kathrin Gerling, David Flatla, Aaron Genest, Adrian Reetz, Andriy Pavlovych, Yudi Xue and Nelson Wolf.

I thank to my friends and well-wishers. In particular, I would like to thank Debby Robertson, Hafizur Rahman, Shahed Kalam, Nasrin Jahan, Gias Uddin, Meher Chowdhury, Atif Farid, Sara Khan, Muhammad Asaduzzaman, Sharif Uddin, Koly Chowdhury, Shamoli Saha and Rikta Parvin.

I express my gratefulness to my family members and relatives for their support and inspiration, especially, my mother Bela Rani Karmaker, my sisters Kalyani Roy (and her husband Pulak Das), Sandhya Roy (and her husband Sanjoy Kumar Saha) and Laboni Roy Karmokar (and her husband Sonjoy Karmokar). My appreciation goes for my brother-in laws Hriday Roy (and his wife Momota Roy), Ratan Roy (and his wife Mukti Proshad) and Ujjal Roy (and his wife Satyakee Roy).

iv
The most magnificent and precious gift of my life is my daughter Prisha, who came as an angel in our lives and drove me in the right direction in successfully finishing my thesis and getting a job.

Finally, I would like to thank my husband Chanchal Roy. It was his love that inspired me through to the end of this thesis work, and I am forever indebted to him.

Thanks to almighty God for giving me the courage and energy to reach at the end of this project.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

(Banani Roy)

(April, 2013)
# Table of Contents

Abstract ................................................................................................................................. ii
Co-Authorship ......................................................................................................................... iii
Acknowledgements ................................................................................................................ iv
Statement of Originality ........................................................................................................... vi

## Chapter 1 Introduction ........................................................................................................ 1

1.1 Motivation ......................................................................................................................... 4
1.2 Problem ............................................................................................................................. 5
  1.2.1 Sub-problem 1: Lack of knowledge about disconnection design space .................... 7
  1.2.2 Sub-problem 2: Managing performance and understandability trade-offs ............... 7
  1.2.3 Sub-problem 3: No simple and flexible development infrastructure exists ............. 8
  1.2.4 Sub-Problem 4: Lack of empirical evidence ......................................................... 9
1.3 Research Questions ........................................................................................................... 10
1.4 Solution ............................................................................................................................ 10
  1.4.1 Evaluation ................................................................................................................ 12
1.5 Summary .......................................................................................................................... 14
1.6 Scope of the Thesis .......................................................................................................... 14
1.7 Dissertation Outline ....................................................................................................... 15

## Chapter 2 Background and Related Research ................................................................. 17

2.1 Synchronous Distributed Groupware ............................................................................ 17
  2.1.1 Terminology .............................................................................................................. 19
    2.1.1.1 Events ............................................................................................................... 19
    2.1.1.2 Shared workspace ......................................................................................... 20
    2.1.1.3 Workspace awareness .................................................................................. 21
    2.1.1.4 Telepointers ................................................................................................... 22
  2.2 Disconnection in Synchronous Groupware ................................................................. 23
    2.2.1 Sources of disconnection ....................................................................................... 23
    2.2.2 User-level disconnection problems ....................................................................... 26
    2.2.3 Disconnection scenario ......................................................................................... 26
    2.2.4 Duration of disconnection and disconnection timeframes ..................................... 27
  2.3 Existing Solutions for Handling Disconnection in Groupware ................................... 28
    2.3.1 User-view of groupware disconnection ............................................................... 29
    2.3.2 Techniques for handling aspects of disconnection ............................................... 30
Chapter 6 Design and Specification of A Plug-in Architecture ............................................. 112

6.1 Why Plug-in Architecture? ......................................................................................... 112

6.2 Specification of Plug-in Software Architecture ......................................................... 115

6.2.1 Mediator components .............................................................................................. 117

6.2.1.1 DiscoTechClient ................................................................................................. 118

6.2.1.2 DiscoTechServer ............................................................................................... 118

6.2.2 Infrastructural Components .................................................................................. 119

6.2.2.1 Networking Manager ........................................................................................ 119

6.2.2.2 Disconnection Identifier .................................................................................... 120

6.2.2.3 Event Queue ....................................................................................................... 121

6.2.2.4 Reconnection Manager ...................................................................................... 121

6.2.3 Summary of mediator and infrastructural components ......................................... 123

6.2.4 Plug-ins .................................................................................................................. 123

6.2.4.1 Compactor Plug-ins .......................................................................................... 125

6.2.4.2 Replayer Plug-ins ............................................................................................... 126

6.2.5 Interactions among components during disconnection ........................................... 126

6.2.6 Interactions among components upon reconnection .............................................. 127

6.3 Conceptual Overlay Architectures .............................................................................. 129

6.4 Quality Goals for the Plug-in Architecture ............................................................... 132

6.5 Summary and Discussion of Plug-in Architecture ..................................................... 134

Chapter 7 Designing DiscoTech Toolkit ........................................................................... 135

7.1 Ease of Use through the DiscoTech Toolkit ................................................................. 137

7.2 Expressiveness through Plug-ins .............................................................................. 139

7.3 High Performance through Light-Weight Operation .................................................. 142

7.4 Details of Plug-in Implementation .............................................................................. 143

7.5 Using the DiscoTech Toolkit ...................................................................................... 145

7.6 Supporting Existing Disconnection Techniques Using DiscoTech .............................. 146

7.6.1 Summary of supporting existing techniques with DiscoTech ............................... 148

7.7 Supporting Disconnection Design Combos with DiscoTech ....................................... 148

7.8 Summary of Designing and Implementing DiscoTech Toolkit ..................................... 152

7.9 Developing Disconnection-aware Groupware using DiscoTech ................................. 152

7.9.1 Text chat application ............................................................................................. 155

7.9.2 Shared UML use case diagram editor .................................................................... 156

7.9.2.1 Collaborative paper reviewing application ....................................................... 157
<table>
<thead>
<tr>
<th>References</th>
<th>108x84</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 9 Conclusion and Future Work</td>
<td>108x274</td>
</tr>
<tr>
<td>9.5 Conclusion of the Thesis</td>
<td>119x103</td>
</tr>
<tr>
<td>9.4 Future Work</td>
<td>119x198</td>
</tr>
<tr>
<td>9.3 Threats to Validity</td>
<td>119x217</td>
</tr>
<tr>
<td>9.2 Contributions to Knowledge</td>
<td>119x236</td>
</tr>
<tr>
<td>9.1 Thesis Summary</td>
<td>119x255</td>
</tr>
<tr>
<td>Chapter 8 Empirical Evaluations of DiscoTech Toolkit</td>
<td>119x312</td>
</tr>
<tr>
<td>8.4 Summary and Discussion of Empirical Evaluation of DiscoTech</td>
<td>130x122</td>
</tr>
<tr>
<td>8.3 High Performance</td>
<td>130x141</td>
</tr>
<tr>
<td>8.2 Expressiveness</td>
<td>130x160</td>
</tr>
<tr>
<td>8.1 Ease of Use</td>
<td>130x179</td>
</tr>
<tr>
<td>Chapter 7 Summary and Discussion of Developing Disconnection-Aware Groupware</td>
<td>130x293</td>
</tr>
<tr>
<td>7.9.6 Summary and discussion of developing disconnection-aware groupware</td>
<td>157</td>
</tr>
<tr>
<td>7.9.5 Collaborative gesture-based game for older adults</td>
<td>158</td>
</tr>
<tr>
<td>7.9.4 Future studies</td>
<td>158</td>
</tr>
<tr>
<td>7.9.3 Tank game</td>
<td>160</td>
</tr>
<tr>
<td>7.9.2 Supporting disconnection due to network latency</td>
<td>160</td>
</tr>
<tr>
<td>7.9.1 Supporting offline work</td>
<td>161</td>
</tr>
<tr>
<td>7.9 Summary</td>
<td>162</td>
</tr>
<tr>
<td>Chapter 8 Empirical Evaluations of DiscoTech Toolkit</td>
<td>163</td>
</tr>
<tr>
<td>8.1 Ease of Use</td>
<td>164</td>
</tr>
<tr>
<td>8.1.1 Limitations</td>
<td>165</td>
</tr>
<tr>
<td>8.1.2 Summary and discussion of ease of use</td>
<td>166</td>
</tr>
<tr>
<td>8.2 Expressiveness</td>
<td>166</td>
</tr>
<tr>
<td>8.2.1 Limitations</td>
<td>167</td>
</tr>
<tr>
<td>8.2.2 Summary and discussion of expressiveness</td>
<td>167</td>
</tr>
<tr>
<td>8.3 High Performance</td>
<td>167</td>
</tr>
<tr>
<td>8.3.1 Experimental Apparatus</td>
<td>168</td>
</tr>
<tr>
<td>8.3.2 Methodology</td>
<td>169</td>
</tr>
<tr>
<td>8.3.2.1 Experiment 1</td>
<td>170</td>
</tr>
<tr>
<td>8.3.2.2 Experiment 2</td>
<td>172</td>
</tr>
<tr>
<td>8.3.2.3 Experiment 3</td>
<td>173</td>
</tr>
<tr>
<td>8.3.3 Limitations</td>
<td>175</td>
</tr>
<tr>
<td>8.3.4 Summary of three performance experiments</td>
<td>176</td>
</tr>
<tr>
<td>Chapter 9 Conclusion and Future Work</td>
<td>178</td>
</tr>
<tr>
<td>9.1 Thesis Summary</td>
<td>178</td>
</tr>
<tr>
<td>9.2 Contributions to Knowledge</td>
<td>180</td>
</tr>
<tr>
<td>9.3 Threats to Validity</td>
<td>182</td>
</tr>
<tr>
<td>9.4 Future Work</td>
<td>183</td>
</tr>
<tr>
<td>9.4.1 Future studies</td>
<td>183</td>
</tr>
<tr>
<td>9.4.2 Supporting offline work</td>
<td>184</td>
</tr>
<tr>
<td>9.4.3 Supporting disconnection due to network latency</td>
<td>184</td>
</tr>
<tr>
<td>9.4.4 Providing disconnection awareness</td>
<td>185</td>
</tr>
<tr>
<td>9.5 Conclusion of the Thesis</td>
<td>186</td>
</tr>
<tr>
<td>References</td>
<td>187</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2-1: Collaboration among distributed users using synchronous groupware ............................................ 18
Figure 2-2: Examples of synchronous groupware applications (collected from Internet) ................................. 18
Figure 2-3: Collaboration by generating events in a groupware application ..................................................... 20
Figure 2-4: Shared virtual workspace in a groupware application .................................................................. 21
Figure 2-5: Telepointers in a groupware workspace ..................................................................................... 22
Figure 2-6: Configuration dialog for a ‘Join with a Replay’ [59] ................................................................. 36
Figure 3-1: A simple text chat tool ............................................................................................................. 46
Figure 3-2: Reconnection behaviours of chat tool ....................................................................................... 47
Figure 3-3: A summary representation of chat conversations .................................................................... 48
Figure 3-4: A shared drawing editor ........................................................................................................... 50
Figure 3-5: Reconnection behaviours in shared drawing editor ................................................................. 51
Figure 3-6: Collaborative paper reviewing application ............................................................................... 53
Figure 3-7: Reconnection behaviours of reviewing application ............................................................... 54
Figure 3-8: Shared UML use case diagram editor ....................................................................................... 55
Figure 3-9: Shared coding editor .............................................................................................................. 57
Figure 3-10: Users login windows to declare a game server and a client in Tank game .............................. 59
Figure 3-11: Tank game ............................................................................................................................ 59
Figure 3-12: Disconnection and reconnection behaviours in Tank game .................................................... 60
Figure 3-13: Gesture-based collaborative game ........................................................................................ 61
Figure 3-14: Smooth transition effects in gesture-based collaborative game .............................................. 62
Figure 3-15: Remote presentation tool ...................................................................................................... 63
Figure 3-16: Modality fusion problem while handling disconnection .......................................................... 64
Figure 4-1: Overlay networks .................................................................................................................... 70
Figure 4-2: Message broadcasting ‘underlay network’ with three types of nodes: Sender, DiscoTech server, and Receiver ................................................................. 71
Figure 4-3: Message broadcasting ‘underlay network’ with disconnection support ................................. 73
Figure 4-4: A Five-dimensional disconnection design space .................................................................... 77
Figure 5-1: Selecting disconnection design combos ................................................................................. 91
Figure 5-2: Fidelity-reducer disconnection design combo ....................................................................... 95
Figure 5-3: Accumulator disconnection design combo ............................................................................. 101
Figure 5-4: Compacted-pacer disconnection design combo ................................................................... 106
Figure 6-1: A plug-in architecture ........................................................................................................... 116
Figure 6-2: Disconnection-aware groupware model .......................................................... 116
Figure 6-3: Plug-in architecture implementing the disconnection design space .................. 124
Figure 6-4: Class diagram for the compactor plug-ins .................................................. 125
Figure 6-5: Interactions among different components during disconnection ..................... 127
Figure 6-6: Interactions among components upon reconnection ..................................... 128
Figure 6-7: Interactions among components in peer-to-peer overlay ............................... 130
Figure 6-8: Interactions among components in client/server overlay .............................. 131
Figure 7-1: DiscoTech server .......................................................................................... 136
Figure 7-2: DiscoTech client .......................................................................................... 137
Figure 7-3: A code fragment for using ‘MessageTag’ and ‘StrategyList’ properties ............. 150
Figure 7-4: A code fragment to use the 'HeatmapTransformer' plug-in ............................. 150
Figure 7-5: DiscoTech architecture communicates with a game server ............................ 159
Figure 8-1: DiscoTech feedthrough time for shared drawing editor ................................. 171
Figure 8-2: DiscoTech feedthrough time for chat tool .................................................... 172
Figure 8-3: DiscoTech feedthrough time for disconnected clients .................................... 173
Figure 8-4: Memory use during disconnection ................................................................ 174
List of Tables

Table 2-1: Timeframes based on ‘loose powers of ten’ principle.......................................................... 27
Table 5-1: Suggested timeframe values and strategies in fidelity-reducer combo ............................... 97
Table 5-2: Suggested timeframe values and strategies in accumulator combo................................. 103
Table 5-3: Suggested timeframe values and strategies in compacted-pacer combo ...................... 108
Table 7-1: Summary of the developed plug-ins.................................................................................. 144
Chapter 1

Introduction

Synchronous distributed groupware (or simply synchronous groupware or groupware in this thesis) is a class of software applications that allows a geographically distributed group of people to collaborate in real time. The benefit of real-time collaboration is that a group of participants can communicate and collaborate with each other immediately through closely coupled interactions. There are various types of groupware, such as collaborative editing software, distributed meeting support tools, and multiplayer games. For example, a collaborative paper reviewing application allows a group of researchers located in different places to review and annotate a research paper in real time [80], WebArrow [103] allows people to conduct a remote meeting, and World of Warcraft [105] allows multiple gamers to play simultaneously over the network.

A frequently occurring problem in synchronous groupware is that during a collaborative session, group members may become disconnected. Disconnection can happen for several reasons, such as network outage, power failure and network latency. Disconnection can range from short timeframes (e.g., a few seconds) to long timeframes (e.g., hours). The disconnected users miss events (e.g., line drawing events in a shared drawing tool) and lose context, and thus have difficulty resuming work after reconnection. According to Whalen and Black “disconnections can lead to lost work and degraded session coordination” ([101], p.21). Therefore, it is important to handle disconnection in synchronous groupware. However, it is difficult for developers to handle disconnection as they encounter different problems summarized as follows:
1. Simple mechanisms (e.g., store, forward, and replay [11, 12, 59, 60]) for handling disconnection do not address the range of wide variety of user-level requirements that arise from different disconnection scenarios and more effective strategies for handling disconnection are not well known. As a result, developers have lack of knowledge about different strategies for handling disconnection.

2. Developers have also lack of knowledge about the mechanisms to combine different strategies in order to manage a trade-off between performance and understandability requirements (discussed in Section 1.2.2) that can arise in various disconnection situations when disconnection lasts longer. This trade-off is difficult to handle because its solutions vary based on a particular design situation.

3. To date no development infrastructure or toolkit exists for handling disconnection in a simple and flexible way. Without an infrastructure, disconnection handling is a complicated problem and it is hard for developers to write code from scratch in order to handle disconnection in their applications.

4. Supporting disconnection may demand extra overhead to the performance of groupware applications, e.g., it can increase message feedthrough time—measured as the time that a receiver takes to receive a message from a sender over the network. It also can lead developers to write extra lines of code. As there are only a few example disconnection-aware groupware applications (such as WebArrow and Skype [89]) out there, so it is difficult to know what the overhead really is.
In order to address these developers’ problems in handling disconnection, we have designed four solutions, summarized as follows:

1. We have developed a five-dimensional disconnection design space that captures five categories of approaches for handling disconnection. These approaches determine how stored information can be manipulated as the system waits for a disconnected user to rejoin, and how information should be replayed upon reconnection.

2. We have formulated three disconnection design combos that select and combine suitable strategies from the design space in order to manage the performance and understandability trade-off that arises in three disconnection design situations.

3. We have designed a plug-in based software architecture that shows how developers can implement different strategies of the design space in a simple and flexible manner while reusing a much of the disconnection handling code.

4. We have built a toolkit following the three solutions (mentioned above), and constructed different applications using the toolkit and conducted user studies and performance analyses to evaluate the toolkit. This step justifies the practicality of the plug-in architecture. It also provides empirical evidence about the overhead of handling disconnection in synchronous groupware.

Previous work in Computer Supported Cooperative Work (CSCW) has shown the kinds of problems that disconnection can cause, and has proposed several solutions for handling reconnection [43], late entrance (e.g., [2, 10, 11, 12, 49, 51, 59]), periods of asynchrony [71, 73], or network faults [8, 48, 88, 100]. However, existing techniques handle only a limited number of
disconnection and reconnection situations (e.g., current solutions focus on short-term disconnections, and hence do not address user-level requirements for longer-term disconnections). Moreover, existing techniques do not provide simplicity and flexibility to developers for handling disconnection.

This thesis offers four codified solutions (a disconnection design space, three disconnection design combos, a plug-in architecture and a toolkit) for making disconnection handling simple and feasible for groupware developers while resolving the shortcomings of existing techniques.

1.1 Motivation

Many synchronous groupware applications have come into wide use in the areas of communication, online gaming and electronic meetings [36]. The user-base of these applications demonstrates their wide use, e.g., Skype [89], (which allows voice and video calls, and chat over the Internet) has millions of registered users, and World of Warcraft, the massively multiplayer online role-playing game (MMORPG) [63] has more than 11 million subscribers. However, disconnection and reconnection in synchronous groupware make it difficult for users to smoothly continue with their collaborative sessions, and hamper the usability of the application. The much interests in latecomer problems [2, 10, 11, 12, 49, 51, 59] imply that it is important that a groupware application is disconnection aware and the previous work in user-level disconnection problems [43] suggests that synchronous groupware should be disconnection aware. Disconnection aware means being able to present missed information to the reconnected users in a suitable format so that they can catch-up with the ongoing collaboration upon reconnection. However, despite the promise of improved group coordination and collaboration, only a few synchronous groupware applications are disconnection aware, such as WebArrow, Skype, World of Warcraft and Age of Empires.
Moreover, these applications only support disconnection handling for limited disconnection scenarios. For example, WebArrow and Skype only present missed chat messages all at once in the chat window without having any temporal spacing between them and without providing any summary of the conversation if a disconnection lasts longer. Both applications do not play back missed voice data [43, 57]. We discussed other disconnection-aware groupware applications, such as World of Warcraft and Age of Empires in Section 2.3.6.

However, it is not surprising that only a few groupware applications are disconnection aware. According to Wu et al. [107] designing effective groupware applications is challenging for developers because they need to handle complex and challenging technical and implementation issues associated with concurrent and distributed work while simultaneously supporting fluid collaboration. On top of this, developers need to do extra work to handle disconnection from scratch. These tasks require extensive effort and can distract the developers from their primary responsibilities of working on application functionalities. Moreover, additional development leads developers to manage additional complexities and additional possibilities for more code to handle. If there were codified solutions for handling disconnection and development infrastructures, these could guide and help developers in handling these tasks, which are the primary focus of this thesis.

1.2 Problem

If users of a synchronous groupware application become disconnected, they cannot continue with the ongoing collaborative activities until the connection is re-established. When the users re-join the session, they can experience a variety of user-level disconnection problems, e.g., they may lose context and may have trouble in understanding what happened in their absence (discussed further in Section 2.2.2). For example, using a shared editing groupware application, a group of researchers
residing in different countries can edit and annotate a conference paper in real time. While collaborating, one of the researchers might become disconnected due to a network outage, and thus she would be unable to see the editing activities of other researchers during her absence; when she reconnects, she might not understand the current state of the paper and the activities of other authors. Appropriate disconnection handling is important in order to overcome such user-level disconnection problems. Disconnection handling allows reconnected users to see missed events in a suitable format (e.g., seeing a quick replay of the missed shared editing activities) so that they can understand the missed events and can get back in the current context as soon as possible and can continue with the ongoing collaboration.

However, disconnection handling is a difficult problem for groupware developers. The simple and obvious solution for handling the user-level disconnection problems is to store missed events during a disconnection period at the sender, and then forward the events to the receiver upon reconnection in order to play back events to the reconnected users. The problem with that solution is that if disconnection continues longer (e.g., 10 minutes - 1 day), many events get stored at the sender and can consume excessive memory, e.g., more than 100,000 telepointer events can accumulate within two hours of a disconnection. Upon reconnection, if all the events are sent to the receiver, they will consume excessive bandwidth and will increase the feedthrough time or transmission time. Due to high feedthrough time, users need to wait for a long time to see the reconnection information, particularly the latest ones. These events would contain information that is unimportant (e.g., telepointer information from an hour ago) and misleading information (if the users thought that they are happening now), and would take prolonged time to play them back to the reconnected users, hampering the usability of the application. Therefore, this solution is not effective. In
particular, developers face four sub-problems while handling disconnection in groupware, discussed as follows:

1.2.1 Sub-problem 1: Lack of knowledge about disconnection design space

In order to resolve performance issues (high memory usage, bandwidth consumption and high feedthrough time) and to reduce the lengthy playback time, developers need to manipulate stored events using different strategies, approaches, techniques or mechanisms. There is a wide range of disconnection scenarios with the possibilities of various disconnection and reconnection situations. For example, in a collaborative paper-reviewing application, a user become disconnected and other users can continue collaboration by doing various user actions, such as free hand gesturing using telepointers, highlighting paragraphs using telepointers, and typing and editing annotations and the user can remain disconnected for any period of time. In order to support disconnection handling for this disconnection scenario, developers need to think of strategies for manipulating stored events generated from each user action while considering different durations of disconnections. Other groupware applications lead to other disconnections scenarios and situations, so more strategies are required for manipulating events. However, more effective strategies than the simple ‘store, forward, and replay’ are not well known or studied. As a result, existing groupware applications typically address limited disconnection and reconnection situations.

1.2.2 Sub-problem 2: Managing performance and understandability trade-offs

Developers need to manage a trade-off between the performance and understandability requirements while handling disconnection. The performance requirement specifies that stored information during disconnection should be transferred over the network and presented to a reconnected user as quickly as possible so that she can rejoin the collaboration in a timely fashion.
On the other hand, the understandability requirement specifies that missed stored information needs to be presented to a reconnected user in a meaningful way so that she can understand the information and can get back in context. If developers use a summarization strategy (that transforms events to different representations) in order to address the performance requirement, this approach will not satisfy the understandability requirement. This summarization approach will allow users to see an overview of what happened during their absence, but will not inform current happenings. Conversely, in order to accomplish the understandability requirement, developers can simply store all missed events during disconnection and quickly replay them upon reconnection with their original fidelity using a speed-up strategy (that compresses time gaps among events). However, this approach might not satisfy the performance requirement because of a high volume of accumulated events during a long-term disconnection (discussed above). In order to get around this trade-off, developers need to apply multiple strategies in different disconnection timeframes because users want to see information in different degrees of richness as a disconnection progresses. However, it is difficult to handle the trade-off because its solution varies based on different disconnection design situations, including handling of different types of events (e.g., awareness and model update events) and their context of use.

1.2.3 Sub-problem 3: No simple and flexible development infrastructure exists

To date no development infrastructure exists to simplify developers’ tasks in handling disconnection. Thus, developers need to support disconnection handling in their applications from scratch, e.g., they need to write a big chunk of code for dealing with various complex issues, such as disconnection identification, queuing of events, and reconnection. They also need to write a great deal of code for implementing the wide variety of strategies (discussed in Sub-problem 1, Section 1.2.1). Each strategy has its own subtlety and difficulty when developers need to implement it. For
example, developers might need to deal with complex data structures in order to implement the strategy that summarizes telepointer gesture activities as an activity map or heatmap [80]. They also need to write code for resolving the issues for combining strategies in order to resolve the trade-off (discussed in Sub-problem 2, Section 1.2.2).

Developers need to apply multiple strategies on the stored events since there could be different types of event streams produced from a single input modality or from multiple input modalities (e.g., mouse and microphone). For example, in a paper-reviewing scenario, telepointer gesture events can be generated with a mouse, and at the same time audio events can be generated with a microphone. After applying two different strategies in these two types of event streams, resulting telepointer events might not be presented semantically coherently with the voice events. Therefore, when applying strategies on two types of related events, developers need to consider maintaining semantic coherence between the events, which is the modality fusion requirement [55].

1.2.4 Sub-Problem 4: Lack of empirical evidence

As there are a few disconnection-aware groupware applications with support of limited disconnection and reconnection situations, developers face difficulty to know how much overhead disconnection handling can incur. For example, developers might not know how much extra memory is required to store events if disconnection lasts for hours, how much extra feedthrough time is required to receive an event over the network or how much extra code they need to write for supporting disconnection handling in groupware.
1.3 Research Questions

While addressing groupware developers’ problems in handling disconnection (discussed in Section 1.2), we have identified various research questions, which are as follows:

- **RQ#1:** What makes disconnection handling difficult for synchronous groupware?

- **RQ#2:** What are the strategies that developers can use for handling disconnection?

- **RQ#3:** What problems do developers face in addressing long-term disconnection?

- **RQ#4:** Can we reuse solutions from one disconnection scenario to other scenarios?

- **RQ#5:** Can a small set of solutions cover all sorts of disconnection scenarios or do developers need a wide range of solutions?

- **RQ#6:** Does disconnection handling demand little overhead to the performance of a groupware application and little programming complexity for addressing different disconnection scenarios?

Research question one and two correspond to Sub-problem 1 and Sub-problem 3, research questions three and four correspond to Sub-problem 2, and research questions five and six correspond to Sub-problem 4. The goal of this research is to help developers in handling disconnection by answering the research questions above.

1.4 Solution

We offer three codified solutions to address the first three sub-problems discussed as follows.
- A five-dimensional disconnection design space that provides a wide variety of strategies that can be applied to manipulate stored events during disconnection. This design space improves developers’ understanding about users’ requirements upon reconnection as well as the performance issues of handling disconnection. In doing so, this design space addresses Sub-problem 1 and answers research questions one and two, in particular when designing a disconnection-aware groupware application.

- Three disconnection design combos handle a common disconnection problem that arises in three disconnection design situations if disconnection lasts longer. The common problem deals with the trade-off between performance and understandability requirements. These combos show how to apply different strategies in different disconnection timeframes in a reusable manner. These combos apply multiple strategies in long-term disconnections based on the fact that users’ requirements to see degrees of richness in reconnection information vary from one disconnection timeframe to others. For example, users want to see the telepointer gesture events that fall in a most recent timeframe with a better fidelity than the events that fall in older timeframes. As these combos show how to manage a common disconnection problem in a reusable manner, they address Sub-problem 2 and answer third and fourth research questions.

- A plug-in software architecture that shows how developers can handle disconnection in a toolkit approach in order to reduce their coding efforts for implementing the wide variety of strategies (captured by the design space) in different groupware applications. Plug-ins are software modules or classes that implement the strategies. We define these software modules as plug-ins because they are code units that serve as parameters to the architecture.
These parameters allow developers to customize the default behaviour of the architecture without changing its structure.

- The architecture allows three types of plug-ins: *fully generic*, which address standard issues such as compaction, sampling, and ordering in the data that is queued during a disconnection; *partially generic*, which are parameterized to allow tuned behaviours (e.g., to allow re-ordering with priority for certain message types); and *customized*, which allow application-dependent transformations of data based on the systems specific needs (e.g., changing movement data to a ‘historical trace’ representation). Different plug-ins and parameters provide support for different disconnection scenarios and timeframes. Moreover, with the *customized* plug-ins option, this architecture satisfies *modality fusion* requirement (discussed in Section 3.1.5). Therefore, the plug-in architecture addresses Sub-problem 3 and answers research questions one and two, in particular when implementing disconnection-aware groupware.

### 1.4.1 Evaluation

In order to address Sub-problem 4 and to answer research questions five and six, the following two steps are taken:

- First, we develop a toolkit based on the three solutions mentioned above. We use the toolkit for constructing five different types of disconnection-aware groupware applications, such as a chat tool, four shared workspace editors, a Tank game, a gesture-based collaborative game and a remote presentation tool (discussed in Section 3.1). These applications are selected for explaining different requirements for handling disconnection. This step justifies the practicality of the designed solutions and investigates our fifth research
question. The investigation results show that with a reasonable number of plug-ins (eleven in total, see Table 7-1), it is possible to address a wide variety of disconnection scenarios.

- Second, we measure the overhead of handling disconnection empirically by evaluating the toolkit. This evaluation is limited to small group sizes (maximum eleven clients), to medium disconnection lengths (maximum twelve hours) and to a local area network setting. With this step, research question six is investigated. The investigation finds that, within the cases we considered, disconnection handling incurs minimal overhead to an application’s performance in terms of feedthrough time and memory usage. Our evaluation results also show that the developed plug-in based toolkit offers simple, reusable and flexible API (Application Programming Interface) to developers (discussed in Chapter 8). This toolkit also addresses the shortcomings of the existing solutions [2, 11, 43, 48, 49, 51, 59, 71, 73] by supporting different disconnection scenarios and timeframes (discussed in Section 2.3). However, our experimental settings were limited to a small group size (maximum eleven clients), limited to disconnection period (maximum twelve hours) and in a local area setting. Therefore, it remains to study further its scalability to a large group size (e.g., over thousands of users), lengthy disconnection period (e.g., more than a year) and in a wide area network setting.

Two evaluation steps provide empirical evidence to developers about the overhead of supporting disconnection in groupware applications and thus address Sub-problem 4 (lack of empirical evidence).
1.5 Summary

Disconnection handling is challenging for groupware developers. They need to address four sub-problems (discussed in Section 1.2). In order to overcome developers’ problems in handling disconnection in groupware, we have provided four solutions including a plug-in based toolkit (discussed in Section 1.3). Our evaluation results show that our provided solutions can mitigate developer’s problems in handling disconnection.

The developed solutions and conducted evaluations provide the answers of research questions one to six (RQ#1 – RQ#6, discussed in Section 1.3), and make disconnection handling simple and flexible for groupware developers.

1.6 Scope of the Thesis

The goal of the thesis is to address human factors of the reconnection process rather than focusing on the low level networking protocols and technical aspects of disconnection, e.g., how to ensure reliability of reconnection messages or how to handle failure of the server that stores missed events. Therefore, with this thesis work, we provide solutions that offer disconnection technologies to developers for building synchronous groupware applications that can help reconnected users catch up following a reconnection. While the provided technologies are helpful for groupware developers to support disconnection handling in their applications, our solutions are directed towards particular kinds of groupware applications (such as chat tools, shared drawing editors, and multi-player online games) that support a small number of users.

Our developed DiscoTech toolkit is based on the centralized message broadcasting architecture. With this architecture developers need to send messages from a sender to receivers via a server
program. Therefore, using the toolkit, developers cannot implement a groupware application (e.g., SubEthaEdit [94]) with an underlying peer-to-peer to groupware architecture [72] where a sender directly sends a message to receivers without having any server program. Although a very few groupware applications have the underlying peer-to-peer architecture, further research requires to explore this option by storing missed events at decentralized peers.

1.7 Dissertation Outline

Chapter 2 summarizes background research, terminology and related work on handling disconnection in synchronous groupware.

Chapter 3 describes requirements for adopting different disconnection and reconnection behaviours using five types of groupware applications.

Chapter 4 describes a model of a disconnection-aware groupware that explains how different disconnection and reconnection behaviours can be implemented by applying strategies. This chapter also describes the five-dimensional disconnection design space that discusses and categorizes different strategies for implementing different disconnection and reconnection behaviours.

Chapter 5 describes three disconnection design combos that offer reusable solutions for selecting and combining various strategies in order to address the performance and understandability trade-off in three different disconnection design situations.
**Chapter 6** specifies the design of a component based plug-in architecture that demonstrates how developers can implement different strategies for handling disconnection in a simple and flexible manner while reusing different architectural components.

**Chapter 7** explains the design and implementation of a toolkit that justifies the practicality of the three designed solutions (the design space, the design combos and the plug-in architecture) by constructing various disconnection-aware groupware applications.

**Chapter 8** describes evaluation techniques for the developed toolkit in order to provide empirical evidence about the overhead of handling disconnection in synchronous groupware.

**Chapter 9** summarizes the thesis. This chapter also highlights contributions and limitations of the research work and presents areas of future work.
Chapter 2

Background and Related Research

The primary goal of this research is to develop design knowledge, design guidelines, and development infrastructure for engineering synchronous groupware that supports disconnection handling. This chapter covers relevant background. There are three sections: the first section covers synchronous groupware, including important terminology; the second section summarizes the problems faced by groupware users due to disconnections; the third and final section presents related work and existing techniques for handling disconnection in synchronous groupware.

2.1 Synchronous Distributed Groupware

This thesis uses the terms “synchronous distributed groupware”, “synchronous groupware” and “groupware” interchangeably. Synchronous groupware is distributed software that lets groups of collaborators discuss and work on a common task in real time. According to Preguica et al. “synchronous applications support closely-coupled interactions that allow multiple users to synchronously manipulate the shared data. During synchronous manipulation, all users are immediately notified about the updates produced by other users” ([73], p.89). In groupware, there are multiple clients (or client applications) that run in different geographically distributed computers. Using the client applications users interact and collaborate with each other via a network (Figure 2-1). A client can communicate with other clients directly or can communicate via a different application or program called a server.
With advances in Internet technology and computing power, synchronous groupware is a technologically feasible solution for remote communication, collaboration and coordination. According to Graham et al. [36] synchronous groupware has been widely used in three application areas:

- **Communication tools** allow geographically dispersed people to communicate in real-time using different modalities. For example, many people use the MSN messenger tool (Figure 2-2 (a)) to communicate by text chat, and currently, millions of users use the Skype tool to communicate through voice and video.
- **Electronic meeting tools** allow remote presentations and online meetings. These tools are becoming increasingly important to users. Examples of commercially successful products include WebEx [104], GoToMeeting [34] (Figure 2-2 (b)), and WebArrow [103]. With these tools, users can edit and annotate a common document, and can brainstorm ideas.

- **Multiplayer games** can be played (via a game server over the Internet) with other players around the world. They have become enormously popular over the past few years. There are various multiplayer games (such as Quake III and Halo 3), and services (such as GameSpy and Xbox Live) that allow millions of people to connect and play together. Massively multiplayer online role-playing games (MMORPGs) such as World of Warcraft (Figure 2-2 (c)) allow a large number of gamers to meet, socialize, and adventure together in the same persistent virtual world.

### 2.1.1 Terminology

Different terminology in this section explains how synchronous groupware works: for example, how it generates events, how users interact with their collaborators using its shared workspace, and how it maintains awareness among collaborators. While explaining the terminology, this section also summarizes how these terms are used in the context of this thesis.

#### 2.1.1.1 Events

In synchronous groupware, an event is an action that can be generated by a client application when a user interacts with its user interface. For example, if a user moves her mouse cursor, the client application generates mouse move events.
The event can be represented as \(<\text{MovingCursor}, x, y, >\), where \(\langle x, y \rangle\) pair is Cartesian coordinate of a mouse location. Applications update their data models and user interfaces in response to events. These events are sent to a remote client over the network and the remote client takes necessary steps to update their states and user interfaces. Figure 2-3 shows that events are generated using a local client and sent to a remote client over the network. This thesis concerns events that are created when a user becomes disconnected.

2.1.1.2 Shared workspace

Many synchronous groupware applications provide a bounded virtual space where people can see and manipulate artifacts related to their activities [40]. Figure 2-4 shows two users’ activities, such as gesturing and highlighting on a shared workspace that contains an electronic conference paper document. These applications emulate the aspects of physical workspaces, such as a chalkboard, a control panel, or a tabletop. Some groupware applications allow shared visual workspaces that allow multiple people to see similar views of objects and environments [31].
For example, a remote expert can tell you how to repair your bike by showing the physical bike using a video channel. We have developed four applications that have shared workspaces, such as a collaborative paper reviewing application, a shared drawing editor, a shared UML use case diagram editor and a shared coding editor (discussed in Section 3.1.2).

2.1.1.3 Workspace awareness

Workspace awareness in synchronous groupware is a well-studied research topic [39, 46, 47, 109]. The basic idea of awareness is to maintain knowledge of what is going on in the surroundings, e.g., who is present, where participants are performing their tasks and what they are doing. According to Gutwin et al., “workspace awareness is the collection of up-to-the-minute knowledge a person holds about the state of another’s interaction with the workspace” ([42], p.282). For example, in a collaborative paper reviewing application if a reviewer moves her cursor over a paragraph of the paper, other collaborators can see the cursor as telepointer in their shared workspace; thus, they keep themselves informed about which paragraph the reviewer is currently focusing on. Different
visual tools or widgets [46, 47] such as telepointers, shared scrollbar and radar view have been designed to provide workspace awareness in synchronous groupware. We have used the telepointers for workspace awareness purpose, discussed below.

2.1.1.4 Telepointers

People use gesturing as a natural means for communication. In particular, people use hand gestures in order to express ideas, to refer to objects, to attract attention or to signal turn taking [109]. To imitate this communication channel in synchronous groupware, telepointers are introduced. Telepointers, also known as multiple cursors, are embodiments of local users’ mouse pointers displayed on remote participants’ shared workspace. According to Xia et al., “as an important groupware interface element, telepointers are able to provide a variety of group awareness information including presence, location and activity” ([109], p.1). Moreover, groupware users exploit telepointers as a communication channel for conveying gestural messages. Therefore, telepointers become a powerful means for providing users with a collaboration context, helping users coordinate group work and improving groupware usability [41].

![Figure 2-5: Telepointers in a groupware workspace](image)
We have used telepointers in four shared workspace editors (discussed in Section 3.1.2). For example, in a collaborative paper reviewing application, telepointers are used to show remote users’ gestures over a paragraph. In Figure 2-5, a local reviewer, Debby sees the telepointer of a remote reviewer, Clive.

2.2 Disconnection in Synchronous Groupware

According to Gutwin et al.: “a central assumption of synchronous groupware is that the members of the group are temporally present – that is, they are actively observing changes to the shared workspace and noticing new updates as they arrive” ([43], p.179). However, this assumption does not always persist when disconnections occur. A disconnected user cannot observe any changes made by connected users. In this section, we present our background knowledge relating to disconnection in synchronous groupware. First, we explain the sources that cause disconnections in synchronous groupware. Second, we describe the problems users of groupware experience during disconnections. Third, we explain disconnection related terms, such as disconnection scenarios, duration of disconnection, and disconnection timeframes that we have frequently used in this thesis document.

2.2.1 Sources of disconnection

There are several different sources of disconnection:

- **Process failure**: A process might stop working due to internal errors in the client application. The process failure might disconnect the collaborators from the ongoing collaboration.
• **Power failure:** A computer system might shut down if the power cord of the computer can accidentally get unplugged or if the charge of a laptop battery dies, or if a brownout occurs. Any of these incidents will disconnect a collaborator from an ongoing collaborative session.

• **Link failure:** The network cable might get unplugged or the adaptor for connecting to the Internet might stop working, causing disconnection. With this type of disconnection, it is possible to avoid restarting an application and to reconnect a user automatically when the network connection gets fixed. In doing so, developers need to write extra code to reconnect the user automatically. For a link failure, a disconnection is reported for the connection-oriented protocols (such as TCP) but for connectionless protocols (such as UDP), link failure goes un-noticed.

• **Network latency:** This is a measure of the time delay between the creation and delivery of events [43]. There are different reasons that cause latency, e.g., congestion in the wide-area network, a low send rate or a high CPU load. This is the case of high (e.g., >50 ms) latency. There is always a small amount of latency (e.g., <50 ms) even on an uncongested network due to the time to physically transmit data (limited by speed of light). During the latency period, messages are not delivered at their intended time. For example, with TCP, if a message gets lost, several other messages may arrive, while the lost message is resent. When the lost message finally arrives, TCP would then give the lost message to a receiver client followed immediately by the other messages that had arrived previously. Thus, messages can lose their desired temporal spacing when network congestion clears and events are delivered all at once. Gutwin et al. determine that the delivery pattern of
events is similar to that of a short-term disconnection. As the network connection remains valid between two end points, the receivers only experience interruptions due to latency (or in particular variance in latency which is termed as network jitter [45]). With the component based plug-in architecture described in this thesis, it is possible to handle this kind of short-term disconnection (discussed in Section 6.2).

- **Intentional logout:** During a collaboration session, collaborators might need to quit the session by explicitly logging out due to some personal reasons. Gutwin et al. [43] term this intentional pause as explicit departure. As explicit departures are intentional, other clients in the groupware systems can be notified that the user is leaving.

Different sources of disconnection result in different kinds of disconnections distinguished by which network endpoints (sender or receiver) recognize the disconnection as soon as it happens and by the intent of the user. Gutwin et al. [43] identified three different kinds of disconnection, such as network outages (due to process, power and link failures), delay-based interruptions (due to network latency), and explicit departures (due to intentional logout). Delay-based interruptions and network outages can be categorized as un-intentional disconnections, whereas explicit departures can be categorized as intentional disconnections. Whether it is intentional or un-intentional, Gutwin et al. [43] show that it is possible to treat the three different kinds of disconnections under a single conceptual umbrella. Their observation is that these three kinds of disconnections cause packet loss for a specified period of time rather than occasionally, so these three kinds of disconnections can be handled in a similar fashion. We have followed the same concept as of Gutwin et al.
2.2.2 User-level disconnection problems

Due to disconnection, groupware users experience several problems. In order to increase the usability of groupware, developers need to mitigate these user-level disconnection problems by supporting disconnection handling in their applications. Gutwin et al. [43] identify several user-level disconnection problems:

- Due to short-term disconnection, collaborators might miss real-time communication and awareness events. As a result of missing of such events, the collaborators might lose context or might get confused about how changes occurred.

- Disconnection causes disruptions in real-time streams, such as audio, video, or telepointers. As a result, users experience difficulties interpreting information consistently, e.g., they might not be able to interpret video data consistently along with audio data.

- Upon reconnection collaborators might need to sign in back to collaborative sessions by providing their usernames once again. In case of frequent disconnections due to network outages, the task of entering usernames repeatedly could be annoying.

- If a disconnected collaborator continues working in parallel with connected collaborators, it might be difficult to merge their changes upon reconnection.

2.2.3 Disconnection scenario

The term 'disconnection scenario' is frequently used in this thesis. Disconnection scenarios express the interactions of an end-user with the system during a disconnection period. The end-user can be a disconnected user or a connected user. For example, “a user is disconnected for ten seconds, but
during her disconnection period a remote user continued gesturing using telepointers” is a disconnection scenario. In this scenario, the end-user is a connected user. Another example scenario is “a user is disconnected for an hour and during her disconnection period she continued editing a paper document using her keyboard”. Here the end-user is a disconnected user. A client application generates user events (discussed in Section 2.1.1.1) in response of a user action in a disconnection scenario. In the first example scenario, “gesturing using telepointers’ is a user action performed during a disconnection period; in response of that user action, the client application generates telepointer gesture events.

2.2.4 Duration of disconnection and disconnection timeframes

Disconnection times can be highly variable, e.g., they can last for a few seconds due to network outages or they can last for hours due to explicit departure, such as stowing a laptop while flying. The amount of time a user is absent from a collaborative session is defined as the duration of disconnection. Depending on the duration of disconnection, developers need to adapt different disconnection and reconnection behaviour in their systems. According to Gutwin et al. “this mechanism is based on people’s real-world behaviour: for example, a person would summarize a situation differently for a period of ten minutes, ten hours, or ten days” ([43], p.182).

Although duration of disconnection can be used as single continuous variable, Gutwin et al. divide the duration of disconnection into several timeframes based on a logarithmic time scale.

Table 2-1: Timeframes based on ‘loose powers of ten’ principle

<table>
<thead>
<tr>
<th>Timeframe</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;1 second</td>
<td>10 seconds- 1 minute</td>
</tr>
<tr>
<td>1-10 seconds</td>
<td>10 minutes –1 hour</td>
</tr>
<tr>
<td>&gt; 1 day</td>
<td>1 hour – 1 day</td>
</tr>
</tbody>
</table>

27
This scale is well used for describing various aspects of aspects of human action (e.g., perception of system feedback). It is based on a ‘loose power of ten’ principle (shown in Table 2-1). According to Gutwin et al. “the divisions in the scale depend on the application’s temporal granularity of interaction. For example, an IM client is unlikely to worry about millisecond-scale gaps, but this scale might be important for systems that send streaming media” ([43], p.182).

In addition, some categories are not feasible for some disconnections – e.g., it is unlikely that outages can be repaired quickly enough for a sender to make use of a millisecond (or even a tenth-of-a-second) time scale”. This thesis adapts this logarithmic timescale for splitting the duration of disconnection into different timeframes.

Duration of disconnection and disconnection timeframes both of the terms are also frequently used in this thesis.

2.3 Existing Solutions for Handling Disconnection in Groupware

Issues of robustness, fault tolerance, network disconnection, and latecomers (discussed below) have been considered in distributed and groupware systems for some time (e.g., [2, 8, 11, 27, 50, 100]). However, the user view of groupware disconnection has not been widely considered. Gutwin et al. [43] have taken the first steps to address the disconnection problem from the users’ perspectives considering different disconnection scenarios and timeframes. However, latecomer support techniques [11, 59, 60, 99] address problems similar to the user-level disconnection problems, but these techniques are not flexible and comprehensive enough to address different disconnection scenarios and timeframes (discussed in Section 2.3.4).
This section first presents the approach that addresses user-view of disconnection. Second, it summarizes different techniques for handling disconnection. Third, it presents different approaches, toolkits, and tools in the area of latecomer support and fault-tolerant groupware, which handle some problems related to user-level disconnection. Finally, this section presents related work that shows how different strategies for handling disconnection can be combined to provide reusable solutions. By studying these approaches, toolkits and tools, we have determined techniques that could be used for handling disconnection, and at the same time identified their limitations.

2.3.1 User-view of groupware disconnection

Gutwin et al. [43] provided an overview of the disconnection problem from the user’s perspective, stating two main factors for handling disconnection: the time scale of the absence, and the phase of the disconnection – identifying a disconnection, determining what to do during the absence, and determining what to do upon reconnection. Time scale is important in these decisions because users want different kinds of information depending on how long they have been away; therefore, duration of absence is used as the basis for adapting a sender’s storage behaviour during disconnection and a receiver’s behaviours upon reconnection. They adapted different disconnection and reconnection behaviours using three example systems, such as instant messaging system, a Tank game, and a drawing editor.

Gutwin et al. proposed a high-level design of the plug-in architecture described in this thesis. They show the practicality of the architecture by implementing the three example systems (mentioned above) following the architecture. In these applications, they adapted different disconnection and reconnection behaviours. However, they did not go into detail about the range of techniques that
could be used in handling disconnection and reconnection, or about development infrastructures that could allow the disconnection problem to be dealt with in an efficient and general fashion by groupware developers. This thesis addresses these issues by developing a five dimensional disconnection design space, three disconnection design combos and a plug-in based DiscoTech toolkit.

2.3.2 Techniques for handling aspects of disconnection

Several strategies for dealing with different disconnection issues have been explored in research on individual systems (e.g., World of Warcraft, Age of Empires, and WebArrow, discussed below), and general approaches and toolkits (e.g., Chung et al.’s latecomer accommodation service, Suite, Habanero, DreamObjects, Corona, Ensemble and YCab, discussed below). The wide range of techniques seen in previous literature is part of the reason why a plug-in architecture was chosen; and many of the techniques described below have been used to inform the design specific of plug-ins. These techniques do not cover some basic issues such as identification of disconnection or re-establishing a network connection (although these are implemented in our developed toolkit). In addition, these techniques do not include fault tolerance for data (e.g., data replication [67] and consistency maintenance for replicated models (e.g., [96]) that are outside the scope of the research. Strategies for handling disconnection include:

- **Persistence.** Maintaining a persistent state for the shared environment allows that state to be sent to users after a disconnection. This strategy is used for place-based groupware (e.g., TeamRooms [79]) where users log into a system that organizes tools and artifacts into persistent locations such as rooms or worlds [41].
- **State recovery.** Even in groupware systems that are not place-based, the state of documents or workspaces can be stored at a central location and sent to a reconnecting user (e.g., the YCab framework [8], discussed later). In pure state-recovery systems, no event-based information (such as awareness events) is stored. In some systems (e.g., DOORS [73]), event-based information exists when users are connected synchronously, but events are not stored during periods of asynchrony.

- **Store-and-forward.** In this approach, events are queued at a central server and are sent to the receiver upon request (e.g., XTV [12] or Chung et al.’s latecomer service [11]). Unlike state-based systems, state is reconstructed at the receiver using forwarded events.

- **Replay.** On reconnection, queued events from store-and-forward systems are delivered to the receiver and can be replayed – both to reconstruct state, and to provide an understanding of what happened during the absence. For example, DreamObjects [59] (discussed later) allows playback of some or all of the queued events, and a system designed by Manohar and Prakash [60] provided a ‘video-player’ interface that allowed playback to be paused or fast-forwarded.

- **Type-based selection.** A few systems select what messages to send after a reconnection, based on the message type. For example, WebArrow only sends chat messages after disconnection, not voice [43, 57].

- **Discarding old state-update events.** Some event-based systems recognize that certain events only update state variables (e.g., a temperature from a sensor), and discard all but the most recent of these events [20].
- **Reordering.** Some games prioritize state updates for nearby avatars ahead of those for distant avatars [7], or move important messages (e.g., deaths and spawns) to the front of the queue.

- **Coalescing and chunking.** When messages are batched at the sender, some systems reduce space requirements by aggregating several messages into one (e.g., ReConMUC [2]) or converting incremental updates into a larger single update (e.g., change multiple draw messages into a single polyline) [73].

- **Compression.** When updates are batched (or are already large), the message can be compressed using standard libraries such as Zlib [110] (e.g., XTV [12], ReConMUC [2], or Chung’s latecomer system [11]).

- **Continuation while disconnected.** When a connection is lost, some systems such as DISCIPLE [50] and Corona [88] (discussed later) allow a client to work offline. These systems are similar in approach to groupware that supports both asynchronous and synchronous work (e.g., DOORS [73] and Software Design Board [107]), but applies the idea to network disconnection.

- **Content-based filtering.** The volume of messages stored for a particular receiver can in some cases be reduced by selecting only those messages that match certain patterns (e.g., keywords in an IRC client [2]).

### 2.3.3 Summary and discussion of techniques for handling aspects of disconnection

The eleven techniques for handling disconnection in synchronous groupware introduced above (except for continuation while disconnected) include different strategies that show how to
manipulate stored data during disconnection. These techniques help us to find different dimensions in our disconnection design space (discussed in Chapter 4). We categorize all of these techniques into four dimensions, such as message formatting, sampling, ordering and pacing (discussed in Section 4.2). Considering these different techniques, this thesis proposes the plug-in based software architecture and implements different plug-ins (shown in Table 7-1) for the plug-in based DiscoTech toolkit.

However, transforming data into different representations is not included in these techniques. Nevertheless, transformation is important for addressing users’ requirements for a long-term disconnection. For example, after ten minutes of disconnection, a collaborator might want to see a heatmap representation [80] of old gestural events in order to get an overview of missed events, rather than viewing them in detail. Therefore, we have included transformation techniques in our design space.

Some toolkits (e.g., Corona and DISCIPLE) allow disconnected users to work offline. However, we did not incorporate this feature in our developed toolkit. We have considered adapting these techniques in the toolkit as a future work (discussed in Section 9.4).

### 2.3.4 Latecomer support

According to Stephan Lukosch “normally, one user starts a collaborative session and other users join it. A user who joins a session is called latecomer. A latecomer needs the current shared state to participate in the session” ([59], p.26).

The existing Computer Supported Cooperative Work (CSCW) toolkits that are closest to our work are those that deal with the ‘latecomer’ problem. This problem has been considered since the
earliest groupware systems (e.g., [11, 59, 60, 99]). Toolkit approaches to the latecomer problem can generally be divided into those that provide state-based recovery (e.g., Habanero [51] or Suite [17]), those that provide replay-based recovery (e.g., Chung et al.’s accommodation service [11] or XTV [12]), and those that combine both approaches (e.g., DreamObjects [59]). In the following three approaches are discussed with examples.

2.3.4.1 State-based recovery

This approach stores the data model in a central location or replicates the model over different clients, and when a latecomer joins, the state of the model is forwarded to the latecomer client.

Suite [17] belongs to centralized state-based approach. The Suite architecture has a central model that implements the shared application state, and multiple views that implement interfaces of different users. Suite supports the transactions principle [18] for developing multi-user systems. Therefore, it does not change the model state in response of an incomplete actions, rather it allows a sequence of actions as a unit complete action to change the model. Thus, Suite supports non-WYSIWYS interaction that means that users’ views are not synchronized and hence, they can see different things at the same time. In order to accommodate a latecomer, according to Chung et al. “when a latecomer joins a collaboration session, Suite calls a load method on the model, which sends to the latecomer's view data structures that have been committed by pre-existing views” ([11], p.131). As a result, the latecomer does not see any values that are not committed in the model.

Habanero [9] belongs to a replicated state-based category. It supports multi-tools (e.g., chat, white board, and MRI gateway) environment in order to support collaboration using various forms of information. According to Chabert et al., “Habanero works by replicating applications across
clients and then sharing all state changes in those clients” [10, p.1]. In order to maintain consistency between replicated states across clients, Habanero provides an arbitrator, which is a general floor control object. The arbitrator controls events that can be performed at a given time and enforces locks or turn-taking so that a single client modify a shared object at a time. When a latecomer joins, the arbitrator sends information about which applications are running in that session. Moreover, Habanero supports record and replay in order to support asynchronous information access. Using the facility, the latecomer can see what happened in the past.

2.3.4.2 Replay-based approach

This approach stores events that can either provide workspace awareness or construct state of the workspace/model. When a latecomer joins, the stored events are forwarded to the latecomer’s client. In the following we discuss a well-known replay based approach.

**Chung et al.’s latecomer accommodation service** [11] is a replay service for a central architecture. The service offers a latecomer accommodation server which is called the logger. A client application (called a loggable) intercepts the user events that change the local user interface and forwards them to the logger server. As a result the logger server records all changes that a client applies to the user interface of a shared application. When a latecomer joins a groupware session, the logger forwards the events to the latecomer client. Based on these events, the latecomer's client or loggable creates the user interface. When the log becomes large, the server keeps the events that result in state change and discards the rest.

2.3.4.3 Combination of state-based and replay-based approach

This approach supports both state-based and replay-based recovery techniques. For example,
**Dreamobjects** [59] allows developers to choose any of the techniques (either state-based or replay-based approach, discussed above) for accommodating a latecomer. The latecomer support is completely decentralized, i.e., it does not rely on a central site. Therefore, in the state-based (or the direct state transfer) technique, a latecomer client or site collects the current state of the session, and in the replay-based technique it collects the history list (contains the messages that result in a state change) from the participating clients. The latecomer’s client uses the history list to show how a groupware session arrived at its current state. For both techniques, Dreamobjects divides the latecomer joining process into three phases: a connection phase, an initial phase, and a final phase.

As soon as the user joins or sets a necessary value, e.g., speed of replay in the configuration dialog window required for the replay-based technique shown in Figure 2-6, the connection phase is started. In the connection phase, the latecomer’s client initializes its connections to the other participating clients. In the initial phase, the latecomer’s client chooses an initial supporter client that supplies it with an initial session state (for the state-based technique) or an initial history list (for the replay-based technique). The latecomer’s client uses the final phase either to get the latest state update (for the state-based technique) or to update the already received history list (for the replay-based technique).

![Figure 2-6: Configuration dialog for a ‘Join with a Replay’](image)

**Figure 2-6: Configuration dialog for a ‘Join with a Replay’** [59]
The final phase is important because initially received data can become outdated as other clients continued their work and perhaps can change the initially received state or the history list.

2.3.5 Summary and discussion of latecomer support

There are three different approaches for supporting latecomers in groupware: state-based, replay-based, and a combination of both. While the state-based approach only presents the current state of the application, replay-based approach shows how the application reaches its current state, by replaying events. Suite offers a centralized state-based approach and Habanero offers a replicated state-based approach. Chung et al.’s latecomer accommodation service offers a replay-based approach. DreamObjects provides support for both state-based and replay-based approaches.

The benefit of the state-based approach is that it does not require lots of memory space and does not consume excessive bandwidth. However, it fails to show different activities of users that result in the current state of the application, which makes it difficult for the latecomer to understand the ongoing collaborative activities. Replay-based approaches tackle the shortcoming of the state-based approach, but these techniques are not suitable if a large volume of events accumulates and if a latecomer joins a session too late.

Although there are often different issues at play in supporting latecomers versus disconnected users, many of the ideas are similar – and in fact some researchers have noted that absences in the middle of a groupware session can also be thought of as ‘latecomers’. However, no groupware toolkit has addressed these concerns in a way that provides both flexibility and simplicity for the application programmer; this research aims to fill this need by developing the DiscoTech toolkit. Moreover, existing techniques handle only a limited number of disconnection and reconnection situations, and there is little support for dealing with different disconnection timeframes, user-level...
concerns such as summarization, and application-specific requirements for changes made to stored data during an absence. This thesis addresses these issues by offering different plug-ins (e.g., *HeatmapTransformer*, discussed in Table 7-1) that provide a summary representation of missed events addressing the user-level concerns. Moreover, this thesis also provides an environment for developers to compose their application specific solutions for handling disconnections in a simple and flexible manner through the plug-in based DiscoTech toolkit. In order to address different disconnection timeframes, this thesis shows how to combine different strategies of handling disconnection so that both performance and users’ understandability requirements can be satisfied through designing three different disconnection combos (discussed in Chapter 5).

### 2.3.6 Fault-tolerance support

A fault-tolerant groupware application allows clients to tolerate client, link, and server (if any) failures considering the unpredictable quality of networking and computing resources [88]. When a client recovers from partial failure, fault-tolerant groupware allows *failover* meaning that it allows the client to get back to its operational state [43]. According to Shim et al. [88], in order to reduce usage overhead in fault-tolerant groupware, “disconnected clients should also be able to rejoin groups without having to restart from scratch…” ([88], p.221). This means that users who become disconnected due to client, link or server failures do not need to restart their clients and do not need to log back again because fault-tolerant groupware provides mechanisms for logging in a client automatically.

There are some groupware applications (such as Age of Empires [5], World of Warcraft [105] and the WebArrow conferencing tool [103]) that support fault-tolerance, and failover in particular. Age of Empires freezes gameplay when a player becomes disconnected; after a certain time threshold,
the disconnected player is discarded and the other players are allowed to continue [5]. In World of Warcraft, a disconnected player’s avatar stays in the world for some time (e.g., a few minutes) after disconnection and then, it is removed from the user interface of the game [43]. WebArrow only sends the chat messages to bring a disconnected user up to the collaboration state but does not take any actions for other types of events (such as voice).

Only a few toolkits allow developers to implement fault-tolerant groupware, such as Corona [88], Ensemble [100], and YCab [8]. These toolkits either provide fault-tolerance using either a centralized or a replicated approach.

2.3.6.1 Centralized approach

The centralized approach deploys a central server to support the fault tolerance. For example, **Corona** offers a server, called Corona that ensures fault tolerance against clients, links and server failures. Developers use the DistView toolkit [76] that provides interfaces for Corona services, e.g., groupJoin(), groupLeave(), and bcast(). In order to support fault-tolerance, Corona introduces stateful group communication in which the Corona server manages shared application state, transfers the state to a new client and logs messages in order to synchronize the disconnected client to the shared state. Thus, in the case of a client or link failure, a new client can join to the group by obtaining the current group state from the Corona server. In Corona, a group is a set of client applications that are communicating with each other by broadcasting messages to the group, and a group can be either stateful or stateless. A stateful group keeps a copy the shared application state, whereas a stateless group does not have any shared state and communicates by broadcasting messages.
When a client wants to update group state, the Corona server associates a lock to it with a grace period, in order to support transient failure of the client. The Corona server waits for the client until the grace period is over, and if it does not recover within the grace period, Corona server makes the lock available for other clients. If the client rejoins within the grace period and if it performs any changes, Corona server updates the change in the application state. However, if the client joins after the grace period and makes changes in the application state that conflict with the current state of the application, the Corona server discards its changes and allows the client to rejoin as a new client; but if the changes are non-conflicting, it merges the changes. Moreover, if the Corona server crashes when a client has a lock, the client can rejoin its group automatically as soon as (could be within the lock’s grace period) the Corona server restarts. In order to support failures of the Corona server, Corona logs messages related to group state in a persistent location. After recovering from failure, it reforms the state from the persistent location. Moreover, during its own failure, the sender clients temporarily store the messages for which they do not receive any acknowledgment from the Corona server. Corona server and manipulates those messages that sender clients store during its absence time.

**Ensemble** [100] offers a component-based infrastructure for mobile systems. Ensemble’s simple plug-and-play architecture can configure the system at runtime in order to satisfy the fault tolerance requirement of an application. Ensemble achieves fault tolerance by providing libraries that can replicate key portions of an application to make it fault tolerant. To handle fault tolerance, Ensemble allows clients to access highly available objects via a RPC (remote procedure call) system. It provides the option to reconnect a client automatically to a new instance of an object if the connection to an existing instance of an object fails. In order to accommodate a latecomer or a disconnected client, Ensemble supports casual logging that provides a central view of all events
during a session. Events are logged using a causal time stamp mechanism. The timestamp mechanism uses logical vector clock to check if the occurrence of an event is a consequence of the occurrence of another event. Each client keeps the logs and later the logs are combined to provide a global view of the session’s actions.

2.3.6.2 Decentralized approach

In this approach, fault-tolerance is supported using the existing clients in a decentralized manner without having any central controller. For example, **YCab** [8] offers a decentralized approach for supporting fault-tolerance in mobile and adhoc networks. YCab deploys a state recovery mechanism which enables clients to be brought to a state that is consistent with other session participants. With this mechanism, a ‘session coordinator client’ is selected to send the current application state to the recovered or a newly joined client. When a session coordinator is not present, the leader election algorithm [8] is used to designate a new coordinator. In order to remove a disconnected client from the client-list or to add a newly joined client, a client sends a ‘ping’ request messages to all other clients in a session. Then, the sending clients wait for a certain period of time to allow other clients to reply. The list of clients that reply to the ping request are compared with the existing clients list of the sending client. Clients in the list who did not reply then removed from the list. The members who replied but are not in the client list are included in the client list.

2.3.7 Summary and discussion of fault-tolerance support

There are two different approaches for handling fault-tolerance in groupware, such as centralized and decentralized. Corona and Ensemble both follow the centralized approach by using central controllers (i.e., servers) for supporting fault-tolerance; on the other hand, YCab follows a decentralized approach by using the existing clients in mobile ad-hoc networks. Although Corona
and Ensemble both offer centralized approaches, Corona uses message broadcasting for both desktop-based and mobile-based systems and Ensemble uses RPC for supporting fault-tolerance in mobile systems. Corona also supports fault-tolerance against server crashes, but Ensemble assumes that a server is always available. Corona also allows a disconnected client to continue to work offline, and supports fault-tolerance if the Corona server crashes at the time when the disconnected client merges its changes with the application state upon reconnection. In contrast, Ensemble and YCab do not merge offline work.

Corona, Ensemble and YCab provide support to rejoin a disconnected client automatically in a groupware session without logging in again, and synchronize a disconnected client with the current application state. However, these toolkits fail to address user-level concerns of losing context, as they do not consider showing how the application reached to its current state. In contrast, this thesis handles the user-level concerns of losing context by using the timeline concept (discussed in Sections 2.2.4 and 5.2).

### 2.3.8 Reusable solutions for handling disconnection

Design pattern provide reusable solutions for addressing common problems. Christopher Alexander says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” ([28], p.2). Gama et al. (known as Gang of Four) [28] used the idea of Christopher Alexander and proposed 23 design patterns which are frequently used by object oriented software designers. However, there are few examples providing reusable solutions for recurrent disconnection problems. This thesis can only refer to limited previous work that provides reusable solutions for handling disconnection.
For example, Gutwin et al. [43] and our published work [80] show some context and application specific patterns for handling disconnection. These two approaches are the first steps for formulating disconnection design combos. However, they do not show any systematic and practical ways to use these combos for various disconnection scenarios in a general and reusable way.

Existing disconnection approaches, such as store and forward [11, 12] and event replay [59, 60] can jointly be treated as another combo of handling disconnection; but unfortunately, this combo is not suitable for handling disconnection for various timeframes. On the other hand, in this dissertation the primary target with the disconnection design combos is to provide reusable solutions including different trade-offs for handling disconnections problems not only for various disconnection timeframes but also for different groupware applications and their context of use.

2.4 Summary of Background and Related Research

In this chapter, we first presented our background knowledge in synchronous groupware including different terminology such as events, shared workspace, workspace awareness and telepointers. Then, we described different sources of disconnections, such as power and network outages, process failures, network latency and explicit departures, that disconnect a user from an ongoing collaborative session. After that, we discussed different user-level disconnection problems, such as interpretation difficulty, inconsistency in viewing information, and tedious reconnection process. Then, we presented the approach [43] that directly addresses user-level disconnection problems considering different disconnection scenarios and timeframes. After this, we present the existing eleven techniques for handling disconnection, including type-based selection, type and age based filtering, coalescing and chunking, compressing, reordering, replaying, recovering the state of the data model and continuation of work while disconnected. Finally, we present the work that handle
similar problems of handling disconnection, including latecomer support [11, 59, 60, 99], fault-tolerance support [5, 8, 88, 100, 103, 105], and reusable solutions [43, 59, 60, 80, 82] for handling disconnection in synchronous groupware.
Chapter 3

Designing Disconnection-aware Groupware

We have provided four codified solutions (a disconnection design space, a plug-in architecture, three disconnection design combos and the DiscoTech toolkit) in order to support disconnection handling in synchronous groupware. We discuss these solutions in chapters four to seven. We have justified the practicality of the proposed solutions by developing different types of disconnection-aware groupware applications using the DiscoTech toolkit. However, to illustrate different disconnection related application behaviours (DRAB) that can be implemented using the solutions, we present the developed applications in this chapter. We explain the requirements for adopting a wide range of disconnection and reconnection behaviours using these groupware applications from users’ experience point of view. We first show that if an application does not support disconnection handling, what problems a user could experience. Then, we show how the application could overcome the problems by adopting different disconnection and reconnection behaviours. An application exhibits a disconnection behaviour during disconnection, whereas it presents a reconnection behaviour upon reconnection.

3.1 Examples of Disconnection-aware Groupware

We identified five types of applications from the areas of online chatting, shared workspace manipulation, online gaming, remote presentation, and gesture-based collaborative gaming. These applications are simple groupware applications designed to explore different disconnection and reconnection behaviours.
Groupware applications are expected to have such behaviours in order to mitigate different user-level disconnection problems (discussed in Section 2.2.2, and discussed below), and hence to improve the usability of the applications. In the following, we discuss how five different types of groupware applications can improve users’ experience by presenting different disconnection and reconnection behaviours to users.

### 3.1.1 Text chat tool

In this simple chat system, users type a chat message and press a ‘Send’ button or ENTER to propagate the chat message to all users (Figure 3-1) in the chat session. The application uses two message types: (1) the content of a chat message, and (2) the ‘is typing’ awareness message, including the user name of the sender. The content of a chat message is sent when the ‘Send’ button or ENTER is pressed. The ‘is typing’ awareness message is sent when a user types a chat message (in Figure 3-1, when Clive pushes a key down, remote users see ‘Clive is typing’ awareness message).
While chatting, a user might become disconnected for many reasons (discussed in Section 2.2.1). Consequently, she cannot see the ongoing conversations among other connected users continued during the disconnection. Upon reconnection, if disconnection handling is not supported by the system, the user will not be able to see any missed chat conversations that were exchanged by other users during her absence. As a result, she can lose context and could experience difficulty in understanding ongoing conversations. For example, consider a scenario as follows:

A student and two of her supervisors were discussing something using the chat tool and when a supervisor asked for her research update, another supervisor became disconnected due to a network outage. During his disconnection, the student informed the connected supervisor about her update by typing chat messages. However, when the disconnected supervisor rejoined the session, he started wondering about the missed conversation during his absence. Consequently, it became difficult for him to rejoin the ongoing conversation. We demonstrate the reconnection situation in Figure 3-2 (a), where only the chatting windows of the reconnected supervisor are shown.

![Figure 3-2: Reconnection behaviours of chat tool](image)

(a) With no support of disconnection handling  (b) With support of disconnection handling
However, the chat system could overcome the supervisor’s disconnection problem by replaying the last minute of missed conversation quickly as soon as the supervisor reconnects (as shown in Figure 3-2 (b)). This reconnection behaviour of the application could help the reconnected supervisor see the missed information and resume the conversation right after rejoining the chat session.

However, if the duration of disconnection was lasted longer (e.g., more than ten minutes), he might not prefer to see the old ‘is typing’ awareness messages and to read all the old or outdated chat messages in the same way as of the recent chat messages. Watching these old messages, even quickly, will prolong the catch-up time (defined as the time to see the missed information), but will not help the supervisor that much to understand the ongoing chat conversation. Therefore, he might rather be interested to see a summary of old chat messages in order to get some idea about activities of other collaborators (as shown in Figure 3-3 at the circled area). In the figure, each user icon is highlighted based on a number of messages sent from a particular user during disconnection (e.g., Banani’s user icon is more highlighted than the rest of the users’ as she sent the highest number of messages). Therefore, we see that a user wants to see chat messages at different degrees of richness.
as a disconnection progresses. The user wants to see recent chat messages in more detail than old chat messages.

With this chat tool, we demonstrated that if disconnection handling is not supported in this application, it can be difficult for a reconnected user to catch-up upon reconnection due to information loss during disconnection. We showed that the reconnected user’s problems can be overcome by adopting different reconnection behaviours, such as displaying events quickly, discarding unimportant events, and presenting a summary representation of old chat conversation.

### 3.1.2 Shared workspace editors

This is a common category of groupware where a group of users collaborate on a shared workspace (discussed in Section 2.1.1.2) by seeing, creating, and manipulating artifacts related to their activities. We develop four applications of this type in order to illustrate how disconnection can be handled in different shared workspace environments summarized as follows:

- **A shared drawing editor** allows sketching a diagram.
- **A collaborative paper reviewing application** allows reviewing a paper document.
- **A shared UML use case diagram editor** allows dragging, dropping and editing use case names and diagram elements.
- **A shared coding editor** allows ‘copy and paste’ commands.

#### 3.1.2.1 Shared drawing editor

With this drawing editor (Figure 3-4), a group of users can draw different shapes, such as circle and lines in their shared workspace. With this drawing editor (Figure 3-4), a group of users can draw different shapes, such as circle and lines in their shared workspace.
Figure 3-4: A shared drawing editor

They can also fill in the colour of a shape. Telepointers (discussed in Section 2.1.1.4) show locations of other users’ mouse pointers. When a user sketches using her mouse pointer, two kinds of events are generated: telepointer move events and sketch commands. The application changes the mouse pointer from an arrow shape to a brush shape when colouring.

If disconnection handling is not supported in this application, it might be difficult for reconnected users to understand the ongoing drawing activities. For example, consider the following scenario:

In a drawing session, someone (e.g., ‘Nick’ in Figure 3-4) was showing Sally (a high school student) how to sketch an animal using some basic diagrams, such as circles and lines. As soon as he drew two circles, Sally became disconnected, because her little brother unplugged the network cable. Having a conversation with her brother and fixing the problem, Sally reconnected after about eleven minutes. When she rejoined the session, she saw a lion in her drawing window (as shown in Figure 3-5(a)). As she missed the intermediate drawing steps, she might be confused how the
two circles became a lion. However, the system could have assisted the high school student by supporting disconnection handling. In this disconnection scenario, if she could see some of the intermediate drawing steps one after another with short gaps, it could have helped her to understand how the two circles changed to a lion. For example, Figure 3-5(b) shows the step-by-step update of the sketch. In each step, a series of sketch events are jointly displayed as a single sketch event. Each frame (shown in a dotted rectangle) aggregate represents two minutes of sketch events. These steps would tell the high school student how the workspace changed to its current state.

Figure 3-5: Reconnection behaviours in shared drawing editor
However, these steps would only tell the reconnected user what happened in her absence, but not the current ongoing activities. In order to show the current happenings, the application needs to show a quick replay of the most recent events (e.g., the telepointer events) to the user. Then, she would be able to know that the remote collaborator was recently colouring the tail of the lion with the paintbrush telepointer (as shown with the right most lion in Figure 3-5(b)). Therefore, it could help the user rejoin the collaboration as soon as reconnection happened.

However, if disconnection lasted longer (e.g., over 3 hours), the reconnected user probably would not want to watch hours of individual events or even many aggregated events. She might prefer to see just the end result of the older events. Moreover, she might be interested to see the drawing events in a different order. For example, the application might show her the latest drawing events at the beginning in order to help rejoin the drawing activities as soon as possible, and then gradually show the aggregated events in a reverse order to help her understand how the drawing objects changed to their current states.

With this application, we illustrated two more disconnection and reconnection behaviours: combining a sequence of sketch events as a single larger update event, and displaying events in a different order.

3.1.2.2 Collaborative paper reviewing application

With this application, a group of authors can review and comment on a paper draft (Figure 3-6). The application allows real-time text highlighting, whiteboard-style annotations, and addition of typed comments, and provides telepointers to convey gestures. The application also shows a summary of comments in a separate window (when the ‘CommentSummary’ button is pressed).
As with the use case editor, the application sends a stream of events specifying the current position of each user’s mouse pointer. The application also sends events capturing users’ markup operations, such as adding a comment or an annotation.

If disconnection is not supported in this application, upon reconnection user will miss important reviewing activities. For example, Debby, a PhD student, was discussing a paper draft with her supervisors. She became disconnected due to a brownout in her residential area, and when she reconnected after 1 hour, she saw that her supervisors are discussing on a different page. She noticed that no new telepointer gestures were performed and no comments were added to previous pages, so she wondered whether her supervisors actually read those pages.

However, if disconnection handling was supported in this application, her problem could be solved. For example, if she is disconnected for a short time (e.g., one minute), a quick replay of missed telepointer gesture events could tell her what happened during her absence.
However, because she was disconnected for a long time (e.g., an hour), she would not prefer to see a quick replay of old missed gestural activities. If she was able to see a summary of old missed gesture events as a heatmap or an activity map (Figure 3-7 (a)), along with the quick replay of the most recent events, it could help her rejoin the reviewing activities without any confusion.

A heatmap shades a document according to the number of telepointer events in each area of the document. This provides a visual cue to the reconnected users about the portion of the document that the connected users actively worked on during a user’s absence. Moreover, it could be helpful for the reconnected user if she was able to see a summary of comments made during her absence (as shown in Figure 3-7 (b)). This comment summary lists the comments made by users with the page numbers, and provides overall idea about how many comments have been made in a particular page and by a particular user during the absence. If she could see this comment summary upon reconnection, she would have a quick look at all the comments without requiring scrolling through all the pages to find and read the comments.

**Figure 3-7: Reconnection behaviours of reviewing application**

(a) Heatmap

(b) Comment Summary
With this application, we demonstrated the requirements for two new reconnection behaviours. First, presenting past events into a colour-coded heatmap and second, presenting character-by-character comment typing events as a comment summary representation.

3.1.2.3 Shared UML use case diagram editor

With this application, a group of designers can sketch a UML use case diagram by dragging and dropping use case diagram elements or symbols (Figure 3-8). A use case diagram captures the tasks that users perform while interacting with a system. In order to use the editor, collaborators select UML elements from a palette – actors, use cases and connectors – and place them on the diagram. Diagram elements can be moved by dragging on the canvas. Text and free-hand annotations can be added to the diagram using a drawing tool.

![Shared UML use case diagram editor](image)

**Figure 3-8: Shared UML use case diagram editor**
If disconnection handling is not supported in this application, users can experience different problems upon reconnection. For example, consider, an undergrad student was working on a class project of designing a use case diagram with her two project mates. When one of her project mates was editing use case names and use case diagram elements (e.g., repositioning and resizing), her mother called her for having dinner. Thus, she left the session by signing out, so that other family member could not accidentally edit the diagram in her absence. After one hour, when she signed back in, she saw a completely different diagram and could not understand how the diagram changed to its current state.

However, the system could help her. For example, it could show her the end result of the old events, and then a quick replay of most recent editing events. There could be a large number of edit events for the longer disconnection, and the older edits possibly are not important to resume collaboration.

With this application, we demonstrated the requirements for transforming older editing operations into a single edit event as an end result.

3.1.2.4 A shared coding editor

With this shared drawing editor, a group of programmers can write, edit copy and paste text data in real time (Figure 3-9). This editor supports the PHP coding format. This application is designed in order to explore how disconnection can be handled for ‘copy and paste’ commands. If a copy command is issued, the user’s client stores the copied data in its clipboard. Then, when a paste command is issued, the data are sent over the network to other users’ client computers.
Figure 3-9: Shared coding editor

The ‘copy and paste’ commands are particularly focused on this application because these commands work differently than typing character-by-character texts already explored in the paper reviewing application.

If disconnection handling is not supported in this application, a programmer will miss a copy and paste command. Consequently, she will not able to compile the shared program. Different disconnection behaviours can be adopted in this editor. In particular, recent copy and paste commands can be displayed quickly by maintaining their original speed, whereas a sequence of older commands can be displayed jointly with a short gap from the immediate past displayed command. Finally, the much older commands can be displayed all at once without any time gap.

With this application, we explained how disconnection handling can be supported for the ‘copy and paste’ commands in a shared coding editor.
3.1.2.5 Summary of shared drawing editors

In this section, we described four shared coding editors in order to identify different disconnection and reconnection behaviours in a shared workspace environment. Each of the editors are used for explaining disconnection and reconnection behaviours for particular events, summarized as follows:

- The shared drawing editor illustrates the disconnection and reconnection behaviours for the sketching events.
- The collaborative paper reviewing application illustrates the disconnection and reconnection behaviours for telepointer gesture and comment writing events.
- A shared UML use case diagram editor illustrates the disconnection and reconnection behaviours for use case diagram elements dropping and editing events.
- A shared coding editor illustrates the disconnection and reconnection behaviours for copy and paste commands.

3.1.3 Tank game

The traditional online real-time game (e.g., World of Warcraft) offers a game server in order to control the game play and to take different decisions based on stored game state. There are some games where a user explicitly declares the role of an instance of the game application as a game server, e.g., Warcraft III [102]. In doing so, the user hosts a game server. On the other hand, the players log in the game clients for playing the game. Therefore, this type of game is different from the other applications described earlier (e.g., the chat tool and the shared workspace editors) where users do not need to declare an application server. In order to explore disconnection and
reconnection behaviours for the application that incorporates a game server, we design the
multiplayer online Tank game, discussed as follows.

A player shoots other players’ avatars. If there is a hit, the shooting player receives a point, which
is displayed at the bottom of the player’s avatar. Moreover, each player maintains her health where
maximum health assigned is five and minimum is one. If a player is shot, her health is decreased
by one. The health is represented by the line thickness of the tank outline.

![Figure 3-10: Users login windows to declare a game server and a client in Tank game](image)

(a) A game server’s login window  (b) A game client’s login window

**Figure 3-10:** Users login windows to declare a game server and a client in Tank game

![Figure 3-11: Tank game](image)

**Figure 3-11:** Tank game
For example, in Figure 3-11, pl2’s line thickness is less than the other players because it got a shot. As mentioned earlier, the game server takes different game decisions. For example, if a player shoots at an opposing player (red dots in Figure 3-11), the game server decides whether the shot is a ‘hit’ or a ‘miss’ or where an initial position of the player is after joining the game.

This game is also developed to show how connected users can be prohibited from interacting with a disconnected player’s avatar using a protection shield. The protection shield provides awareness to connected users for not interacting with a disconnected user. In Figure 3-12(a), when ‘pl2’ shoots the disconnected player ‘pl3, the game shows a red circle around p13’s tank. The red circle visual representation is a protection shield that prohibits ‘pl2’ from shooting ‘pl3’. As with the other applications, this application can exhibit different reconnection behaviours in order to help a reconnected player gets back in game play. For example, displaying events quickly and historical traces of movements of tanks (Figure 3-12(b)). The historical traces visually show the paths of the connected players and might help a reconnected player to understand the course of other player’s movements during her absence at a quick glance.

Figure 3-12: Disconnection and reconnection behaviours in Tank game
With this Tank game, we explained an application server can be incorporated into the architecture, and demonstrated an additional disconnection behaviour. This behaviour prohibits connected players from interacting with a disconnected player, and shows a “protection shield” to provide awareness to the connected players. We also illustrated how older events can be used to generate visual historical traces. This behaviour is already explained in the collaborative paper reviewing application.

3.1.4 Gesture-based collaborative game for older adults

In this game, two players produce and grow plants by moving their hands in front of the monitor (Figure 3-13). This game is designed based on Gerling et al.’s work [32]. A Microsoft Kinect [62] is used to detect hand gestures of the players.

Figure 3-13: Gesture-based collaborative game
The game is developed for older adults who have restricted range of motion and slower interactions, in order to improve their health conditions. In the game, a player can move one of her hands at a time and can see the hand movements in her game window (green colour palm as shown in Figure 3-13). She can also see one of the hands’ movements (gray colour palm as shown in Figure 3-13) of her partner at a time. When players move their hands up, the plants produce and grow. They can see the plants are being produced and grown in their game windows. The players score ten if they can produce a plant and another ten if they can grow the plant one size further.

We design this game in order to show that although quick catch-up behaviour is suitable for different varieties of groupware applications (such as the chat tool, the shared drawing editor and the Tank game), it is not well-suited for all kinds of applications. As this game is for older adults, it is a slow-paced game. If a player gets disconnected for even a short period of time, e.g., less than 15 seconds, in order to catch-up upon reconnection, quick playback behaviour would display both the movements of the hand and growing of plants rapidly which might negatively impact the style and pacing of the game.

![Figure 3-14: Smooth transition effects in gesture-based collaborative game](image-url)
Therefore, after disconnection, a quick replay of missed events would be unsuitable. In order to fit well with the pacing and style of the game, alternative ways are required to bring the reconnected user back in context.

For example, the *smooth transition effect* can be used, which will blend the hand to its current state through a fade animation, and will show the growing of the plants in a smooth way as shown in Figure 3-14 (also see a video figure elsewhere [98]). In this figure, we only showed a partial view of the reconnected player’s game window in order to depict only the transitions. However, although the playback cannot be fast and rapid, the overall catch-up time should not be lengthy.

With this application, we analyze the requirement of having another new behaviour in handling disconnection. It shows that quick playback is not always suitable; rather a smooth transition effect is desirable.

![Figure 3-15: Remote presentation tool](image)

---

63
3.1.5 Remote presentation tool

With this application, a presenter can present her slides to remote collaborators by annotating and gesturing over slides using telepointers (Figure 3-15). This application demonstrates how disconnection can be handled where users can share their workspace and pointing data with remote collaborators. Disconnection handling for this kind of application leads us to study the modality fusion requirement, discussed below.

3.1.5.1 Modality fusion requirement

Multimodal fusion allows the combination of a number of data from a number of input modalities in a multimodal interactive system [55]. An application can generate different event streams from a single input modality or multiple modalities. We specify the modality fusion requirement for combining and presenting two types of event streams in a semantic coherent manner based on the context of a workspace upon reconnection. In the following, we discuss it using the remote presentation tool.

Figure 3-16: Modality fusion problem while handling disconnection
In the remote presentation tool, a presenter can gesture and annotate over slides to explain their contents. For this action, the application generates telepointer gesture events. The presenter can go to a new slide by pressing the “Next” button. For this action, the application generates slide switch events.

Upon reconnection, a reconnected viewer might want to see a quick replay of the gesture events. In order to satisfy this requirement, the application plays back gestural events quickly. The viewer might also prefer to see only the latest slide upon reconnection in order to quickly get back in context. Due to this requirement, the application might not display the slide where the gesture was made. As a result, the gesture information can be presented at the wrong slide upon reconnection. For example, the viewer might see a quick replay of the gesture events that display a researcher’s name at a slide that contains an image of the I/O device (as shown in Figure 3-16 and also see a video figure elsewhere [98]). This kind of inconsistent information will leave the viewer confused, e.g., she might wonder how an I/O device’s name can be “Mr. Bateman”. In order to overcome such information inconsistency, two types of event streams need to be manipulated and combined at a semantic level so that they can be presented coherently along with the context of the workspace.

Therefore, in situations where an application needs to consider the modality fusion requirement, it should take necessary actions. For example, in the remote presentation tool, the application can combine the two types of events at a semantic level. The resulting events will allow the viewer to see the appropriate slide numbers with the annotations. However, this behaviour is applicable for the remote presentation tool. Other applications need to show information differently for their specific events and workspace context.
With the remote presentation tool, we analyze the modality fusion requirement for handling disconnection.

### 3.1.6 Discussion and summary of disconnection-aware groupware

This section presented five different groupware applications: a chat tool, four shared drawing editors, a Tank game, a remote presentation tool, and a gesture-based collaborative game. These applications demonstrate in total ten disconnection related application behaviours (DRABs), summarized as follows:

- **DRAB#1**: Displaying missed events quickly, demonstrated with different applications, e.g., the chat tool and the collaborative paper reviewing application. In these applications, most recent missed events are played back at a faster pace than the original pace of the events.

- **DRAB#2**: Displaying events in a different order, demonstrated with the shared drawing editor. In this application, the latest drawing events are presented before the older ones.

- **DRAB#3**: Showing all incremental updates of a data model as a single update, demonstrated with the shared drawing editor. In this application, older sketch events are presented all at once to show the large single update of the sketch, instead of showing incremental updates of it.

- **DRAB#4**: Showing a series of edit events as an end-result edit event, demonstrated with the UML use case editor diagram. In the editor, the use case symbol position edit events or use case name edit events are aggregated into a final edit event.
- **DRAB#5**: Not displaying unimportant events, demonstrated with the chat tool and the shared drawing editor. For example, the chat tool discards the older ‘is typing’ awareness events.

- **DRAB#6**: Presenting overview of older missed events, demonstrated with different applications, such as the chat tool, the collaborative paper reviewing application, and the Tank game. For example, the Tank game presents the historical traces of older tank movements.

- **DRAB#7**: Presenting missed information with different degrees of richness and fidelity as a disconnection progresses. This demonstrated with different applications, such as the chat tool and the collaborative paper reviewing application. For example, the collaborative application displays the recent telepointer gesture events with their original fidelity and the older events with lower fidelity than the original ones by using a heatmap display.

- **DRAB#8**: Presenting two different types of events consistently along with workspace context while satisfying the modality fusion requirement, demonstrated with the remote presentation tool.

- **DRAB#9**: Presenting information with smooth transition effects for maintaining the slow pace of a game, demonstrated with the gesture-based collaborative game.

- **DRAB#10**: Prohibiting connected users to interact with a disconnected user, demonstrated with the Tank game.

The first nine application related disconnection behaviours guided us to design our three solutions (such as the five-dimensional design space, three disconnection design combos and the plug-in architecture) for handling disconnection in synchronous groupware. In Chapter 4, Chapter 5 and Chapter 6, we present these solutions and show how these behaviours can be implemented using
these solutions. As the focus of the thesis is on the reconnection process, we leave the implementation of DRAB #10 to future work (discussed in Section 9.4.4).
Chapter 4

Disconnection Design Space

In Chapter 3, we explained requirements of ten disconnection-related application behaviours using eight different applications. In this chapter, we discuss an architectural model that explains how a groupware application can apply different strategies to implement the behaviours. This model shows that some of the strategies operate over the events that are stored in the event queue of a message-broadcasting server during disconnection, while others operate over the events received by a client application upon reconnection. Then, we categorize and discuss different strategies using five design dimensions. These dimensions provide codified solutions to developers for addressing Sub-problem 1 (lack of knowledge about disconnection design space, discussed in Section 1.2.1). After that, we analyze the expressiveness of the design space based on our development experience with eight disconnection-aware groupware applications. Then, we discuss the benefits of the design space, and summarize how it addresses the ten applications’ behaviours related to disconnection handling. Finally, we summarize the chapter.

4.1 A Model of Disconnection-aware Groupware

In distributed system, the term overlay networks [58, 69] is frequently used to describe how participant nodes (representing the computation platforms such as PCs, laptops, and servers) are connected together at a conceptual level. Two commonly used overlays are:
Figure 4-1: Overlay networks

- Peer-to-peer overlay where each participant node runs an instance of the full groupware application. When a user interacts with an instance of the application, the instance generates events (discussed in Section 2.1.1.1), which are broadcasted to other participating nodes, allowing all the instances to maintain consistency among their views. Each instance of the application holds a replica of the shared data model that determines the state of the application. Different commercial groupware applications have been developed based on this type of overlay architecture, such as Microsoft’s Groove [37], Age of Empires [5], and SubEthaEdit [94].

- Client/server overlay where functionality of an application is divided between a server and a client. The responsibilities of a server and a client vary from one system to another, and the server is capable of providing services to multiple clients concurrently. In this architecture, a client can generate events when a user interacts with it. The client sends the events to other clients via the server. The server might hold the shared data model and determine the state of the application using client events. Different commercially successful groupware applications have been developed based on this traditional
client/server overlay architecture, such as WebEx, WebArrow, GoToMeeting, and World of Warcraft.

We have implemented most of our groupware applications (including the chat tool, shared drawing editors, remote presentation tool and gesture-based collaborative game, discussed in Section 3.1) using a peer-to-peer overlay architecture. Conversely, we have built the Tank game (discussed in Section 3.1.3) following a traditional client/server overlay architecture. In order to support both of the overlays, we built these applications on top of a single message broadcasting underlay network. This layer sits on top of the TCP/IP underlay network [66, 69] that describes the connection structure of the groupware system. It shows the real nodes and the real links between those nodes.

![Message broadcasting ‘underlay network’ with three types of nodes: Sender, DiscoTech server, and Receiver](image)

*Figure 4-2: Message broadcasting ‘underlay network’ with three types of nodes: Sender, DiscoTech server, and Receiver*
In our groupware model, the message broadcasting underlay network comprises of a set of participant nodes for each participant in the groupware session. These nodes are connected through a message broadcaster, which broadcasts messages to participant nodes. At this underlay level, we define three nodes to explain the functionality of a disconnection-aware groupware application: *Sender*, *DiscoTech server*, and *Receiver* as shown in Figure 4-2 (we used the term DiscoTech server in order to distinguish it from the server program of the client/server overlay network and to relate it with the disconnection technology). In Figure 4-2, the sender and receiver nodes are the participants’ nodes and the DiscoTech server node is the message broadcaster. The sender node sends events using its client program to the server node. The DiscoTech server relays the events to the receiver using its *Message Broadcaster* component. The receiver’s client presents the events to the user or the participant. The sender and the receiver can switch their roles and there could be multiple receivers based on the number participants in the groupware session.

As discussed earlier, on top of this single underlay, we have implemented both the peer-to-peer and client/server overlay architectures. Implementing the peer-to-peer architecture is straightforward where we instantiate a complete groupware application both at the sender and receiver sides. However, in order to implement the client/server architecture, this underlay allows a sender or receiver client to declare itself as an application server. In this case, the message broadcaster does not directly broadcast messages to receivers, but via the application server (discussed in further in Section 6.2 and 7.9.3). If there is no application server, it simply broadcasts messages to receivers.
In order to handle disconnection, we add new capabilities both to the DiscoTech server node and the participants’ nodes (both sender and receiver) as shown in Figure 4-3. We extend the message-broadcaster to store events during disconnection and to forward them to the receiver upon reconnection. We extend the receiver to replay the stored events when they arrive from the DiscoTech server. In Figure 4-3, the disconnection manager components perform these tasks. The disconnection managers in all the nodes have event queues that store events.

When disconnection happens for any reason (discussed earlier in Section 2.2.1), the sender continues sending events but the receiver cannot receive them. At the disconnected state, the DiscoTech server’s Disconnection Manager stores events in its event queue for the disconnected receiver as long as it remains disconnected. The DiscoTech server applies different strategies (discussed in Sections 4.2.1, 4.2.2 and 4.2.3) to the event queue in order to keep its size reduced for handling performance issues (discussed in Section 1.2). In doing so, it can implement
reconnection behaviours three to six (DRAB#3 to DRAB#6, discussed in Section 3.1.6). For example, we can use the strategy *sampling by type* for not displaying unimportant events (DRAB#3, discussed in Section 3.1.6).

Upon reconnection, the DiscoTech server forwards stored events to the receiver. It stores the events in its event queue and applies different strategies (discussed in Section 4.2.4 and 4.2.5) in order to implement reconnection behaviours one and two (DRAB#1 and DRAB#2 as discussed in Section 3.1.6). For example, if we apply a uniform speed-up strategy, the application will display stored events quickly (i.e., reconnection behaviour one, specified in Section 3.1.6). The receiver’s *Disconnection Manager* does not augment any new user interface for displaying the stored events, but passes the events to the receiver’s client application using a callback. The receiver’s client application displays the events to the reconnected user via its existing interface.

The *Disconnection Managers* at the server and the receiver can apply multiple strategies to the event queue at different disconnection timeframes (discussed in Section 2.2.3). The multiple strategies are required to address a trade-off between performance and understandability requirements if a disconnection lasts longer (discussed in Chapter 5).

4.1.1 Summary of the disconnection-aware groupware model

The disconnection-aware model shows that a groupware application can be implemented using either a peer-to-peer or a client/server overlay architecture at the conceptual level. Applications supporting both of these overlays can physically be implemented using a single *underlay network* consisting of participant nodes in a groupware session and a message broadcaster. In order to support disconnection handling, this model shows how to extend functionality of the message broadcaster and participants’ nodes by adding event queues. With this model, we have shown that
some of the strategies for handling disconnection can operate to the event queue of the message broadcaster, while others can operate to the event queue of the receiver node.

4.2 A Five-Dimensional Disconnection Design Space

As discussed in Section 1.2 and in our disconnection-aware groupware model above, while handling disconnection, developers may need to resolve performance issues (e.g., high memory usage, and high feedthrough time) and may need to reduce the lengthy playback time or change the ordering of events. Therefore, developers need to apply different strategies that determine how stored information in the event queues (at both the DiscoTech server and the receiver, discussed above) can be manipulated. However, there is a lack of knowledge about different strategies for handling disconnection (also discussed in sub-problem one, Section 1.2.1). Although there are different approaches for handling disconnection (discussed in Section 2.3), developers are basically well acquainted with the simple ‘store, forward, and replay’. As developers build applications using only the simple mechanisms for disconnection handling, existing groupware applications can only address limited disconnection and reconnection situations (discussed in Section 1.1). On the other hand, there is a wide range of disconnection scenarios with the possibilities of various disconnection and reconnection situations (as discussed in Chapter 3). In order to inform developers about a wide range of strategies for handling disconnection in a codified manner, we have developed a five-dimensional disconnection design space, discussed as follows.

The overall goal of handling disconnection is to help disconnected collaborators rejoin and continue the collaborative work session as smoothly and seamlessly as possible. This involves three main questions from the user’s perspective:
- **What happened during the absence?** The returning user needs to ‘catch up’ on important events and changes that have occurred during the absence. This requirement involves two further issues – what it means to ‘catch up,’ and what constitutes an important change.

- **What is the state of the workspace?** The user needs to understand the current state of the collaboration artifacts; this is not only related to understanding the changes, but also involves a high-level understanding of the workspace (e.g., in a chess game, it is not enough to know that the other player has moved her knight; one also needs to understand the current board state).

- **What is happening now?** The user needs information about current activities in order to smoothly re-integrate into the group’s activity. For example, the user needs the last few messages of a chat conversation in order to understand what is being discussed, and to restart their participation in the conversation.

These requirements involve competing goals: first, the user wants to get as complete an understanding as possible of the happenings that were missed in the disconnection; but second, the above questions also need to be answered as quickly as possible if the user is to re-join the collaboration in a timely fashion.

This implies, for example, that in many cases the events cannot simply be replayed in their original form (since this might take too much time); in addition, this means that the transmission time of stored messages after reconnection is an issue that needs to be considered. From these user-level requirements, we have identified various strategies using five a dimensional design space (shown in Figure 4-4). Along the five dimensions of the design space, groupware messages can be manipulated or altered during or after a disconnection.
4.2.1 Message Formatting

The message formatting dimension shows how to change the way messages are represented and packaged, primarily to reduce volume. This dimension shows developers how to combine a
sequence of stored events into a single event without changing their original content. The strategies falling in this dimension are described as follows:

- **Compression.** Two types of compression techniques, such as lossless and lossy can be applied to reduce message size. A lossless compression technique (such as Zlib, ZIP or General Message Compressor, GMC [38]) can be applied to all the events in event queue after combining them in a single message upon reconnection. A lossy technique can also be applied for the image, video or audio data (e.g., re-encoding JPEGs or MPEGs).

- **Aggregation.** This is a technique for repackaging to incorporate several individual events per message. For example, sending multiple telepointer locations per message saves the overhead of identifying each event as a telepointer. Figure 4-4 shows how a set of lines and circles drawing events can be combined into a single UML actor event that essentially conveys the same information with reduced data volume.

- **Chunking.** This is a technique for adding together several events to create a single larger event. For example, changing a set of character-by-character typing events into a single chat message event; or coalescing several pixel edits into a larger bitmap region event.

4.2.2 Transforming

The transforming dimension shows how to change the content or representation of messages, either to reduce size or interpretation time. Transformations are often paired with visualizations that show the information in a way that is different from the original presentation. The strategies falling in this dimension are described as follows:
- **Averaging.** Some messages (such as position or movement events) can be combined through averaging (e.g., showing the average location of an avatar over one-second intervals, rather than at each movement).

- **Summarization.** Messages can be summarized without changing the basic data representation. For example, text chat could be replaced with a shorter text summary or individual comments made in different pages of a paper document can be summarized together.

- **Representation.** Messages can be processed to create a completely new representation. For example, the text of a chat conversation could be replaced with information about participants’ activity level; similarly, telepointer movements could be replaced with a ‘heatmap’ showing overall activity and tank movements can be replaced with historical traces to show past trails.

### 4.2.3 Sampling

The sampling dimension shows how to select a subset of the stored messages based on a particular property. Sampling techniques imply that some messages are deleted from the event queue. The strategies falling in this dimension are described as follows:

- **Sampling by priority or type.** This is a technique for selecting messages based on an explicit priority designation in the message, or on an implicit designation based on message type. An example of the latter approach is to select messages that involve changes to the data model and discard awareness messages (e.g., telepointer movement, shown in Figure 4-4).
• **Sampling by time.** Messages may be selected either by time period (e.g., randomly retain one message per second, to show the distribution of events), or by time cutoff (e.g., retain the last ten seconds of activity).

• **Sampling by count.** Similar to time sampling, but using the number of messages rather than time as the governing structure. This approach might retain every $N$th message, or just the most recent $N$ messages.

### 4.2.4 Pacing

The pacing dimension shows how to change the temporal spacing of a stream of messages, primarily to reduce the time needed for the user to interpret events on reconnection. The strategies falling in this dimension are described as follows:

- **Timestamp-based uniform speed-up.** Message timestamps can uniformly be altered to decrease playback time; for example, events could be replayed at double speed (shown in Figure 4-4). The order and timing of the messages match their original input, but the timeline is compressed. This allows users to see the rate at which messages were originally typed, but at a faster pace. For example, in the chat application the character-by-character typing events could be replayed at double speed.

- **Artificial uniform speed-up.** A fixed artificial time gap can be added in the events streams in order to display them with a uniform pacing. For example, the chunked comments in the collaborative paper reviewing application can be displayed one after another with a 100 milliseconds gap.

- **Non-uniform time change.** Times can be altered to achieve specific purposes, such as removing or adding gaps in the message stream, or reducing less important sequences.
- *Quick-as-possible playback.* In extreme cases, events can be played with no time delay – that is, messages are processed directly one after another.

### 4.2.5 Ordering

The ordering dimension shows how to change the original ordering of messages in the queue without removing any messages. As with sampling, decisions are made based on properties of the message data. The strategies falling in this dimension are described as follows:

- *Ordering by priority.* Some messages may be considered as more important and can be moved to the front of the queue. Examples include important game events (e.g., hits), or messages that are important for causal processes.

- *Ordering by recency.* In a standard queue, recent messages would arrive last after reconnection; in situations where messages do not have causal dependencies, however, recent messages could be delivered first. For example, a chat system might show the recent parts of an ongoing conversation, and then gradually fill in earlier parts or in a shared drawing application, missed drawing events can be replayed in the reverse order (shown in Figure 4-4) to show the latest events first.

- *Ordering by size.* If small messages can be processed faster, it may be useful to collect them at the front of the queue.

### 4.2.6 Expressiveness of the design space

We have implemented eight groupware applications by studying five substantially different categories of groupware (the chat tool, four shared workspace editors, Tank game, gesture-based game and remote presentation tool, discussed in Section 3.1). These applications show ten behaviours (summarized in Section 3.1.6) related to disconnection handling. We found that most
of the behaviours (e.g., DRAB#1 to DRAB#8, discussed in Section 3.1.6) can be implemented with the design space. Reconnection behaviours one to six (DRAB#1 to DRAB#6, discussed in Section 3.1.6) can be implemented using an individual strategy. For example, first reconnection behaviour (DRAB#1: displaying missed events quickly) can be implemented with the *timestamp-based uniform speed-up* strategy. On the other hand, seventh reconnection behaviour (DRAB#7: presenting missed information with different degrees of richness and fidelity as a disconnection progresses) can be implemented by combining strategies (discussed in Chapter 5). Although the reconnection behaviour eight (DRAB#8, related to the modality fusion requirement) needs special techniques to implement, developers can use strategies such as aggregation or summarization to adopt this behaviour. Therefore, our study implies that the five-dimensional disconnection design space is highly expressive.

Implementation of the ninth reconnection behaviour (DRAB#9: displaying events with smooth transitions effects) will depend on where the data model resides. If the model resides in clients following the peer-to-peer architectural model, the aggregation strategy can be used to manipulate the messages that control the change of the model state. To create the smooth transition effect the initial and final states of a graphical object or entity is required. Therefore, to get the final state after a disconnection, the aggregated messages need to be executed at the reconnected client side. Then, the client can display the change of the entity using the smooth transition effect. In contrast, if the model resides in the server, the chunking strategy can be used to coalesce the state change events into a large single bitmap event. Two bitmap events representing the initial and final states of the model can be stored in the event queue. Upon reconnection these two events can be forwarded to a reconnected client. The client generates the smooth transition effects using the two bitmap events.
The tenth behaviour (DRAB#10: prohibiting connected users interacting with disconnected users) does not fit in the design space, as a groupware application would exhibit this behaviour to users during disconnection, whereas the design space covers the behaviours that groupware applications exhibit upon reconnection.

### 4.2.7 Summary and discussion of the design space

The disconnection design space categorizes a wide variety of strategies for handling disconnection. The first three dimensions of the design space (message formatting, transforming and sampling) primarily address performance issues by reducing the size of the event queue (and for this reason in Figure 4-4, the same colour is used for these three dimensions). On the other hand, the other two dimensions (pacing and ordering) address usability issues upon reconnection by reducing the playback time of events (and for this reason in Figure 4-4, the same colour is used for these two dimensions). Our model of disconnection-aware groupware (discussed in Section 4.1) shows how a groupware application can apply different strategies in order to adopt different disconnection related behaviours.

The design space has three benefits:

- First, it helps developers deeply understand both the notions of users’ requirements upon reconnection and the performance issues of handling disconnection. In doing so, it explains to developers why disconnection handling can be difficult. Consequently, it answers research question one which asks, “what makes disconnection handling difficult for synchronous groupware” (RQ#1, described in Section 1.3), particularly at the design level.
Second, it informs developers about different strategies for handling disconnection. Thus, it answers research question two, which asks, “what are the strategies that developers can use for handling disconnection” (RQ#2, described in Section 1.3).

Third and finally, it breaks down the different approaches of handling disconnection into five basic approaches. Thus, it suggests that developers need to design a toolkit that should at a minimum cover these five basic approaches.

We have developed five different types of disconnection-aware groupware applications where we have applied different strategies from the design space for adopting eight disconnection and reconnection behaviours.

4.3 Summary

In this chapter, we first described an architectural model of a disconnection-aware groupware application. This model shows how different strategies for handling disconnection can be applied in the event queues of the DiscoTech server and receiver nodes. Then, we presented the design space that categorizes different strategies using five design dimensions. Finally, we showed how we can implement different disconnection and reconnection behaviours (discussed in Section 3.1.6) by applying the five different categories of strategies of the design space.
Chapter 5

Disconnection Design Combos

In Chapter 3, we analyzed the requirements for adopting different disconnection and reconnection behaviours in various groupware applications. One of the behaviours specifies that a disconnection-aware groupware application needs to present stored information with different degrees of richness and fidelity as a disconnection progresses (DRAB#7, discussed in Section 3.1.6). The rationale for having this behaviour is the timeline concept (discussed in Section 2.2.4) specifying that user needs for information change in different disconnection timeframes. The timeline concept helps us to come up with solutions for a common disconnection problem dealing with the trade-off between the performance and understandability requirements. We have formulated three disconnection design combos based on this concept for addressing the common problem in three different disconnection design situations.

This chapter first explains the performance and understandability trade-off. After that, it briefly presents three disconnection design combos. Then, it specifies our solutions for the proposed disconnection design combos. After that, it explains how the combos relate to each other. For example, the first two combos are orthogonal to each other, whereas the third one is related to first two combos. Finally, it analyzes how the disconnection design combos can answer research question 4 (RQ#4: can we reuse solutions from one disconnection scenario to other scenarios?" discussed in Section 1.3).
5.1 Performance and understandability trade-off

In software design, trade-offs are fundamentally difficult problems to handle. A trade-off allows a developer to gain something, but at the cost of giving up something else. For example, in a client/server based distributed system, both performance and availability can be increased if the number of servers is increased. However, this architectural decision can negatively impact the security of the system, because an increasing number of servers adds potential points of attack and failures. On the other hand, if the number of servers is reduced in order to make the system secure, it can negatively affect availability and performance [106].

While handling disconnection, developers need to tackle the trade-off between performance and understandability requirements. The trade-off specifies that if developers want to satisfy performance requirement properly, they cannot achieve the understandability requirement and vice versa. In the following, we first discuss the performance and understandability requirements and then explain how the trade-off arises while handling disconnection with an example scenario.

5.1.1 Performance requirement

The performance requirement specifies that when a reconnection happens developers need to deliver and present reconnection information as quickly as possible so that users can resume collaboration in a timely fashion upon reconnection. In order to satisfy the performance requirement, developers need to reduce the volume of accumulated events in the event queue during disconnection by applying different strategies such as sampling, compaction, and summarization (discussed in Section 4.2) and need to reduce the catch-up time of the users by applying speed-up
strategies (discussed in Section 4.2.4). Developers need to address the performance requirement particularly when they consider long-term disconnection.

Developers also need to consider two aspects while characterizing the performance requirement discussed as follows:

- The amount of time that a user wants to spend for receiving and watching reconnection information varies based on durations of disconnections. However, whatever the duration is, the required time to spend should be reasonable to a reconnected user. For example, a user will neither want to spend an hour to see reconnection information nor expect to see reconnection information as soon as she reconnects after one day of disconnection.

- The approach for sending stored events over the network upon reconnection varies, e.g., events can sent all together as a single large message or events can be sent as a group of events in separate smaller messages (discussed in Section 6.2.2.4). Each approach has advantages and disadvantages. For example, if all stored events are sent as a single message after a long-term disconnection, it will be possible to change the ordering of events upon reconnection, but a user might need to spend too long to see the reconnection information contained in the message due to its high transmission and playback time.

5.1.2 Understandability requirement

The understandability requirement specifies that developers have to present reconnection information to users in an understandable manner. By seeing reconnection information, a user will get answers of three user-level questions (what happened during the absence, what is the state of the workspace and what is happening now, discussed in Chapter 4) so that she can get back in a
collaborative session. However, these questions do not apply for all kinds of events. For example, awareness events (e.g., telepointer gesture events) cannot be used to determine the answer of the question “what is the state of the workspace”. In order to get this answer, developers need to use model-update events. On the other hand, model-update events cannot alone reflect the answer of the question “what is happening now”, so awareness events also need to be used for answering this question. Therefore, developers have to determine how to present different types of events in order to achieve the understandability requirement.

5.1.3 An example scenario explaining the performance and understandability trade-off

In order to satisfy the performance requirement, developers might apply a single strategy considering a particular duration of disconnection, e.g., if disconnection continues between zero to one minute, a compaction strategy is applied or if disconnection lasts more than a minute to one hour, a summarization strategy is applied. However, this procedure will not allow developers to present the most recent stored information with original fidelity. Conversely, watching most recent events with original fidelity is important to know the answer of the question “what is happening now” (discussed above in Section 5.1.2) and hence to satisfy the understandability requirement. Therefore, developers might decide not to use any compaction or summarization strategy, rather they might decide to present all stored information as quickly as possible using the timestamp-based uniform speed-up strategy (discussed in Section 4.2.4). However, although this procedure will tell users what is happening now, it might not satisfy the performance requirement. For example, if telepointer events are sent over the network with the rate of 30 telepointer messages per second and if disconnection lasts for an hour, there will be in total at least 32 MB (Megabytes) of telepointer data to store in the event queue. In this calculation, a telepointer event is stored with the format <telepointergesture, x, y> (discussed in Section 2.1.1.1) and along with some basic
information, such as ‘Timestamp’ and ‘Username’. As a result, this amount of data would require excessive memory usage and bandwidth consumption, and will ultimately take too long to be transferred over the network depending on the Internet connection speed. Moreover, this approach will also take prolonged playback time if developers use the uniform speed-up strategy (such as double speed).

In this scenario, we can see that there is a trade-off between the performance and understandability requirements while handling disconnection.

5.2 Timeline Concepts to Address the Trade-off

The timeline concept specifies that users’ requirements vary at different disconnection timeframes (discussed in Section 2.2.4). Users want to see different degrees of richness in reconnection information as a disconnection progresses. For example, in the telepointer gesture scenario (mentioned above), users might be interested to see a quick playback of recent gestural activities to resume synchronous interactions upon reconnection, whereas they might prefer to see a ‘heatmap’ representation (discussed in Section 3.1.2.2) of older telepointer gesture events. Therefore, in order to address the trade-off in the telepointer scenario, developers can gradually reduce the fidelity of telepointer gesture events by applying different strategies at different timeframes. They can keep most recent events in their original form to reflect the answer of the user-level question “what is happening now”, they can compact older events to reflect the answer of the question “what happened in the recent past” and they can apply the heatmap summarization strategy to know “what happened in the past”. The compaction and summarization strategies will address the performance requirement, whereas the patterns of applying these strategies based on
the ages of stored events will address the understandability requirement. Therefore, by using the timeline concept, we can manage the trade-off.

5.3 Disconnection Design Combos

The trade-off is difficult for developers to handle because its solution varies depending on disconnection contexts, disconnection design situations and durations of disconnections. There are three disconnection design situations where we identify this problem:

- When handling awareness events/messages (e.g., telepointer gesture events in a collaborative paper reviewing application, discussed in Section 3.1.2.2).

- When handling model-update events (e.g., use case diagram element dropping events in UML use case diagram editor, discussed in Section 3.1.2.3).

- When handling events that need to be compacted during disconnection and need to be presented with pacing upon reconnection in their original form (e.g., character-by-character comment typing events in a collaborative paper reviewing application as discussed in Section 3.1.2.2).

Different events have different requirements for reliability. E.g., the tank move events of the Tank game have lower reliability requirements than the use case element dropping events of the UML use case diagram editor. If use case element dropping events are discarded, it is not possible to display the UML use case diagram consistently to reconnected users. On the hand, the older Tank move events can be discarded without affecting the consistency of the model. Therefore, the precise distinction between model-update and awareness events depend on the reliability requirements of
events. On the basis of this fact, we classify model-update events as critical events reduction of which will create inconsistency in data model. On the other hand, we classify awareness events as the non-critical events reduction of which will not create inconsistency in data model.

This thesis proposes three disconnection design combos (such as fidelity-reducer, accumulator and compacted-pacer) to address the trade-off in the three disconnection design situations (summarized in Figure 5-1). These combos divide the total duration of a disconnection into different disconnection timeframes considering the timeline concepts (as discussed in Section 5.2).

Figure 5-1: Selecting disconnection design combos
Three disconnection design combos offer a suitable strategy to address an individual issue corresponding to a particular timeframe. Consequently, the proposed combos select and combine multiple strategies for addressing the overall disconnection issue (i.e., the trade-off).

We design the combos based on our development experience with eight disconnection-aware synchronous groupware applications, which include five different types of groupware, such as a chat tool, four shared workspace editors, a Tank game, a gesture-based game and a remote presentation tool (discussed in Section 3.1). These applications were developed using our DiscoTech toolkit (discussed in Chapter 7).

The advantage of the disconnection design combos are that they provide structured and reusable knowledge to developers for solving a common disconnection problem. Without having the knowledge of disconnection design combos, they might end up with disconnection solutions that will not effectively address the trade-off. As these combos offer reusable solutions, they are also used for investigating our fourth research question (can we reuse solutions from one disconnection scenario to other scenarios?). Three combos are summarized as follows.

**5.3.1 Fidelity-reducer combo**

The *fidelity-reducer* combo shows how to select and combine strategies for addressing the trade-off in the design situation of handling disconnection for awareness events. During disconnection, this combo suggests applying a new strategy on the stored events as they grow older and enter into a longer disconnection timeframe. The applied strategy in an older disconnection timeframe reduces more fidelity of the events than the strategy applied in a newer timeframe. In this way, the *fidelity-reducer* combo addresses the performance requirement. Upon reconnection, users can see
recent awareness events with higher fidelity than the older events, allowing the combo to address the understandability requirement. We illustrated the fidelity-reducer combo using the paper-reviewing scenario discussed in Sections 5.1.3 and 5.2.

5.3.2 Accumulator combo

The Accumulator combo shows how to select and combine strategies for addressing the trade-off in the design situation of handling disconnection for model-update events. During disconnection, this combo applies a new strategy on the stored events as they grow older and enter into an older disconnection timeframe. The applied strategy in an older disconnection timeframe aggregates events that can do larger update of the model at a time than the strategy applied in a newer timeframe. For example, in the shared drawing editor (described in Section 3.1.2.1), we can keep recent sketch events in their original forms and then aggregate the older events using a time-based aggregation strategy, and then aggregate all the much older events into a single bitmap image. This will allow developers to satisfy the performance requirement by keeping memory usage low and at the same time will allow them to present stored events in an understandable format as shown in Figure 3-5(b).

5.3.3 Compacted-pacer combo

This combo shows how to select and combine different strategies for addressing the performance and understandability trade-off in the design situation of handling disconnection for the events that need to be compacted during disconnection and need to be presented in their original form with pacing upon reconnection. For example, in the collaborative paper reviewing application, developers need to compact character-by-character typing events using the chunking strategy so
that they can present the comments one after another with some added artificial delays. However, a large volume of compacted messages will increase the message transmission time at some point, and will prolong the catch-up time. Consequently, this will not satisfy the performance requirement. In order to address the performance requirement in this situation, if developers use a transformation strategy, it will not preserve the original content of the events and will hamper the understandability requirement. For example, in case of the collaborative paper reviewing application, if a comment summary strategy (discussed in Section 3.1.2.2) is applied, it will not allow users to see at which paragraph a collaborator made the comment, making it difficult for her to understand the context of the comment. In order to get around this trade-off in this design situation, the compacted-pacer combo suggests that developers need to apply a playback strategy on the events falling in an older timeframe that will display the events quicker than the strategy applied in a newer disconnection timeframe (further discussed in Section 5.3.3).

5.4 Specification of Fidelity-Reducer Combo

As we discussed in Section 5.3.1, the fidelity-reducer combo addresses the performance and understandability trade-off when handling awareness events. In Figure 5.1, we show a structure of the fidelity-reducer combo where the total duration of disconnection is divided into three timeframes, short (0-‘k’ seconds), medium (‘k’ seconds – ‘m’ minutes) and long (‘m’ minutes- ‘n’ hours), considering the fact that awareness events can be presented in three forms (original, compacted and transformed). The values of the variables (‘k’, ‘m’ and ‘n’) used for the timeframes are application specific. For example, timeframe values for the context of “moving tanks in a Tank game” are different from those of “gesturing using telepointers in a collaborative paper reviewing application”.

94
The **fidelity reducer** combo suggests applying multiple strategies to the event queue based on the disconnection timeframes and the age of events. The appropriate strategies for each timeframe are described as follows:

- **For a short disconnection timeframe** (0 – ‘k’ seconds), developers do not need to apply any strategy for manipulating stored events. Events that fall in this timeframe are low in number and hence, do not use much space in the event queue. Thus, upon reconnection, no performance issues arise for transferring the events over the network. At this timeframe, events remain fresh and convey recent information. By presenting the small number of events with full fidelity, developers can help users rejoin a collaborative session quickly upon reconnection. Therefore, stored events falling in the short disconnection timeframe can be presented with full fidelity while managing the trade-off.

- **For a medium disconnection timeframe** (‘k’ seconds – ‘m’ minutes), developers should apply a compaction strategy (i.e., chunking, averaging, compression or aggregation, discussed in Section 4.2) to the events which ages fall in the range ‘k’ seconds – ‘m’ minutes. The events within the range 0-‘k’ seconds should be kept in their original form.
The compaction strategy does not change the original content of the stored events. Applying such a compaction strategy at this timeframe can manage the trade-off because it reduces the event queue size and at the same time preserves the original content of events falling at this timeframe. For example, at the timeframe ‘60 sec– 10 min’, the **averaging** strategy is applied on the telepointer gesture events described in the example scenario in Section 5.1.3 (where 30 telepointer events are generated per second); if the **averaging** strategy keeps one event (as an average value) from the events whose age gaps fall in one second time interval, among 18000 telepointer events, it will discard about 17600 telepointer events and will keep about 600 events in the event queue. As a result, upon reconnection, events falling at this medium disconnection timeframe will be presented in their original form but will convey information with lower fidelity than the events falling at the short disconnection timeframe.

- For a **long disconnection timeframe** (‘m’ minutes to ‘n’ hours), developers should apply a transformation (i.e., **summarization** or **presentation**, discussed in Section 4.2.2) strategy to the events that are older than ‘m’ minutes. It will change the original content and representation of these events. By transforming the events into a different representation, it can discard a large volume of accumulated events, allowing the stored events to transfer over the network as soon as possible. For example, in the scenario mentioned above in Section 5.1.3 (where 30 telepointer events are generated per second) if the long disconnection timeframe ranges from 10 min to 1 hour, about 90,000 telepointer events will accumulate within this timeframe. By applying the heatmap summarization strategy, we can transform these large number of events into a single heatmap event, which will take reduced time to be delivered over the network. Information presented in transformed form
also takes reduced time to play back upon reconnection. Thus, the performance requirement can be satisfied. As mentioned earlier reconnected users prefer to see an overview of outdated missed events. For example, the heatmap display (shown in Figure 3-7(a)) of older telepointer gesture events will satisfy this requirements. Thus, by applying a transformation strategy at this timeframe, we can present reconnection information with further reduced fidelity while satisfying both the performance and understandability requirements and can manage the trade-off.

5.4.1 Applicability of the combo

Different disconnection scenarios consisting of different user actions can be classified under this category, e.g., “gesturing over paragraphs using telepointers for discussing their contents in the collaborative paper reviewing application”, “moving a tank of a player in the Tank game”, or “typing chat messages in the chat tool”. In these scenarios, if reconnection information is presented with lower fidelity with an increasing disconnection period, it does not hamper the understandability requirement. All of these applications are discussed in Section 3.1.

Table 5-1: Suggested timeframe values and strategies in fidelity-reducer combo

<table>
<thead>
<tr>
<th>User actions in different disconnection scenarios</th>
<th>Short timeframe, No strategy</th>
<th>Medium timeframe, Compaction strategy</th>
<th>Long timeframe, Transformation strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gesturing over paragraphs using telepointers while discussing their contents in the collaborative paper reviewing application</td>
<td>0 - 60 sec, none</td>
<td>60 sec - 10 min, Averaging</td>
<td>&gt;10 min, Heatmap</td>
</tr>
<tr>
<td>Typing character-by-character comments in the collaborative paper reviewing application</td>
<td>0-60 sec, none</td>
<td>60 sec- 10 min, Chunking</td>
<td>&gt;10 min, Comment Chat frequency Summary</td>
</tr>
<tr>
<td>Typing chat messages in the chat tool</td>
<td>0 - 60 sec, none</td>
<td>60 sec– 10 min, Chunking</td>
<td>&gt;10 min, Comment Summary</td>
</tr>
<tr>
<td>Moving a tank of a player in the Tank game</td>
<td>0 - 10 sec, none</td>
<td>10 sec – 30 sec, Averaging</td>
<td>&gt;30 sec, Historical traces</td>
</tr>
</tbody>
</table>
In Table 5-1, we show how this combo is used in different disconnection scenarios using our three defined disconnection timeframes above. However, we see that timeframe values vary from one scenario to another. In this table, we set the timeframe values based on our own usage experience with the applications. While setting the timeframe values, events were generated continuously, meaning not all users remain idle in a collaborative session and at least one user manipulates workspace artifacts or embodiments.

### 5.4.2 Pros and cons analysis

By applying different strategies in different disconnection timeframes, the fidelity-reducer combo manages the performance and understandability trade-off, compared to applying a single strategy throughout the total disconnection period. This combo changes and transforms the original contents of the events based on their age, e.g., no strategy is applied on the recent events and a transformation strategy is applied on older events. In doing so, this combo allows reconnected users to get a bird’s eye-view of what happened in the past, and a concrete idea of what is happening now. However, as this combo reduces the original fidelity of older events, there always remains the possibility that it might not able to convey some important past activities. By using an appropriate set of strategies, developers can mitigate this disadvantage to some extent. For example, for the user action “moving a tank in a Tank game” or “gesturing over a conference paper using telepointer in a collaborative paper reviewing application”, developers need to transform the contents of missed events into different representations for addressing the long disconnection timeframe. However, for the moving tank event, showing ‘historical traces’ (Figure 3-12 (b)) would be a good option for the reconnected users, whereas for the gesturing event, showing a ‘heatmap’ (Figure 3-7) display would be a good
option for the reconnected user. Therefore, developers need to choose a strategy based on the context for maintaining better understandability while degrading the fidelity of stored information.

5.4.3 Implementation issues

In order to use the combo, developers need to deal with applying a strategy to the events that are already manipulated by another strategy applied in a different timeframe. For some scenarios, a strategy might need to use the manipulated events, whereas for other scenarios, it might need to use the original ones. For example, if the averaging strategy is applied at the medium disconnection timeframe on the telepointer gesture events, the strategy removes the original telepointers events and keeps the new events that contain the average positions. Consequently, the ‘heatmap’ summarization strategy applied at the long disconnection timeframe cannot access the original telepointers events for reflecting the activities of connected users at a fine granular level. On the other hand, in the same application if the chunking strategy is applied at the medium disconnection timeframe to the character-by-character typing comment events, the summarization strategy can use the chunked comment typing events to create the ‘comment summary’ at the long disconnection timeframe. Although the summarization strategy can use the original comment typing events, it needs to apply the chunking algorithm once again to those messages potentially increasing the computational overhead. Therefore, developers need to determine how different strategies will work together at different disconnection timeframes. We address these implementation issues in the DiscoTech toolkit (discussed in Section 7.7).
5.4.4 Where the combo does not work

Developers should use this combo to address different disconnection timeframes for awareness events. However, some awareness events in some contexts are valuable for real time collaborations, but might not be important for asynchronous interactions. For example, in case of the context “moving telepointers while drawing a sketch”, a user might be interested to see only most recent telepointer movement events in order to resume synchronous interactions, whereas they might not want to see older events in the form of heatmap. Therefore, for these kinds of awareness events developers might avoid using this combo for simplicity; instead, they can just play back the latest events even after a long-term disconnection. However, it is completely developers’ choice based on an application context.

Moreover, in some situations, developers need to address the modality fusion requirement (discussed in Section 3.1.5.1) by combining different types of event streams at a semantic level. However, this combo can be applied only a single type of event and does not take into account the modality fusion requirement.

5.4.5 Summary of fidelity-reducer combo

This combo helps developers address the performance and understandability trade-off when handling disconnection for awareness events. We have derived this combo from our development experiment with eight groupware applications (discussed in Sections 3.1 and 7.9). This combo can be used to handle disconnection for different scenarios (as shown in Table 5-1) However, this combo is not suitable when developers need to resolve the modality fusion requirement (see Section 3.1.5.1).
5.5 Specification of Accumulator Combo

As we discussed in Section 5.3.2, the accumulator combo addresses the performance and understandability trade-off when handling model-update events. In Figure 5-3, we show a structure of the fidelity-reducer combo where we divide the total duration of disconnection into three timeframes: short, medium and long. Three time divisions are done considering the fact that model-update events can show the change of the workspace in three ways: with original incremental update, intermediate incremental update and single large update.

In our proposed solution, the accumulator combo suggests applying multiple strategies to the event queue based on the disconnection timeframes and the ages of events. The appropriate strategies for each timeframe are described as follows:

- For the short disconnection timeframe (0 – ‘k’ seconds), developers do not need to apply any strategy on the accumulated events (as the fidelity-reducer combo).

![Accumulator disconnection design combo](image)

**Figure 5-3: Accumulator disconnection design combo**
- For the medium disconnection timeframe ('k' seconds – 'm' minutes), developers should apply a lossless compaction strategy (chunking or aggregation) on the events whose ages fall in the medium disconnection timeframe. As with the fidelity-reducer combo, the events within the range 0- 'k' seconds should be left in their original form. The lossless compaction strategy combines a group of events (that need to be conveyed without changing their contents) of the same type into a single event. At this stage, a lossless compaction strategy is suitable because a moderate volume of events will be accumulated, so sending multiple similar events in a single message saves the bandwidth and overhead of identifying each event as a particular type. In addition, sending such a large volume of events without compaction would require increased bandwidth and increased playback time. In contrast, applying aggregation-to-state (which combines events into a single large event) would unnecessarily increase the computational overhead of the receiver for interpreting the single aggregated event and will not satisfy the understandability requirement. By applying a lossless compaction strategy, the stored events will be presented without changing their content, but with reduced temporal spacing, allowing the combo to present information in an understandable format.

- For the long disconnection timeframe ('m' minutes - 'n' hours), developers need to use the aggregate-to-state strategy to the events that are older than ‘n’ minutes. This strategy combines all the model-update events that fall in this timeframe into a large update event without changing their content. For example, all the accumulated events such as the dropping use case diagram elements, drawing connectors and typing use case names are aggregated into a single diagram state. Aggregate-to-state is suitable at this timeframe because it will reduce memory and bandwidth consumptions as a disconnection progresses.
Upon reconnection, developers might need to parse the aggregated event into separate events. As the *aggregate-to-state* strategy does not change the original representation and content of events, after a long term disconnection, reconnection information can be presented with full fidelity, allowing the combo to satisfy the understandability requirement. There will be no temporal spacing among events, allowing the combo to display events quickly and to satisfy the performance requirement.

### 5.5.1 Applicability

A wide variety of disconnection scenarios consisting of different user actions can be classified under this category. For example, “drawing lines using telepointers in a shared drawing application”, “dropping use case diagram elements on the drawing palette in a shared UML use case diagram editor”, or “copying and pasting code in a shared coding editor”. For such scenarios, developers need to make sure that the same information as original is conveyed upon reconnection. e.g., discarding even a single ‘drop circle’ event will inconsistently present the data model.

**Table 5-2: Suggested timeframe values and strategies in accumulator combo**

<table>
<thead>
<tr>
<th>User actions in different disconnection scenarios</th>
<th>Short timeframe, no strategy</th>
<th>Medium timeframe, Lossless compaction</th>
<th>Long timeframe, Aggregate-to-State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropping a use case diagram element in a shared UML use case editor</td>
<td>0 - 10 sec, none</td>
<td>10 sec – 10 min, Aggregation</td>
<td>&gt;10 min, Aggregation</td>
</tr>
<tr>
<td>Writing character by character code in a shared coding editor</td>
<td>0-10 sec, none</td>
<td>10 sec – 10 min, Chunking</td>
<td>&gt;10 min, Aggregation</td>
</tr>
<tr>
<td>Copying and pasting code in a shared coding editor</td>
<td>0 – 60 sec, none</td>
<td>60 sec – 10 min, Aggregation</td>
<td>&gt;10 min, Aggregation</td>
</tr>
<tr>
<td>Drawing lines using Telepointers in the shared drawing editor</td>
<td>0 - 60 sec, none</td>
<td>60 sec -10 min, Aggregation</td>
<td>&gt;10 min, Bitmap Image</td>
</tr>
</tbody>
</table>
All the applications are discussed in Section 3.2. In Table 5-2, we show how this combo is used in different disconnection scenarios using our three defined disconnection timeframes above. However, we see that timeframe values vary from one scenario to another. As mentioned in the *fidelity-reducer* combo, these timeframe values are set based on our usage experience with the applications.

### 5.5.2 Pros and cons analysis

The benefit of using this combo is that it controls the size of the accumulated messages while conveying the original information and maintaining the understandability of missed information. However, developers need to keep different issues in mind when using this combo. If aggregation is done based on the types of events, it might break the ordering of events. For example, if the aggregation strategy combines the ‘ellipse dropping’ events in one event, and the ‘connector drawing’ events in another event, it might occur that some dropping events generated after the connector drawing events will appear first. Therefore, developers need to keep in mind whether ordering is an issue in presenting the reconnection information. If this is an issue then developers need to do aggregation based on a time interval. However, in this case, the mixing of different types of events into a single event may cause issues with parsing the stream of the aggregated events upon reconnection. Therefore, developers need to balance these pros and cons by using the appropriate strategy based on the context of use. In addition, developers need to parameterize the strategies so that multiple event types can be considered for aggregation.
5.5.3 Implementation issues

This combo also has the same implementation issues as of the fidelity-reducer combo. For example, developers need to consider, how the aggregate to state strategy at the long timeframe will use the aggregated events at the medium timeframe. The proposed solution is discussed in Section 7.7.

5.5.4 Where the combo does not work

Similar to the fidelity-reducer combo, this combo might not be suitable if developers need to consider the modality fusion requirement (discussed in Section 3.1.5.1).

5.5.5 Summary of accumulator combo

This thesis proposes the accumulator combo for addressing different disconnection timeframes for model-update messages. It shows how to select and combine strategies to balance the performance and understandability trade-off. Although this combo can be used to handle disconnection for different scenarios (Table 5-2), it might not be suitable when the modality fusion requirement needs to be taken into account.

5.6 Specification of Compacted-Pacer Combo

As discussed in Section 5.3.3 the compacted-pacer combo deals with the performance and understandability trade-off in the design situation of handling the events that require to be compacted during disconnection and to be presented in their original form with pacing. A solution of the combo is structured in Figure 5-4.
Figure 5-4: Compacted-pacer disconnection design combo

In the figure (Figure 5-4) we divide a particular disconnection timeframe (‘k’ sec – ‘n’ min) into three more timeframes (short, medium and long) considering three different types of pacing.

In our proposed solution, this combo suggests that in a particular disconnection timeframe (k sec – n min), if a compaction strategy (e.g., aggregation, chunking and averaging) is applied on the events of the event queue, upon reconnection the compacted events can be presented with the following rules in order to address the trade-off:
If the ages of events fall at the short timeframe (′k′ sec – ′l′ sec) events are displayed quickly but with constant pacing. If the timestamp relationships of events are preserved, timestamp-based uniform speed up strategy can be applied on the event streams. On the other hand, if timestamp relationships are not preserved, the artificial uniform speed-up strategy can be applied. Either of the strategy will show a steady rate of change of the workspace and will allow users to understand the most recent missed activities.

For the events falling at the medium timeframe (′l′ sec – ′m′ min), developers can apply increasing speed-up strategy. This strategy adds an artificial time gap between events in a decreasing manner until the time gap value reaches to a certain threshold, e.g., ten milliseconds. It will give an opportunity to users to change the workspace by the older events quickly but smoothly. Thus, this strategy will satisfy both the performance and understandability requirements.

For the events falling at the long timeframe (′m′ min – ′n′ min), developers can apply as quick as possible strategy in order to display the events one after another without any delay. It will allow users to see the much older events without spending that much time and will cut short the playback time.

5.6.1 Applicability

A wide variety of disconnection scenarios can be characterized by this combo. All the scenarios mentioned in the fidelity-reducer and accumulator combos can be characterized by this combo (shown in Table 5-1 and Table 5-2). In particular, this combo is suitable for the medium disconnection timeframe because at this stage for the both combos (fidelity-reducer and accumulator) a compaction strategy is applied.
Table 5-3: Suggested timeframe values and strategies in compacted-pacer combo

<table>
<thead>
<tr>
<th>User actions in different disconnection scenarios</th>
<th>Short timeframe, Constant speed-up</th>
<th>Medium timeframe, Increasing speed-up</th>
<th>Long timeframe, As quick as possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gesturing over paragraphs using telepointers in a collaborative paper reviewing application</td>
<td>60 sec - 3 min, Timestamp-based uniform speed-up</td>
<td>3 min- 8 min, Increasing speed-up,</td>
<td>8 min – 10 min,</td>
</tr>
<tr>
<td>Typing character chat messages in a chat application</td>
<td>60 sec – 2 min, Artificial uniform speed-up,</td>
<td>2 min- 8 min, Increasing speed-up</td>
<td>8 min – 10 min, Non-uniform speed up</td>
</tr>
<tr>
<td>Drawing lines using telepointers in a shared drawing application</td>
<td>60 sec – 2 min, Artificial uniform speed-up,</td>
<td>2 min- 5 min Increasing speed-up</td>
<td>5 min – 10 min, Progressive quick speed-up</td>
</tr>
</tbody>
</table>

In Table 5-3, we show that how these combos are reusable for different disconnection scenarios, whereas the timeframe values vary. As of the other two combos, we set the timeframe values based on our usage experience.

5.6.2 Pros and Cons Analysis

This combo is particularly suitable if a compaction timeframe is not too short and not too long during disconnection. For a too short compaction period, it will increase the computation complexity, and for a too long compaction period, it will not address the performance and understandability trade-off. This combo suggests that if a developer uses a ‘timestamp preserver compaction’ strategy during disconnection, upon reconnection a uniform speed-up strategy (e.g., double speed playback) can be used to replay the compacted events. However, there could be cases where time gaps between two compacted events might vary a lot. For example, when a collaborator performs gesturing on a paragraph, he may pause for thinking. The pause times might be long for some cases (e.g., more than a few seconds) and can increase the time gap between two averaged events. In this case, if developer uses double speed playback, the pause times would unnecessarily prolong the playback time. In order to reduce the time gaps for the pauses, developers can either
use the artificial uniform speed up strategy, or developers can implement the uniform speed up strategy in a way that will inject a particular time gap (e.g., ten milliseconds) if it finds a time gap that exceeds a threshold value, such as 1000 ms. However, while using the artificial uniform strategy, developers need to pick gaps in a way that it neither presents events very quickly nor very slowly. Depending on the context of information, developers need to pick reasonable time gaps.

### 5.6.3 Implementation Issues

As the other two combos, for using this combo, developers do not need to think of the implementation issues for combining strategies, e.g., they do not need to think how a strategy will work on the events manipulated by another strategy in the other timeframe. However, in order to have an effective use of this combo, all the stored events have to be received by the receiver all at once (discussed in Section 5.1.1). Developers also need to consider some of the issues while implementing the timestamp-based and artificial uniform speed-up strategies. For example, for the artificial uniform speed-up strategy, they can parameterize the implementation of the strategy with different time gaps so that events can be displayed with varying speed, which would increase the reusability of the strategy. Similarly, the timestamp-based uniform speed-up strategy can be parameterized for displaying events with variable pacing, and can be implemented in a way that removes any unnecessary big gaps between events (as discussed in Section 5.6.2).

### 5.6.4 Where the combo does not work

There are a few situations where the combo does not work well. For example, this combo is not suitable for a slow-paced game, such as a gesture-based collaborative game for older adults (discussed in Section 3.1.4).
5.6.5 Summary of compacted-pacer combo

This combo balances the trade-off by using the compaction strategy and different replay strategies in the design situation when events need to be presented with pacing in their original form. Although this combo is applicable for a wide variety of user actions (shown in Table 5-3), it is not suitable for a slow-paced game (discussed in Section 3.1.4).

5.7 Relationship with the Three Disconnection Design Combos

The compacted-pacer combo is related to both the fidelity-reducer and accumulator combos (discussed in Sections 5.4 and 5.5). As per the solutions (Figure 5-2 and Figure 5-3), both fidelity-reducer and accumulator combos apply compaction strategies in the medium disconnection timeframe, whereas compacted-pacer combo addresses the trade-off that can arise due to the application of compaction at some point of disconnection. Therefore, if developers use either the fidelity-reducer or the accumulator combo during disconnection, upon reconnection they can also use the compacted-pacer combo if needed. However, if developers simply use a compaction strategy during disconnection, they can apply the compacted-pacer combo.

5.8 Summary and Discussion of Disconnection Design Combos

In this chapter, we described three common design situations where the trade-off between performance and understandability requirements need to be tackled. In order to address the trade-off problem in three disconnection design situations, we specified three disconnection design combos, such as fidelity-reducer, accumulator, and compacted-pacer. These combos offer reusable solutions and inform trade-offs and implementations issues for choosing a particular set of strategies. The fidelity-reducer and accumulator combos combine strategies that reduce event
queue size, whereas compacted-pacer combo combine strategies that also reduce event queue size but also compresses timeline of events to reduce playback time.

Disconnection design combos show how solutions of a disconnection problem can be generalized for a particular message or event type with varying disconnection timeframe values. Thus, it gives us the answer of research question 4 (RQ4: can we reuse solutions from one disconnection scenario to other scenarios, discussed in Section 1.3) with two constraints: types of events and timeframe values. Therefore, it is possible to generalize solutions from one disconnection scenario to other scenarios if user actions of the scenarios generate same type of events (such as awareness) and if developers set the timeframe values based on the context of use.
Chapter 6
Design and Specification of A Plug-in Architecture

In Chapter 4, we discussed different strategies for handling disconnection, and in Chapter 5, we explained how these strategies can be combined to address long-term disconnections. In this chapter, we discuss how we can implement these strategies in a simple and flexible manner using a plug-in based software architecture. This architecture provides codified solutions to Sub-problem 3 (no simple and flexible development infrastructure exists, discussed in Section 1.2.3). The plug-ins are code units that implement the strategies from the design space. We used the term ‘plug-ins’ because code units serve as parameters to the architecture and can customize or extend the default behaviour of the architecture without changing its basic structure. In this chapter, we first discuss why a plug-in architecture is important for handling disconnection. Then, we provide an in-depth specification of the plug-in architecture based on our background study on software architecture [84, 85, 108]. After that, we describe how the plug-in architecture can achieve our desired quality goals. Finally, we provide a summary of the plug-in architecture.

6.1 Why Plug-in Architecture?

To date there is no development infrastructure or toolkit that developers can use in a simple and flexible manner in order to handle disconnection for different disconnection scenarios and timeframes. Previous work in Computer Supported Cooperative Work (CSCW) has shown the kinds of problems that disconnection can cause, and has proposed several solutions for handling reconnection [43], late entrance (e.g., [2, 10, 11, 49, 51, 59]), periods of asynchrony [71,73], or network faults [8, 48, 88, 100]. However, existing techniques handle only a limited number of
disconnection and reconnection situations. For example, current solutions focus on short-term disconnections, and hence do not address the user-level requirements for watching a summary representation of older missed events. Moreover, existing techniques do not provide simplicity and flexibility to developers for implementing application specific disconnection and reconnection behaviours. While handling disconnection with these techniques, they even need to worry about basic disconnection handling issues, such as disconnection identification, queuing and reconnection.

Therefore, developers need to hard code solutions to address disconnection-handling issues, increasing the challenge of developing groupware applications. Developers need to address difficult issues, such as disconnection detection, event queuing, and reconnection. As we have discussed earlier, there is a wide range of disconnection and reconnection behaviours (discussed in Section 3.1.6). Developers therefore need to implement different strategies (discussed in Section 4.2) for adopting the behaviours in their applications. Moreover, disconnections may occur over different disconnection timeframes, requiring developers to combine different strategies that can manage the performance and understandability trade-off (discussed in Chapter 5). While combining strategies, developers need to consider different implementation issues (discussed in Sections 5.5.3 and 7.7). Further, they need to satisfy the *modality fusion* requirement (discussed in Section 3.1.5.1). Therefore, handling disconnection becomes yet another complicated aspect of the development of groupware.

In order to help developers implement strategies for handling disconnection in a simple and flexible manner, we design a plug-in software architecture. The main advantage of this architecture is that it separates the basic disconnection handling parts that take care of disconnection identification,
queuing and reconnection from the parts that implement the strategies. As a result, developers need to implement the basic architectural components only once and can reuse them for implementing different disconnection-aware groupware applications. This architecture implements the strategies as plug-ins. The plug-ins can be reusable software modules or classes or can be partially reusable classes or application specific classes. The plug-in architecture is capable of accommodating three types of plug-ins based on their reusability, summarized as follows (for details see Section 7.2):

- **Fully generic plug-ins** address standard issues such as message formatting, sampling, and ordering in the data that are queued during a disconnection;

- **Partially generic plug-ins** are parameterized to allow tuned behaviour (e.g., to allow reordering with priority for certain message types); and

- **Customized plug-ins** allow application-specific transformations of data based on the system’s specific needs (e.g., changing movement data to a ‘historical trace’ representation).

It is possible to implement disconnection handling for different disconnection scenarios and timeframes by using the three levels of plug-ins above. In order to implement new disconnection behaviours in a groupware application, developers either can simply reuse existing plug-ins or can create their own custom plug-ins by reusing the basic components. In particular, by having the customized plug-in option, it is possible to satisfy the modality fusion requirement. This option allows developers to write application specific strategies that combine two or more different event streams, while reusing the basic architectural components. The plug-ins can be implemented in a way which will allow the developers to apply a strategy to the events in the event queue based on
their age falling within a particular disconnection timeframe. In doing so, developers can combine different strategies discussed using the disconnection design combos (Chapter 5). Developers can parameterize a plug-in in order to specify the timeframe. However, developers need to do more tasks for combining strategies discussed in Section 7.7. In the following sections, we specify the plug-in architecture in detail.

6.2 Specification of Plug-in Software Architecture

While there are several definitions of software architecture, a commonly used definition of software architecture is as follows (Bass et al. [4]):

“The software architecture of a program or computer system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them”.

Software architecture is designed by decomposing a large problem in smaller problems. In order to resolve the smaller problems, functionality of its components are characterised at an abstract level while hiding low level implementation details. Components and their relationships play vital roles for the performance, ease of modification, and other properties desired by users and programmers of the application [17, 93].

Following the definition of Bass et al., the designed plug-in-based software architecture in this thesis defines different architectural components and relationships among the components. Figure 6-1 shows a UML deployment diagram of the plug-in architecture, explaining how different components and their relationships can handle disconnection in synchronous groupware. In this figure, the black box components are reusable across different groupware applications without any
modifications. On other hand, the gray and white box components are the partially generic and custom plug-ins (discussed in the above section). A groupware application handles disconnection exploiting these components.

Figure 6-1: A plug-in architecture

Figure 6-2: Disconnection-aware groupware model
The functionality of different architectural components of the plug-in architecture are specified based on our disconnection-aware groupware model (discussed in Section 4.1 and shown again in Figure 6-2). The disconnection-aware groupware model describes a message broadcasting underlay network that shows connections among participant nodes in a groupware session. The underlay network defines three nodes: sender, receiver, and DiscoTech server. The sender and receiver nodes are the participant nodes containing instances of a full groupware application (in case of peer-to-peer overlay architecture, discussed in Section 4.1) or partial instances of a groupware application (in case of client/server overlay, discussed in Section 4.1). The DiscoTech server contains the message broadcaster component that relays messages from a sender to receivers. There could be multiple receivers, where sender and receiver can switch their roles. In order to support disconnection handling, we add storing, compacting, and forwarding of events capabilities to the message broadcaster. We also extend the functionality of the groupware application by adding the capability to store and replay stored events.

In Figure 6-1, architectural components at the DiscoTech server are shown in the left side box, whereas architectural components at the sender or receiver are shown in the right side. In the following, we first present mediator and infrastructural components that can be reused across different applications. Then, we describe the plug-ins that allow the architecture to adapt new disconnection and reconnection behaviours without requiring to change the mediator and infrastructural components. We also specify the relationships between different components.

6.2.1 Mediator components

There are two mediator components that manage interactions between different components in the plug-in architecture and the groupware application, discussed as follows.
6.2.1.1 DiscoTechClient

The *DiscoTechClient* is a mediator component that maintains interactions between the *Groupware Application* and other architectural components as shown in Figure 6-1. The groupware application generates events and sends events including event types to the DiscoTech server using the *DiscoTechClient* component. This component saves this event as a *DiscoEvent* object. Finally, it creates command that encloses the *DiscoEvent* with additional information, such as a time stamp in order to track when the event was generated, and the *username* of the groupware session in order to indicate the creator of the event. The *DiscoTechClient* saves the username when a user logs in the groupware session. The command is used by the *Networking Manager* component (discussed below) for message passing over the network. When the *DiscoTechClient* receives a command from the *Networking Manager*, it converts the command into a *DiscoEvent* and delivers the event to the *Groupware Application* via a callback. The *DiscoTechClient* tracks the sequence number of events, e.g., if a message is received, it increments a counter to keep a record of the number of received events. The sequence number is used to send an acknowledgment of received messages using a *Heart Beat* message (discussed below in Section 6.2.2.3).

6.2.1.2 DiscoTechServer

The *DiscoTechServer* mediates communication between different architectural components at the DiscoTech server node. If a client is declared as an application server (when using client/server distribution overlay, discussed in our disconnection-aware groupware model and below in Section 6.3), it unicasts a received message from an application client to the application server via the *Networking Manager* (discussed below). If it receives a message from the application server, it manages to broadcast them to all the clients via the *Networking Manager*. 
6.2.2 Infrastructural Components

There are four infrastructural components that take care of networking, reconnection, event-queuing, and disconnection identification tasks. Developers can reuse these components across different groupware applications. Four components are discussed as follows.

6.2.2.1 Networking Manager

The Networking Manager component is designed to communicate with remote nodes for sending and receiving events over the network. This component can be developed on top of an existing groupware development toolkit (such as GT [15]) that provides networking functionality or can be developed using the standard socket classes [91]. This component at the DiscoTech server side essentially functions as a message broadcaster as discussed in the model.

When the Networking Manager is implemented using the TCP socket classes, it will create a TCPLetener class using an IPEndPoint, a Local IP address and port number. A TCPLetener listens for and accepts incoming connection requests at the specified port. The Networking Manager at the sender or receiver will create a class using TCPClient to connect with the TCPLetener.

At the DiscoTech server, the Networking Manager has sender and receiver modules to send and receive events from each sender and receiver (discussed further in Section 7.1 and shown in Figure 7-1). It can also unicast a command to a particular receiver if needed (e.g., to the reconnected receiver).

When a receiver disconnects due to a network outage, these components (at the DiscoTech server and receiver) raise socket errors and pass the error to their corresponding Disconnection Identifier components (discussed below) via callbacks.
6.2.2.2 Disconnection Identifier

At the receiver, the Disconnection Identifier maintains its connection state by using a ConnectionState enumeration type variable, which can take three values: disconnected, connected, and reconnected. If a receiver becomes disconnected due to a network outage, the receiver side Disconnection Identifier sets its ConnectionState as disconnected. Before sending any message to the Networking Manger, all the components check this variable. If its value is disconnected, no message is sent in order to avoid a runtime socket error. At the DiscoTech server, the Disconnection Identifier keeps a list for the managing connection state of all the receivers.

The Disconnection Identifier at the receiver can also be used to identify a disconnection that happens due to network latency (discussed in Section 2.2.1). For example, in case of network latency, it sets the connection state based on a received time gap between a current message and the message received immediately before the current message. If the time gap is high (e.g., exceeds a predefined threshold), it sets the receiver’s connection state as disconnected. If the time gap comes within a lower range (e.g., below a predefined threshold), it sets the receiver’s connection state as reconnected and passes this status to the Reconnection Manager. The Reconnection Manager takes steps to display events quickly but not all at once. If the time gap comes to an allowable latency range depending on an application [1, 19, 45] the Disconnection Identifier sets the receiver’s connection state as connected. However, we did not simulate disconnections due to network latency by injecting artificial delays in the system.

The Disconnection Identifier component at the sender periodically sends Heart Beat messages to the DiscoTech server in order to send acknowledgements of received events (discussed further in Section 6.2.2.3).
6.2.2.3 Event Queue

On the DiscoTech server, the Event Queue maintains an event queue for each receiver and stores events in each of the event queues when an event arrives (further discussed in Section 7.1 and shown in Figure 7-1). If an event arrives, the Event Queue stores the event in the event queue for each receiver. For a connected receiver, the Event Queue purges events periodically based on a sequence number specified in a Heart Beat message received as an acknowledgment from the receiver. The receiver (taking the sender role) sends the Heart Beat message periodically mentioning the sequence number of the latest received message. The sequence number indicates that the receiver has received messages up to this number, so it is safe for the Event Queue to delete messages up to that number. If a receiver becomes disconnected (identified by Disconnection Identifier), the Event Queue does not purge messages stored for the receiver. Instead, it calls compactor plug-ins (discussed in Section 6.2.4.1) periodically in order to reduce the volume of the stored events for that receiver. Upon reconnection, the receiver side Event Queue stores the received reconnection information (events that are stored at the DiscoTech server during disconnection) and manipulate the events using a replayer plug-in (see below in Section 6.2.4.2).

6.2.2.4 Reconnection Manager

The Reconnection Manager takes necessary actions to send the stored events (via the Networking Manager) to a receiver as soon as it reconnects. When reconnection happens, the DiscoTech server side Reconnection Manager retrieves stored events from the Event Queue. The Reconnection Manager can combine them using any of the approaches discussed below:

- AP#1: It can aggregate all the stored events into a single message. If the size of the message becomes large, the Reconnection Manager compresses it using a standard compression
technique (such as Zlib [110]). However, if there are too many events, the compressed message size could be large, thus the message transmission time will be high. Moreover, the receiver will take time to decompress the large message. As a result, the receiver will not be able to display reconnection information as soon as it reconnects, rather it will take time to display them.

- **AP#2:** It can forward events individually. If there are many events in the event queue, the network will be flooded with small messages. These small messages will consume excessive bandwidth and cause delays in message transmission. Consequently, it will take time for the receiver to receive most recent information and the receiver will be able to present the most recent information after a long time. Moreover, there will be no provision to change the ordering of events.

- **AP#3:** It can combine a sequence of events using an aggregation method, but not using an aggregator plug-in because architecturally a plug-in is called by the Event Queue. Each combined events can be sent as a separate message. In this case, both the problems we mentioned in the above two approaches (AP#1 and AP#2) can be overcome. However, the receiver will not able to do any reordering of stored events.

We have used the first technique (AP#1, discussed above) in the DiscoTech toolkit as it allows for reordering (we did not, however, implement the compression technique). Then, the Reconnection Manager adds a special symbol to the aggregated message in order to indicate that this message contains reconnection information. Finally, it transforms the message into a command and passes the command to the Networking Manager that delivers the message to the reconnected receiver.
When a reconnection happens, the client side Reconnection Manager sets its reconnection flag, and parses the received combined event and retrieves all the individual stored events. Then, it adds the individual events in the event queue.

6.2.3 Summary of mediator and infrastructural components

In the plug-in architecture, we have specified four infrastructural components, such as the Event Queue, Reconnection Manager, Disconnection Identifier and Networking Manager that can be reused in different groupware applications. Two reusable mediator components are the DiscoTechServer and the DiscoTechClient that mediate interactions among the four infrastructural components. The DiscoTechClient also maintains interactions between the groupware application and the infrastructural components.

6.2.4 Plug-ins

The basic idea is that in the disconnection-handling architecture (Figure 6-2, which depicts an abstract view of the plug-in architecture), we have identified two important points as being the parameters of the architecture. With these two parameters, developers can customize the architecture for a specific application. The two important points are the choice of compaction strategies around the queue on the DiscoTech server side, and the choice of replaying strategies around the queue on the client side. Developers use compactor plug-ins (specified in Section 6.2.4.1) in order to instantiate the compaction strategies (covered by the first three dimensions in the disconnection design space, Figure 6-3).
Developers use replayer plug-ins (specified in Section 6.2.4.2) in order to instantiate the replaying strategies (covered by the last two dimensions in the disconnection design space, Figure 6-3). Therefore, by using only these two types of plug-ins, developers are able to instantiate all the strategies in the design space without changing other infrastructural components (specifically, the
Event Queue, the Reconnection Manager, the Disconnection Identifier and the Networking Manager, shown in Figure 6-1).

6.2.4.1 Compactor Plug-ins

Compactor plug-ins can have access to the queued events, and can perform whatever application-dependent algorithm they wish upon these events. We have defined three levels of compactor plug-ins based on the design space for manipulating groupware messages (discussed earlier). These three levels of plug-ins are implemented as objects descended from an abstract Compactor base class, which is shown in Figure 6-4.

Each compactor plug-in has two properties (Events and Timeline) and a method (Compact). With the Events property, the content of the event queue can be accessed and with the Timeline property, an application programmer can specify time range at which the compactor plug-in will be executed. The Compact () method is used to transform the content of the event queue.

Figure 6-4: Class diagram for the compactor plug-ins
An application programmer can use the built-in generic compactor plug-ins without any customization. However, for using the partially generic compactor, the programmer needs to customize a function of a concrete subclass, which is parameterized via the application specific event type. For writing a custom compactor plug-in, the programmer has to create a custom class (e.g., for the *ChatFrequencyTransformer* compactor) that will inherit the abstract base class *Compactor*. This custom compactor class will override the *Compact()* method. With this plug-in, a programmer can change the content of the event queue arbitrarily. All types of compactors are further discussed in Section 7.2.

6.2.4.2 Replayer Plug-ins

The replayer plug-ins manipulate stored events in the *Event Queue* using the strategies that reduce playback time of the events or using the strategies that present events in a different order than the original. The replayer plug-ins process events and pass them to the groupware application. The application then displays events to the reconnected user. As of the compactor plug-ins, we can classify replayer plug-ins into three categories: generic, partial generic, and custom. However, in order to simplicity, we have only implemented the generic replayer plug-ins that replay events quickly and that reverse the order of the events (discussed in Section 7.2).

6.2.5 Interactions among components during disconnection

As described in our model, during a disconnection of the receiver, the sender continues sending events and the DiscoTech server stores the events in the event queue. Using the UML sequence diagram notation, Figure 6-5 depicts the interactions between different components at the sender and the DiscoTech server during disconnection.
As discussed before, the Groupware Application sends a message to the DiscoTechClient, which processes the message and sends it to the DiscoTechServer via both the Networking Manager components. The DiscoTechServer mediator component changes the received message into a DiscoEvent and stores it in the Event Queue. The DiscoTech server periodically applies different compactor plug-ins in order to reduce the size of the event queue. In the sequence diagram, a compactor plug-in is called by the Event Queue every ten seconds interval using the plug-in’s Timeline property (discussed in Section 6.2.4.1).

6.2.6 Interactions among components upon reconnection.

When a receiver reconnects, the DiscoTech server side Reconnection Manager obtains events from the Event Queue.
The Reconnection Manager at the DiscoTech server aggregates the events as a single large event and adds a special symbol in order to distinguish as a reconnection event (discussed earlier, in Section 6.2.2.4). Then, it transforms the combined event into a command and forwards the command to Networking Manager, which then sends the command as a network message to the reconnected receiver over the network.

Upon reconnection, the receiver receives the message from the DiscoTech server. Figure 6-6 depicts the interactions among different components at the receiver upon reconnection. The Networking Manager of the receiver transforms the network message into a command and passes it to the DiscoTechClient that changes the command into a DiscoEvent and passes the event to the Reconnection Manager at the receiver.
The Reconnection Manager identifies the reconnection message and sets its reconnection flag to true until all parsed events are passed to Groupware Application. Therefore, if a sender sends events by this time, these are added in the event queue and treated in the same way as of the stored events. Event Queue calls a replayer plug-in. The plug-in can apply a speed-up strategy or can apply a strategy for re-ordering events, and finally, passes the events to Groupware Application using a callback.

### 6.3 Conceptual Overlay Architectures

An overlay architecture defines connections among participants’ nodes at a conceptual level. It is also known as a distribution architecture. According to Phillips [72], “distribution architectures describe the run time distribution of system state and computation across computing platforms connected by a network”. While there are several overlay architectures [58, 69, 72], Graham et al. [36] identified five architectures that are used in commercial groupware systems, such as Skype and MSN Messenger. These architectures follow the message passing computing model and are the variants of either client/server or peer-to-peer or hybrid (combined both client/server and peer-to-peer models) architectural models. The simple and widely used client/server model allows implementation of a groupware application based on a central program called the server that maintains the application state and keeps it consistent by communicating with clients at the level of window system events. In contrast, the scalable peer-to-peer model allows replica of the application and data model at each participant’s computer system and maintains the consistent state of replicated data model by allowing communication among application replicas directly.

With the designed plug-in architecture, it is possible to implement both the peer-to-peer and client/server distribution architectures.
We have already described this possibility in our disconnection-aware groupware model. As per our model, if we follow the peer-to-peer distribution architecture, the sender/receiver client represents the full groupware application with the capacity to store and replay events for handling disconnection. As Figure 6-7 shows, if a sender sends a message, it can receive the same message via the DiscoTech server without any change.

Conversely, if we follow the client/server distribution architecture, a sender client (which is a receiver also) can be declared as an application server that is able to do different tasks, e.g., in the Tank game, a game server takes decision about a hit or a miss. The other sender and receiver clients take the role of the groupware clients. These clients are responsible for doing different tasks, e.g., in the Tank game, the game client processes inputs from the players and renders output. Figure 6-8
depicts how different components can interact in this overlay architecture. For example, if an Application Client sends a message, it can receive the same message or a different message via the Application Server. The DiscoTechServer forwards the event to the Application Server, which can modify the message and send a new message to the client via the DiscoTechServer. Both the Application Client and the Application Server will have the capability to store and replay the events as per our model.

**Figure 6-8: Interactions among components in client/server overlay**
6.4 Quality Goals for the Plug-in Architecture

In the IEEE Glossary of Software System Engineering Terminology [77], quality is defined as “the degree to which a system, a component, or a process meets customer or user needs or expectations”. The quality of the software is measured against the degree to which user requirements (such as correctness, reliability and usability) are met. The factors that affect quality are termed as quality attributes. Some relevant quality attributes are usability, performance, maintainability and reusability [95].

Software architecture plays a key role for determining whether a system’s quality requirements can be satisfied [4, 18, 75, 70, 77, 87]. Bass et al. [4] mentioned, “in software intensive systems, software architecture provides a powerful means to achieve the system qualities over the software life cycle”. In the following, we discuss how the plug-in architecture can achieve our quality goals while handling disconnection. Guided by the design space (described in Chapter 4), in this thesis, we designed the plug-in architecture with the following key quality attributes:

- **Ease of Use.** Groupware developers should be able to use the architecture without having to substantially modify it or their applications. The architecture should be available as a usable toolkit with a simple API.

- **Expressiveness.** The architecture should support a wide range of disconnection and reconnection behaviours, as captured in the five-dimensional design space and as discussed for five different groupware applications (see Section 3.1).

- **High Performance.** Applications using the architecture should not pay a major performance penalty, and any penalty should be predictable and quantifiable.
The plug-in architecture can help developers to achieve the *ease of use* quality goal. With this architecture, developers do not have to re-implement the infrastructural components for handling disconnections, whereas they only need to implement appropriate plug-ins. When developers create a new application, they might choose to use different strategies for handling disconnection. Then, they can reuse all the infrastructural components except the plug-ins. However, building plug-ins can be difficult in some cases because developers might need to deal with a complex data structure (e.g., heat map). In this case, following this architecture, a toolkit can be developed with a library of plug-ins. Using the DiscoTech toolkit and its plug-ins, developers should be able to implement disconnection handling in their applications in a simple and flexible manner (further discussed in Section 7.1 and a user study results are shown in Section 8.1).

There are three levels of plug-ins (generic, partial-generic and custom, discussed earlier), which cover the whole design space for handling disconnection. The three levels of plug-ins should help us achieve the *expressiveness* quality goal (further discussed in Sections 7.2 and 8.2).

This architecture is responsible for event handling (such as saving events during disconnection), compressing events in order to optimize the storage and bandwidth requirements, and passing events to the groupware application for replaying. The architecture separates disconnection-handling tasks from application specific tasks. Therefore, it can support disconnection handling without interrupting the application’s flow of operations (e.g., the application never needs to wait while the *Event Queue* component stores events or the *Compactor* plug-ins compact events in the event queue) and adds minimal overhead to the groupware application’s performance (shown in Section 8.3) Thus, by using this architecture, developers can achieve the *high performance* quality goal, further discussed in Section 7.3.
6.5 Summary and Discussion of Plug-in Architecture

The plug-in architecture offers codified solutions to address developers’ problems while implementing disconnection in their groupware applications. The goal of the plug-in architecture is to make disconnection handling simple and flexible for groupware developers. It offers developers the option to reuse, modify and compose plug-in solutions for handling disconnection without modifying other infrastructural components. In Sections 7.1, 7.2, and 7.3, we discuss how this architecture is capable of achieving the three quality goals: ease of use, expressiveness, and high performance. In Chapter 8, we provide empirical evidence that the plug-in architecture can indeed achieve these goals in practice.
Chapter 7

Designing DiscoTech Toolkit

In order to justify the practicality of the three forms of codified solutions described in Chapters 4, 5 and 6, we developed a toolkit, called DiscoTech, and used it to construct eight disconnection-aware groupware applications. The toolkit uses the plug-in architecture (described in Chapter 6) and the disconnection design combos (described in Chapter 5). The plug-in based toolkit allows developers to implement the wide varieties of strategies captured by the design space by reusing code (described in Section 4.2). The concepts of disconnection design combos adapted in the toolkit allow developers to execute multiple strategies at a disconnection timeframe based on the age of events. The overall goal of DiscoTech is to simplify groupware developers’ tasks for addressing different disconnection scenarios and timeframes.

In this chapter, we first present the DiscoTech toolkit. After that, we describe how the toolkit can achieve the three quality attributes offered by the designed plug-in architecture. Then, we explain the implementation details of DiscoTech. Finally, we show how DiscoTech can implement disconnection handling in five different kinds of groupware applications.

As mentioned earlier, DiscoTech uses the plug-in architecture comprising of several components. We specify functionalities of the components based on our disconnection-aware groupware model (discussed in Sections 4.1 and 6.2). Following the model and the plug-in architecture, DiscoTech offers a server program (which is a dynamic link library), called ‘DiscoTech Server’ residing at the DiscoTech sever node. The DiscoTech sever is a message broadcaster that receives messages from the sender nodes and broadcasts the messages to the receiver nodes. Both at the receiver and sender
nodes, DiscoTech has copies of a ‘DiscoTech client’ program (which is a dynamic link library) that sends and receives messages via the DiscoTech server.

In order to handle disconnection, the DiscoTech server (shown in Figure 7-1) has event queues for all the clients and each client (shown in Figure 7-2) has its own queue. The DiscoTech server has the capability to store and forward events and to compact events. The clients are instances of a full groupware application (in case of peer-to-peer overlay architecture, discussed in Section 4.1) or partial instances of a groupware application (in case of client/server overlay, discussed in Section 4.1). The clients can store and replay events. Both the architectural diagrams (Figure 7-1 and Figure 7-2) depict a partial of the plug-in architecture specified in-depth in Section 6.2 and shown in Figure 6.1. In these architectural diagrams, the *Sender* and *Receiver* modules are the classes of the *Networking Manager* component (discussed in Section 6.2.2.1). In order to focus on the relevant components and to simplify our discussion, we used these abstract and the logical architectural views [54, 87].

![Figure 7-1: DiscoTech server](image)

136
As discussed in Chapter 6, the designed plug-in architecture can achieve three quality goals: *ease of use*, *high performance* and *expressiveness*. We realize these goals by implementing the DiscoTech toolkit following the plug-in architecture. Groupware developers can use DiscoTech as a networking API that provides built-in disconnection handling. Developers stream their application events via the DiscoTech API, and events are propagated via the network to other clients. In case of disconnections, events are queued on a DiscoTech server, and sent to the disconnected client upon reconnection. Developers can easily select among available strategies in the DiscoTech API for dealing with disconnection, or can develop custom plug-ins.

7.1 Ease of Use through the DiscoTech Toolkit

DiscoTech allows groupware developers to easily add disconnection handling to their application. Developers simply route their application’s events through the toolkit. Similarly, to other middleware solutions, DiscoTech broadcasts the application’s events to all other clients. DiscoTech handles disconnection and reconnection events by constantly queuing application events in case of disconnection, and by automatically providing catch-up functionality when the client reconnects.
Figure 7-2 shows how applications interact with DiscoTech. The application sends events to DiscoTech via its API. DiscoTech then propagates the events to other clients via its Sender module. Events received from other clients are queued on an event queue, and are forwarded to the application via a call-back. DiscoTech therefore propagates events for applications, providing the network functions usually provided by a middleware such as GT [15] or by raw socket communication. DiscoTech has no knowledge of event semantics; events are simply viewed as serializable objects that can be queued or sent over a network.

DiscoTech provides a server (Figure 7-1) whose primary function is to broadcast messages from each client to all other clients. Incoming events for client $i$ are read by a Receiver process, which then enqueues the events on each client’s Event Queue. A Sender process for each client waits for new messages to arrive, and sends them to the client.

If a client disconnects, its sender/receiver pair attempts to reestablish the connection. In case of a brief network disruption, the connection might be re-established immediately. In a longer disconnection (e.g., a computer being powered down for the weekend), the connection may take hours or days to reestablish. In the meantime, events are queued so that they can be sent to the client upon reconnection. During longer disconnections, the contents of the event queue must be modified in order to reduce space requirements and to avoid the need for lengthy data transmission following reconnection. The groupware developer can configure the techniques used to reduce the space requirements of the DiscoTech server-side event queue. The data stored on the event queue is reduced through the approaches discussed in the design space (e.g., compression, aggregation, and sampling).
Following reconnection and receipt of data, the client controls how the data are sent to the application. Events can simply be forwarded to the application directly, or can be modified to help the user re-establish context. For example, the client’s plug-ins may reorder events so that highest priority events are processed first, or may adjust playback to show missed activities at double speed.

7.2 Expressiveness through Plug-ins

Application programmers can use the DiscoTech toolkit as-is, selecting among a set of predefined compaction and replaying behaviours. This allows easy addition of disconnection handling – all that the developer needs to do is to route the application’s events through DiscoTech.

This approach limits disconnection handling, however, to behaviours that can be programmed without knowledge of the underlying semantics of the events. For example, the following generic behaviours are possible with the toolkit:

- Compression – compress all events using standard libraries such as Zlib [110]
- Truncation – discard the last k events (or k seconds of events)
- Sampling – discard all but a subset of events
- Accelerated playback of events

However, many interesting behaviours require knowledge of the semantics of the events themselves, such as:

- Aggregation of multiple events to a single event conveying the same information
- Discarding events which are superseded by later events
Transforming events into a summary representation

DiscoTech uses a plug-in mechanism to open the toolkit, allowing programming of custom disconnection behaviours. Plug-ins can be attached to either the DiscoTech server or client-side event queues. Plug-ins are periodically invoked by the toolkit (as triggered by timer interrupts or by the queue reaching a threshold size), and may arbitrarily modify the contents of the queue. Plug-ins fall into three categories: generic, partially generic and custom (see Section 6.1).

- **Generic Plug-ins**: Generic plug-ins, provide functionality that can be implemented with no knowledge of events’ semantics. Events are considered to be generic objects, tagged with a time stamp specifying when the event originated at the sender client. The toolkit provides a set of generic plug-ins, providing simple and useful disconnection behaviour. Generic plug-ins are parameterized by the type of message on which they should operate (e.g., awareness messages only), and by the timeframe in which they should operate.

- **Partially Generic Plug-ins**: The toolkit provides plug-ins that must be parameterized by the application programmer to adapt them to the event types used by the application. For example, a Chunker plug-in combines groups of events into coarser-grained events. This plug-in could transform individual keystroke events from a chat system into a single event, or could coalesce line-segment events into a poly-line for a drawing program. While the aggregation behaviour is generic, the specifics of how a set of events are transformed into a single event are application-specific. Users of this plug-in must parameterize it by providing the implementation of a Chunk method that takes a list of events and combines them into a single event. In DiscoTech, this plug-in is implemented as an abstract class,
requiring the application programmer to create a concrete subclass that implements the *Chunk* function.

As a second example, the *AggregateToState* partially-generic plug-in allows a sequence of events to be collapsed into a single state update event, possibly involving a significant change in form. As we shall see, *AggregateToState* can be used to convert a sequence of line drawing commands to a single bitmap image, or a sequence of telepointer move events to a matrix representing a heatmap. This plug-in must be customized with the underlying data structure (e.g., the bitmap) and with a method that applies events to the data structure (e.g., drawing the line on the bitmap).

Partially generic plug-ins significantly extend the expressiveness of the toolkit. The cost to the application programmer is that she must understand the functionality of the plug-ins well enough to for customizing it as per the application’s event types.

- **Custom Plug-ins**: Some functionality is so dependent on the underlying event types that no generic behaviour is applicable. For example, a sequence of moves in a chess game might be summarized using standard chess notation, or a sequence of video frames might be converted through image analysis to a summary of the video’s content.

These three types of plug-in represent a progression from the simplicity of full genericity to the power of custom components. This allows developers to choose the degree to which they wish to engage with the toolkit – if standard behaviours are acceptable, there is little for developers to learn. If they wish to have sophisticated, custom behaviours, the toolkit makes it possible.
7.3 High Performance through Light-Weight Operation

While disconnection handling can greatly improve the usability of groupware applications, it must not unreasonably affect performance of the application. DiscoTech is designed to provide minimal overhead during normal connected state, and to provide developers with hooks to manage resource consumption during disconnected states.

Application events sent from one client to another pass through the DiscoTech server, which broadcasts them to other clients. This message broadcasting architecture is widely used in groupware implementation (e.g., GT [15], or many commercial games). DiscoTech adds the overhead of queuing events on the DiscoTech server rather than sending them directly. When a client is connected, however, events can be removed from the queue as soon as they are added. As shown below, the overhead of the queue is minimal compared to the cost of sending messages over the network.

When a client is disconnected, the DiscoTech server enqueues its messages until the client reconnects. This can lead to unbounded memory use, as there is no limit to the length of disconnection. Server-side plug-ins allow the developer to reduce queue size with a variety of pruning and compaction techniques. Through parameters on these plug-ins, the developer can control how aggressively these algorithms should applied – e.g., how often they should be invoked and how much data should be retained.

Following reconnection, the queued data is sent to the reconnected client. As the event queue grows, so does the bandwidth required to transmit these events. Bandwidth requirements are controlled through two mechanisms. First, as described above, developers can control the amount of data
stored on the event queue. Second, the network module can minimize transmission time by delivering the queue as a whole, and by using compression to reduce size.

Later in this thesis, we present evaluation results showing that DiscoTech indeed imposes minimal overhead during normal connected operation, and provides hooks for control over resource requirements during disconnection.

7.4 Details of Plug-in Implementation

The primary novelty of DiscoTech is its plug-in approach to handling disconnection and reconnection. Plug-ins are classes that descend from the abstract base class DTPlugin, and are all designed around a similar pattern – plug-ins have access to the queued events, and can perform manipulations on that queue depending on the algorithms built into the plug-ins (discussed in Section 6.2.4, and the class diagram for the different types of plug-ins are shown in Figure 6-3). We implemented plug-ins to correspond to all of the types detailed in the design space, including reformatters, resamplers, re-orderers, re-pacers, and transformers (see in Section 4.2).

A plug-in’s ‘main’ method is called ApplyPlugin. When invoked by the toolkit, this method applies the plug-in’s algorithms to the event queue. Plug-ins have two properties: EventQueue, which links to the event queue that is to be manipulated; and Timeframe, which specifies the time period for which the plug-in is to operate.

In addition, certain plug-ins have additional properties (e.g., the Truncator plug-in takes an argument on creation to determine the age at which events will be deleted from the queue). Last, all plug-ins have a method that carries out the desired actions on the given queue (this method is specific to the plug-in type).
Table 7-1: Summary of the developed plug-ins

<table>
<thead>
<tr>
<th>Plug-ins</th>
<th>Description of the plug-ins</th>
<th>Addressed dimension in the design space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>plug-ins</td>
<td></td>
</tr>
<tr>
<td>Truncator</td>
<td>Truncates events from the event queue based on type or age of events.</td>
<td>Sampling</td>
</tr>
<tr>
<td>Speed-up</td>
<td>Compresses timelines of events either uniformly or non-uniformly.</td>
<td>Pacing</td>
</tr>
<tr>
<td>ReverseOrderer</td>
<td>Changes the order of the events in a reverse manner.</td>
<td>Ordering</td>
</tr>
<tr>
<td>Partially Generic</td>
<td>plug-ins</td>
<td></td>
</tr>
<tr>
<td>Chunker</td>
<td>Combines a set of events into a single event.</td>
<td>Message Formatting</td>
</tr>
<tr>
<td>Averager</td>
<td>Replaces a set of events representing visual embodiments in a shared workspace (e.g., telpointer) in each one-second interval by a single event whose position and timestamp are average of the events it is replacing</td>
<td>Transforming</td>
</tr>
<tr>
<td>AggregateToState*</td>
<td>Allows a sequence of events to be collapsed into a single state update event, possibly involving a substantial change in representations</td>
<td>Message Formatting</td>
</tr>
<tr>
<td>BitmapTransformer</td>
<td>Converts a sequence of drawing commands into a single bitmap image. This is an AggregateToState* plug-in</td>
<td>Message Formatting</td>
</tr>
<tr>
<td>DiagramState Transformer</td>
<td>Transforms a sequence of editing events into a single diagram state. This is an AggregateToState* plug-in</td>
<td>Message Formatting</td>
</tr>
<tr>
<td>Heatmap Transformer</td>
<td>Transforms telepointer events into a heatmap data structure. This is an AggregateToState* plug-in</td>
<td>Transforming</td>
</tr>
<tr>
<td>Custom</td>
<td>plug-ins</td>
<td></td>
</tr>
<tr>
<td>CommentSummaryGenerator</td>
<td>Transforms comment typing events into a comment summary event.</td>
<td>Transforming</td>
</tr>
<tr>
<td>ChatFrequencyTransformer</td>
<td>Counts the number of chat messages from each user.</td>
<td>Transforming</td>
</tr>
<tr>
<td>HistoricalTrace Transformer</td>
<td>Transforms move events of a visual embodiment (e.g., a tank in the Tank game,) of workspace into a ‘HistoricalTrace’ data structure.</td>
<td>Transforming</td>
</tr>
</tbody>
</table>

*It is a common name for different plug-ins

Application programmers can use the built-in generic plug-ins (e.g., the Truncator) without any customization. For the partially generic plug-ins, such as the Chunker plug-in (see above), the programmer needs to redefine the behaviour of the Chunk() method.
For a custom plug-in, the programmer creates a new class that inherits the abstract base class `DTPlugin`. With custom classes, programmers can arbitrarily change the content of the event queue. Multiple plug-ins can be applied to the event queue (e.g., to provide different treatment for old versus new events).

Each plug-in must therefore be capable of recognizing event types generated by other plug-ins. For example, if chunking introduces a new event type into the event queue to represent the newly ‘chunked’ data, other plug-ins must be capable of dealing with that event type. The developed plug-ins are summarized in Table 7-1.

7.5 Using the DiscoTech Toolkit

DiscoTech is implemented in C#, and can be used in any .NET project by including DiscoTech client project or DLL at the client application and the DiscoTech server project or DLL at the server application (further explained in Appendix A). There is a clean separation between the application and DiscoTech’s components; the application only interacts with the toolkit’s `DiscoTech API` component (which acts as a mediator between the application and DiscoTech). At the simplest level of use (i.e., generic plug-ins only), developers only need to learn the DiscoTech APIs in order to pass events through the DiscoTech instance; disconnection and reconnection behaviours are handled transparently. Programmers instantiate DiscoTech with details of the server IP address and port, and then define two event handlers in order to receive commands and data messages.

The programmer also needs to specify the set of plug-ins they want to use: e.g., to use the `HeatmapTransformer` plug-in between 60 seconds and ten minutes of disconnection, the following lines are used:
DTPlugin hm = new HeatmapTransformer(60,600);

DiscoTech.Instance.AddPlugin(hm);

To send a message over the network, the programmer uses the following call:

DiscoTech.Instance.SendMessage(message);

where ‘message’ is a string or an event object. The receiver will get the message via the event handler mentioned above.

The guidelines of how developers can use the DiscoTech source code and its API are explained in Appendix A, and can also be found at the DiscoTech web page [20].

7.6 Supporting Existing Disconnection Techniques Using DiscoTech

DiscoTech can support a wide variety of disconnection techniques seen in previous work. These techniques are discussed in Section 2.3. In the following list, we show how several earlier approaches, including latecomer support, are accomplished using DiscoTech.

- **Maintain model state** [8, 79]. We provide the partially-generic ‘aggregate-to-state’ plug-in, which can convert model updates to a single state representation. Although not a purely state-based approach, this plug-in allows developers to come arbitrarily close to this behaviour.

- **Store-and-forward** [11, 12]. DiscoTech provides this policy natively through its queues, although not all updates will be forwarded (i.e., if pruned by another plug-in).
- **Event replay** [59, 60]. This is possible with the generic ‘Speed-up’ plug-in. One earlier system provided a ‘VCR-style’ interface for replay [10]; we do not supply this specific interface, but the *Speed-up* plug-in could easily support a replayer with ‘pause’ and ‘fast-forward’ controls.

- **Selection filtering** [43, 57]. DiscoTech supports filtering with generic plug-ins (which are parameterized by message type and age of events); more complex filtering (e.g., by message content [2], and the “freshest data” policy [21]) can be achieved through custom plug-ins.

- **Reordering** [7]. This policy is part of the DiscoTech design framework, but must be implemented as a custom plug

- **Coalescing and chunking** [2]. We handle these strategies through partially-generic plug-ins such as the *Chunker* described above.

- **Working while disconnected** [50, 88]. We did not implement this approach using DiscoTech, considering it as potential future work (discussed in Section 9.4.2).

- **Latecomer support** [11]. Support for latecomers is built into DiscoTech; we implement this by creating an extra client at system startup that is considered to be always disconnected. When a late entrant joins, we copy the extra client’s queue to the new entrant and send it to them. Note that whatever policies are in place for disconnected clients will also affect the late entrant – e.g., they may receive a summary representation of chat rather than all of the text. This is different from schemes that guarantee the sending of a full model, but our approach provides more flexibility for the application programmer – we believe that what
is sent to a late entrant should be an application-level decision, which can be achieved with DiscoTech.

7.6.1 Summary of supporting existing techniques with DiscoTech

DiscoTech is capable of addressing different exiting techniques for handling disconnection in synchronous groupware, including *Maintain model state*, *Store-and-forward*, *Event replay*, *Selection filtering*, *Reordering*, *Coalescing*, and *chunking*. DiscoTech supports these techniques using its plug-ins. DiscoTech also supports latecomer for its inherent plug-in based architectural characteristics. The capability of supporting these techniques shows its expressiveness to support a range of disconnection and reconnection behaviours. However, the current implementation of the DiscoTech does not support the technique ‘*working while disconnected*’. We consider this as a future work discussed in Section 9.4.

7.7 Supporting Disconnection Design Combos with DiscoTech

In chapter 5, we discussed that developers need to manage the trade-off between the performance and understandability requirements if disconnection lasts for a longer time. We have formulated three disconnection design combos (such as *fidelity-reducer*, *accumulator* and *compacted-pacer*) to address the trade-off in three disconnection situations. These combos suggest that it is possible to manage the trade-off by applying multiple strategies to the event queue at different timeframes based on the age of events. We implemented the concepts of the three disconnection design combos in DiscoTech for supporting different strategies at different timeframes. However, we did not offer any specific API in DiscoTech (e.g., *CreateFidelityReducerCombo*) to directly use the combos.
While applying multiple strategies to the different portions of the event queue, we identify two major implementation issues (discussed in Section 5.4.3). In the following, we discuss the issues and explain how we have addressed the issues in DiscoTech:

**Issue 1:** A strategy applied at a particular timeframe generates new events by manipulating events in the event queue, and removes the original events when the manipulation is done. As a result, a strategy applied at another disconnection timeframe cannot manipulate the original events. For example, in the paper reviewing application, when an averaging strategy is applied on telepointer gesture events at the shorter timeframe (1 min – 10 min), this strategy generates new events that contain average positions of sequences of original gesture events, which are removed after the manipulation. As a result, the heatmap strategy applied at the longer timeframe (10 min – 1 hour) has no longer access to the original telepointer events.

In order to tackle Issue 1 in the DiscoTech toolkit, we flag an event after applying a strategy rather deleting it from the event queue. A *strategy list* is added as a property in the *event* data structure. The *strategy list* contains all the strategies that have already been applied to an event in the event queue. If an event is chosen by any strategy, the strategy name is added in the *strategy list* of the event. Having this *strategy list*, it is possible to apply a strategy periodically in the unprocessed events of the event queue while at the same time keeping all processed events in the event queue. That means that the applied strategy does not consider the processed events for further manipulation. For this, we write two lines of code in each of the plug-ins implementation as shown by the code fragment in Figure 7-3.
if(eventList[i].Message.Contains(MessageTag) && eventList[i].Contains(StrategyName))
{
    eventList[i].StrategyList.Add(StrategyName);
    ...
}

Figure 7-3: A code fragment for using ‘MessageTag’ and ‘StrategyList’ properties

//Instantiating the plug-in at the timeframe 10 min - 1 hour
DTPlugin pl = new HeatmapTransformer(60,3600);
pl.StrategyName = CompactionStrategies.Summarization;
pl.MessageTag = "TelepointerGesture";
DiscoTech.Instance.AddPlugin(pl);

Figure 7-4: A code fragment to use the 'HeatmapTransformer' plug-in

If developers create a custom plug-in, they need to write these two lines of code. During the instantiation of a plug-in (that implements a strategy), developers need to mention the strategy name. For example, Figure 7-4 shows a code fragment to assign the strategy name when instantiating the HeatmapTransformer plug-in. In the figure, the MessageTag property of the plug-in allows a developer to specify the type of events in the event queue that the plug-in will manipulate. For example, the HeatmapTransformer plug-in will manipulate the telepointer gesture events.

However, a huge number of events cannot be kept in the event queue for a long period of time because of their higher memory space requirements. In addition, upon reconnection, only the required events are needed to be delivered, but if all redundant events remain in the event queue, it would take high processing time to deliver the required events by discarding the redundant ones. Therefore, in order to get rid of the redundant events, the system needs to periodically clean up unnecessary events based on heuristics. For example, once one cycle of summarization process is
done, older original events are no longer required, and thus the system can get rid of those events based on their ages.

**Issue 2:** As explained in Issue 1, for some types of events (e.g., telepointer gesture), the strategy used at the longer disconnection timeframe might need to use the original set of data, instead of using manipulated events created by a strategy at the shorter timeframe. However, for other types of events, it might prefer to use the manipulated events generated by a strategy applied at the shorter timeframe, especially to reduce the computation overhead. For example, in the paper reviewing application, at the shorter disconnection timeframe (1 min – 60 min), developers might apply the *chunking* strategy to combine the character-by-character typed events. This strategy will create new events called *ChunkedComment*. At the longer timeframe (60 min- 3 hours), developers might use a summarization strategy to create the comment summary (discussed in Section 3.1.2.2). However, unlike the case of Issue 1, developers might prefer to use chunked comments for making the comment summary instead of using the original character-by-character typing events. In order to resolve Issue 2 in the DiscoTech toolkit, we have used the *MessageTag* property of a plug-in (discussed above in Issue 1). The *MessageTag* property allows developers to specify the kind of events the summarization strategy will manipulate.

```java
//Instantiating the plug-in at the timeframe 60 min - 3 hours
DTPlugin pl = new CommentSummaryGenerator(3600, 10800);
pl.StrategyName = CompactionStrategies.Summarization;
pl.MessageTag = "ChunkedComment";
DiscoTech.Instance.AddPlugin(pl);
```

*Figure 7.5: A Code fragment showing how to parameterize a summarization plug-in*
Figure 7-5 shows that using the MessageTag property developers can specify that the CommentSummaryGenerator plug-in will manipulate the chunked comment events.

7.8 Summary of Designing and Implementing DiscoTech Toolkit

Handling user-level disconnection and reconnection issues in synchronous groupware is difficult for developers. In order to simplify this support, we have developed DiscoTech, a toolkit that covers a wider range of policies than any previous infrastructure but that is easy to understand and imposes only minimal performance overhead (shown in Chapter 8). DiscoTech’s main novelty is that it addresses disconnection problems with a plug-in architecture, which provides both simple use in basic cases with the generic plug-ins, as well as options to handle increasingly complex issues with the partially generic and custom plug-ins. DiscoTech is based on a new representation of a five-dimensional disconnection design space (discussed in Chapter 4) that characterizes the types of behaviours that applications might need to exhibit as a consequence of disconnection. Using DiscoTech, it is possible to support the existing techniques for handling disconnection (discussed in Section 7.6). Using the DiscoTech toolkit, we have implemented the concepts of disconnection design combos (discussed in Section 7.7) while addressing different implementation issues.

7.9 Developing Disconnection-aware Groupware using DiscoTech

We have developed substantially different disconnection-aware groupware applications using the plug-in based DiscoTech toolkit. Developing these applications helped us to determine the range of plug-ins required to implement wide varieties of disconnection and reconnection behaviours. Therefore, this phase helps us answer our fifth research question (RQ#5: can a small set of solutions cover all sorts of disconnection scenarios or do developers need a wide range of solutions?"
discussed in Section 1.3). In this way, we have also shown the expressiveness of DiscoTech. In addition, the different varieties of applications will work as example applications for developers.

In the developed eight groupware applications, we adapted different disconnection and reconnection behaviours (e.g., replaying events at double speed and summarizing telepointer gesture events as an activity map or heat map) by developing eleven plug-ins (summarized in Table 7-1). In these applications, different infrastructural components are reused that take care of disconnection handling, queuing, networking and reconnection. In the following, we summarize the reasons for selecting each of the applications:

1. A text chat application demonstrating how to handle disconnection in case of discrete character-by-character typed events. This application is also chosen for performance analysis.

2. Shared workspace editors demonstrating how to handle disconnection in a shared workspace where users can see and manipulate artifacts related to their activities. Four shared editors are developed to explore disconnection handling in different workspace environment.

   a. A collaborative paper reviewing application demonstrating how disconnection handling can be implemented in the application in which streams of events are generated with the command of free hand gesturing using telepointers.

   b. A shared UML use case editor demonstrating how disconnection handling can be implemented in a shared drawing and editing application in which UML diagram elements can be dragged and moved.
c. A shared drawing editor demonstrating how disconnection handling can be implemented in the application in which streams of events are generated with the command of free hand sketching using telepointers. This application is also developed for the user study and performance testing.

d. A shared coding editor demonstrating how disconnection handling can be implemented in the application in which a group of programmers can copy and paste code collaboratively.

3. Tank game demonstrating how disconnection handling can be implemented in the genre of collaborative gaming in which a game server is required.

4. A remote presentation tool demonstrating how disconnection can be handled where users can share their workspace and pointing data with remote collaborators. This application allows us to study the modality fusion requirement (discussed in Section 3.1.5.1).

5. A gesture-based collaborative game that shows how disconnection can be handled in a slow-paced game.

The applications show a range of strategies for handling disconnection, and show that DiscoTech supports both low-cost generic behaviours and as well as complex and application-specific behaviours when required. Therefore, these applications lead us to implement eleven plug-ins. These plug-ins lead us to determine the answer to the fifth research question (RQ#5, Section 1.3) that with a reasonable number of plug-ins, it is possible to implement disconnection handling for a wide variety of user actions (discussed below).
7.9.1 Text chat application

In this simple chat system, users type character-by-character chat messages for conversation (discussed in Section 3.1.1). Different plug-ins are used for implementing different disconnection and reconnection behaviours. When a client is disconnected, messages are queued on the DiscoTech server; for long disconnections, messages are transformed to take up less space according to the following rules:

- Awareness events older than ten seconds are removed from the queue using the generic \textit{Truncator} plug-in.

- Chat messages older than ten minutes are replaced with a ‘chat frequency’ event that counts the number of chat messages from each user. This uses a custom plug-in requiring knowledge of the application’s event types.

On reconnection, messages are replayed at the client according to the following rules:

- ‘Chat frequency’ events are passed to the client, which highlights users’ icons based on the number of messages (as of Figure 3-3) using the \textit{ChatFrequencyTransformer} Plug-in.

- Newer chat messages are replayed at double speed using the generic \textit{Speed-up} plug-in until the playback catches up. The order and timing of the messages match their original input, but the timeline is compressed.

Despite its simplicity, this example illustrates much of the power of the DiscoTech architecture. It is easy to configure the DiscoTech server to limit the amount of information that is stored, an approach that allows a client to be disconnected indefinitely. A custom plug-in is used to create the
chat frequency event, but if developers prefer not to create this plug-in, the generic truncation plug-in could be used to discard old chat messages. This shows that developers can choose between simple behaviours with generic plug-ins, or extended behaviours with custom plug-ins.

The reconnection plug-ins are used to help users regain context. Here, highlighting user icons gives a general picture of who were active during the disconnection. Older chat messages are simply posted to the window, while newer messages are played back at a relative rate they were originally typed.

The plug-ins used in this application are parameterized by thresholds such as the age at which awareness messages should be discarded, or the age at which double-speed playback should be used on the client. These parameters are simple properties that are set through the DiscoTech API.

7.9.2 Shared UML use case diagram editor

Our second application allows a small group of people to edit and discuss UML use case diagrams (discussed in 3.1.2.3). As with the chat example, when a client disconnects, the DiscoTech server queues pending events until reconnection. Although the use-case editor is substantially different from the chat application, similar disconnection-handling strategies can be employed for different types of events, explained as follows:

During disconnection, Telepointer and intermediate-drag events older than 60 seconds are removed from the queue using the generic Truncator plug-in, where events between 10-60 seconds old are averaged using a partially generic Averager plug-in (described in Table 7-1), which is customized with code to deal with averaging of telepointer positions. If the disconnection lasts more than an hour, older editing operations are transformed into a single ‘diagram state’ event that specifies the
positions and attributes of all diagram elements. This is performed using the partially generic
DiagramStateTransformer plug-in (which is an AggregateToState plug-in, Table 7-1), customized
with knowledge of how to apply a single edit event to a representation of the diagram’s state.

Following reconnection, the telepointer and drag events are played back at double speed, and all
other events are simply passed to the application for immediate execution.

This example shows the reusability of standard plug-ins. Although the use case editor is
substantially different, from the chat application, it uses the same Truncator and Speed-up plug-
ins. It also introduces two more partially generic plug-ins, Averager and AggregateToState, which
will also be used in the next examples.

7.9.2.1 Collaborative paper reviewing application

Our third application is collaborative paper reviewing application (discussed in Section 3.1.2.3).
Again, although this application is substantially different from the previous two, the required plug-
ins are similar. All editing operations must be preserved on the DiscoTech server’s event queue. If
the disconnection is lengthy (more than ten minutes), the partially generic AggregatoState plug-in
is used to combine all edit events into a single event that conveys the current state of the
annotations. For handling the telepointer events in different disconnection timeframes, we have
applied Speed-up, Averager and HeatmapTransformer (which is a custom plug-in, see Table 7-1)
plug-ins.

7.9.2.2 Shared drawing editor

This is a canonical drawing program allowing users to perform free hand sketching with
telepointers (discussed in Section 3.1.2.1). This application is also developed to do the user study
(Section 8.1) and performance testing (Section 8.3), in order to show that DiscoTech offers an ease of use API and imposes minimal overhead in terms on application’s performance.

Different application behaviours related to disconnecting handling are adapted in the application by reusing plug-ins, for example, older drawing events are applied to a special bitmap message using the BitmapTransformer plug-in (an AggregateToState compactor, see Table 7-1); telepointer events between one and ten minutes old are averaged to provide at most one event per five seconds using an Averager compactor; Telepointer events older than ten minutes are converted to a heatmap the using the HeatmapTransformer plug-in.

7.9.2.3 Shared coding editor

This is a canonical coding editor for PHP programming (discussed in Section 3.1.2.4). This editor allows a programmer to copy and paste source code in real time. This feature makes it different from the other three shared workspace applications above. However, similar set of plug-ins are also reused for these two user actions, e.g., the Speed-up plug-in is reused to replay newer copy and paste commands, and the AggregateToState plug-in is reused to combine all the copy and paste commands into a single event.

7.9.3 Tank game

With this game a group of players can play game in online by moving their tanks (discussed in Section 3.1.3). This application is different from other applications as it follows client/server overlay and has a game server (discussed in Section 4.1).
In order to check how DiscoTech’s architecture can adapt a game server, we have developed this application. Following the model of the disconnection-aware groupware, DiscoTech implements the option of having a game server (discussed in Section 4.1). The message communication structure between the DiscoTech server and the Tank game server during a normal connected situation discussed as follows (shown in Figure 7-5).

A player sends events via its DiscoTech’s client (DC) to the DiscoTech Server (DS). It then passes the events to the Game Server (GS). The GS updates the game state and broadcasts the events to other clients via DS. GS also manipulates the events if necessary, e.g. for the fire event, it calculates whether it is a ‘hit’ or a ‘miss’; if it is a ‘hit’ then it generates a ‘hit’ message and sends it to DS which broadcasts the message to DiscoTech’s clients.

Again, the Tank game is substantially different from the previously described applications as it uses a separate game server. We have still adapted different disconnection and reconnection behaviours in the Tank game using a similar set of plug-ins. For example, the Speed-up plug-in replays tank
move events quickly and the Averager plug-in generates average positions of tanks. This essentially shows the power of DiscoTech’s plug-in architecture that it is capable of handling disconnection for diverse varieties of groupware applications using similar plug-ins and without modifying the architecture. However, for longer disconnections, tank move events are transformed into a historical trace using the HistoricalTraceTransformer custom plug-in which transforms ‘tank move’ events into a ‘HistoricalTrace’ data structure. This also shows the power of DiscoTech that if needed it is possible to implement a new disconnection behaviour using the custom plug-in option, while reusing code.

7.9.4 Remote presentation tool

This application allows a presenter to present her slide decks to remote viewers in real time (discussed in Section 3.1.5.) This application is developed to study the modality fusion problem (discussed in Section 3.1.5.1). In so doing, different disconnection and reconnection behaviours are adapted in this application by reusing different plug-ins, for example, generic Truncator plug-in used to discard all slide events except the last one and generic Speed-up plug-in is used to display the annotations at double speed.

7.9.5 Collaborative gesture-based game for older adults

This application is a slow-paced game for elderly adults (discussed in Section 3.1.4). For this game, the smooth transition effect style is proposed to replay events quickly instead of using the double-speed replayer plug-in. The smooth transition effect style will blend the current state to a final state through a fade animation. We have implemented the game using the peer-to-peer overlay architecture discussed in Section 4.1. Therefore, as discussed in Section 4.2.6, the AggregateToState plug-in can be applied to manipulate stored plant grow events during
disconnections. However, we did not implement this reconnection style, as our main purpose was to study a case where the quick playback reconnection style does not work and to determine an alternative behaviour that fits well with reconnected users requirements. We also wanted to design the necessary plug-in to implement the behaviour. Therefore, the implementation and evaluation of this reconnection style remains as future work.

### 7.9.6 Summary and discussion of developing disconnection-aware groupware

Using the DiscoTech toolkit, we have developed eight disconnection-aware groupware applications. These applications were drawn from the domains of collaborative gaming, streaming media, chatting, and drawing, editing, pointing and reviewing. These examples show common ways in which disconnection behaviours are handled by DiscoTech. A small number of plug-ins (in total eleven) allows a wide range of application behaviours across several application types. Plug-ins act as general solutions to common problems that recur across several applications (e.g., truncation of awareness events, accelerated replay, and shifting to a state representation as the length of the disconnection increases).

Using one of the developed applications (i.e., the remote presentation tool), we have studied the modality fusion problem. With the gesture-based collaborative game, we showed a widely used plug-in, the *Speed-up* plug-in will not work for all disconnection scenarios.

As the DiscoTech toolkit with eleven plug-ins is capable of implementing the wide varieties of disconnection and reconnection behaviours, we can answer our fifth the research question (RQ#5, Section 1.3) that with a reasonable number of plug-ins, it is possible to implement disconnection handling for a wide range of disconnection scenarios. However, although we carefully selected the representative applications to investigate our fifth research question, there is no guarantee that we
have covered the whole spectrum of groupware applications that exist in today’s Internet-based era. In the case where the provided plug-ins including the custom plug-ins are not sufficient for a new context, the developers can use the partially generic and custom plug-ins option for handling disconnection. The designed plug-in architecture provides required features and flexibilities in building custom plug-ins where the developers can use the provided library of plug-ins and supported features in making new plug-ins.

7.10 Summary

In this chapter, we presented the DiscoTech toolkit and showed how it can be used for developing five different categories of groupware applications. In doing so, we showed the practicality of our three (the design space, the plug-in architecture and the disconnection design combos) codified solutions for handling disconnection.
Chapter 8

Empirical Evaluations of DiscoTech Toolkit

As only a few synchronous groupware are disconnection aware, developers might not know about the overhead of handling disconnection in reality (discussed in Sub-problem 4, Section 1.2.4). The overhead of handling disconnection includes extra development time, extra storage requirements and additional feedthrough time. Therefore, in order to gather evidence about the overhead of handling disconnection, we have evaluated the DiscoTech toolkit (discussed in Chapter 7). The goal of this evaluation is to provide a check that for at least two typical cases restricted to a local area network, DiscoTech does not add significant overhead compared to an implementation not using DiscoTech. This chapter presents our empirical evaluations of the DiscoTech toolkit.

In our empirical evaluation phase, we have performed late architectural evaluations [26, 74, 85, 86] of the designed plug-in architecture using the toolkit. The late architectural evaluations allowed us to empirically verify whether the toolkit with its underlying plug-in architecture can achieve our desired quality goals. As discussed in Chapter 7, our key claims are as follows.

- That DiscoTech is easy to use, with a plug-in architecture that hides the disconnection handling infrastructure, but still provides customizability;

- That DiscoTech is expressive, supporting a wide range of disconnection behaviours; and

- That DiscoTech is performant, demanding only modest overhead in feedthrough times (in a local area setting) and memory use.
In order to empirically validate our claims, we have evaluated DiscoTech based on our three quality goals: expressiveness, ease of use, and high performance. The evaluation methodologies and results are discussed as follows.

8.1 Ease of Use

We have estimated the coding effort for handling disconnection based on the *ease of use* attribute. For this, the toolkit’s API usability is analyzed by conducting a user study. In this study, we have solicited opinions of four software developers not associated with this project. Two developers were undergraduate research interns, and two were graduate students. All of them were familiar with the concepts of groupware and with development using C#, but none had programmed disconnection handling in a groupware application. We have supplied an instruction document (Appendix A) to each participant. The instruction document provides step by step guidelines to the users for showing how to install the toolkit and how to use its API with examples. The programmers were asked to adapt the disconnection and reconnection behaviours that can be implemented using either the generic plug-ins or the partial generic plug-ins in order to avoid coding complexity. Each was asked to carry out a one-hour tutorial, which showed how to convert a single-user drawing application into a multi-user version including disconnection handling. We observed each developer as he worked through the tutorial, and then conducted a semi-structured interview after the session. In the interview session, several questions are asked as follows:

- Were you able to easily follow the steps to add disconnection functionality to the application?
Did the toolkit’s plug-in architecture help you understand the range of disconnection behaviours you might put into an application?

Now that you have seen this example, can you imagine how you would use DiscoTech in your own groupware application?

Assuming DiscoTech were refined to a production-quality toolkit, do you anticipate it would be easier to develop disconnection-aware groupware using DiscoTech versus doing it by hand?

Did you find the API s of the toolkit easy to use?

All four developers were able to complete the tutorial in one hour, correctly adding multi-user functionality to the application and experimenting with a variety of disconnection behaviours by swapping plug-ins. All reported that they found the toolkit’s API straightforward, and that DiscoTech would be a suitable candidate for future projects. One participant commented that “The API is really generic… Since there is only one method call and a callback, it is very straightforward.” Others stated “I would use DiscoTech because I don’t want to reinvent the wheel.” and “I would use DiscoTech. It does [it] all for me.” While this was far from a comprehensive study, it does provide encouraging feedback that people with strong programming background and familiarity with groupware can learn the DiscoTech concepts quickly, and see value in it for programming disconnection behaviours.

**8.1.1 Limitations**

The user study for the toolkit is limited to a tutorial session where four programmers converted a single user application into a disconnection-aware groupware application using the toolkit. While
this is not a comprehensive study, we believe it still provides useful feedback and shows how programmers can learn the DiscoTech concepts, and see effectiveness in it for programming disconnection behaviours. Although the tutorial study is sufficient for demonstrating the practicality of the toolkit, a comprehensive user study with industrial participants will be an interesting piece of future work.

8.1.2 Summary and discussion of ease of use

Four programmers are recruited to use DiscoTech’s API for building a disconnection-aware groupware from a single user application. The study shows that programmers were successfully able to do their assigned tasks and they felt DiscoTech’s API is not that much complex to use.

8.2 Expressiveness

Using DiscoTech, we have developed eight groupware applications, such as a chat tool, four shared workspace editors, a Tank game, a remote presentation tool and a gesture-based collaborative game (discussed in Section 7.9). As discussed before, these applications justify the practicality of our proposed solutions, but at the same time it empirically shows the expressiveness of the DiscoTech toolkit. With these eight groupware applications, we have captured ten disconnection and reconnection behaviours (summarized in Section 3.1.6). By using DiscoTech, we were able to implement most of the disconnection related behaviours in these applications (except DRAB #10, discussed in Section 4.2.6). In doing so, we have empirically validated that DiscoTech is reasonably expressive. Moreover, in Section 7.6, we have explained that using the toolkit, it is possible to implement various existing techniques (including the latecomer support) for handling disconnection which also shows DiscoTech’s expressiveness.
8.2.1 Limitations

We have evaluated the expressiveness quality goal by implementing eight groupware applications using the DiscoTech toolkit. All of these applications support small group sizes. Therefore, further research is required to explore whether new techniques would be required to handle disconnection for large-scale distributed groupware applications, such as Dropbox [22] or World of Warcraft. Moreover, as DiscoTech is implemented using the centralized message broadcasting architecture, developers cannot implement the applications that need to follow a pure peer-to-peer groupware architecture where messages are directly sent to receivers from a sender.

8.2.2 Summary and discussion of expressiveness

DiscoTech is expressive because it is capable of implementing different disconnection related application behaviours, which are summarized in Section in Section 3.1.6 and shown in Section 7.9. Moreover, DiscoTech is capable of implementing different existing techniques explained in Section 7.6.

8.3 High Performance

In order to characterize the overhead of using DiscoTech, we have tested its effect on an application’s feedthrough time and memory requirements. These two metrics are described as follows:

- **Feedthrough time**: this is measured as the time from the event being sent by the sender client to the time that the message is received by the receiver client.
Memory usage: this is measured by calculating the size of the event queue which stores reconnection information.

8.3.1 Experimental Apparatus

We have used substantially different groupware applications in order to estimate the performance overhead of handling disconnection for different disconnection scenarios: a shared drawing editor and a chat tool. Maximum three computers are used for the performance testing:

- Laptop Intel (R) Core (TM) i3 CPU M330@ 2.13 GHz and RAM 4.00 GB, 64 bit Windows 7 Operating System
- Desktop PC with Intel(R) Core (TM) i7 -2600 CPU @ 3.47 GHz, RAM 8 GB, 64 bit Windows 7 Operating System.
- Laptop Pantium (R) Dual Core CPU T4200 @ 2.00 GHz, RAM: 2.00 GB, 32 bit Windows XP Operating System.

The first laptop mentioned in the above list was used for running the DiscoTech server and rest were used for running maximum eleven client applications.

Automated random inputs were generated for the both applications. These random inputs allowed us to run experiments for a longer period of time (e.g., 12 hours). We ran the experiments in a home local area network with high speed Internet connection (100 Mbps). In experiments one and two (discussed below), we opted ten messages per second in order to create a moderate traffic load. According to Whalen and Black, “if network traffic becomes heavy, users are subject to long delays and connection loss” ([101], p. 20). Moreover, message rate depends on update rate of groupware
applications and hence, varies from one application to another. According to Gutwin et al., “groupware applications need to send updates at maximum rate of about 25 messages per second” ([45], p.173). Therefore, by choosing the message rate ten for the both applications, we attempted both to maintain a moderate traffic load and an average update rate.

8.3.2 Methodology

Three different experiments were conducted to measure the overhead of DiscoTech on applications’ performance. Three experiments are summarized as follows:

- Experiment 1: The feedthrough time is compared with and without enabling the queuing architecture of the DiscoTech toolkit during the connected situation. This evaluation step measures the overhead of handling disconnection during normal operations of the applications.

- Experiment 2: The feedthrough time is tested with the increasing number of disconnected clients. This step checks whether the increasing number of disconnected clients effects the feedthrough time.

- Experiment 3: The memory usage is compared with and without compaction enabled during the disconnected situation. This step verifies whether the use of compactor plug-ins can reduce the memory requirements of handling disconnection during a disconnection period.

In order to estimate the feedthrough times, we have run the sender and receiver clients at the same computer. In doing so, we have avoided the problem of synchronizing the clocks of different computers. DiscoTech’s sender client adds the timestamp to a message before sending it over the
network. When DiscoTech’s receiver client receives the message from the server, it subtracts the timestamp from the current time and the difference of time is calculated in ms. The following API call is used to measure feedthrough times.

```csharp
_feedthroughTime = DateTime.Now.Subtract(msgSentTimeStamp).TotalMilliseconds;
```

We have run these experiments several times to determine the control variables or factors that impact the experimental results. We have found the experimental metric feedthrough time is sensitive to CPU speed, usage and bandwidth rate.

In order to have low CPU usage, we closed all applications except the Microsoft Visual Studio 2010. In order to have maximum bandwidth rate, during experiments one and two, we did not allow anyone to browse Internet in the home network setting. Although we have run several experiments, we have reported one run for each experiment that showed the best outcome.

If someone runs experiments one and two in different experimental setup (e.g., with different computer hardware, CPU usage and bandwidth rate), it is highly possible that different results for feedthrough time may come.

8.3.2.1 Experiment 1

This experiment shows that DiscoTech’s effect on feedthrough time is negligible during the normal connected situation both for the shared drawing editor and the chat application. The experiment with the two applications is described as follows:

**Shared drawing editor.** We ran a shared drawing editor (discussed in Section 3.1.2.1) under two conditions: with and without disconnection handling enabled. In the “no disconnection handling”
condition, DiscoTech’s server-side queue was removed, and the server simply forwarded incoming events to all clients, thus simulating a traditional message broadcasting architecture. In both conditions, a ‘sender’ client was automated to inject a new random drawing command every 100 ms, over a total of 60 seconds. Between one and nine additional ‘receiver’ clients received these drawing events. Feedthrough time was measured as the time from the event being sent by the sender client to the time that the message was received by a receiver client. As explained earlier the sender and receiver clients resided in the same computer. A local area network was used between the client and server computers with ping times not exceeding one milliseconds. The results of this experiment are shown in Figure 8-1 that the impact of DiscoTech’s message queuing on feedthrough time is negligible for the drawing editor.

**Chat tool.** As of the shared drawing editor, we ran the chat tool (discussed in Section 3.1.1) under two conditions, with and without disconnection handling. For this application, random character messages are generated with the same setting as of the shared drawing editor. The result of this experiment is shown in Figure 8-2 that shows that the impact of DiscoTech’s message queuing on feedthrough time is negligible for the chat application as well.

![Graph showing feedthrough time comparison between Simple Message Broadcasting and DiscoTech Queuing Architecture](image-url)

**Figure 8-1:** DiscoTech feedthrough time for shared drawing editor
8.3.2.2 Experiment 2

This experiment shows that DiscoTech’s effect on feedthrough time is minimal for an increasing number of disconnected clients. For this experiment, both the canonical share drawing editor and the chat applications were used.

In this experiment, the feedthrough time is measured by increasing the number of disconnected clients. A ‘sender’ sends events using the similar fashion as of the first experiment. For this scenario, the server must use its event queue for each disconnected client to store information allowing the disconnected clients to be brought up to date following reconnection. Figure 8-3 shows the results of this experiment.

The graph in the figure shows that the impact of DiscoTech’s message queuing on feedthrough time with the increasing number of disconnected clients is negligible for the both applications.

For the drawing program, the feedthrough time difference between zero to ten disconnected clients is only 0.278 ms and for the chat application the difference is 0.065 ms. Although both values are small, the shared drawing editor has little bit higher feedthrough values than the chat application.
The reason we noticed that feedthrough values are sensitive to the bandwidth rate and the CPU usage, so little higher values might come due to bandwidth condition or high CPU usage. The overall results will be different if bandwidth condition and CPU usage are changed.

8.3.2.3 Experiment 3

This experiment shows that by using compactor plug-ins, it is possible to control the memory use during the disconnection time. This experiment measures DiscoTech’s memory usage when a client is disconnected. For this experiment, we tested the drawing program, with one “sender” client, one connected “receiver” client, and one disconnected “receiver” client. In this scenario, as of Experiment 2, the server must use its event queue to store information allowing the disconnected client to be brought up to date following reconnection.

The application transmitted two forms of events: telepointer movement events, and drawing commands. We measured server-side storage usage under two conditions. The “without compaction” condition measured the space required to store the telepointer and drawing messages as-is, with no attempt to compress them to a smaller size. The “with compaction” condition used
partially generic plug-ins to reduce storage size: (1) drawing events: any drawing events occurring over the last ten minutes are represented in the queue as sent by the client; older drawing events are applied to a special bitmap message (using an AggregatetoState compactor). (2) Telepointer events: the last minute of telepointer events are stored as sent by the client; events between one and ten minutes old are averaged to provide at most one event per five seconds (using an Averager compactor), and events older than ten minutes are converted to a heatmap (using HeatmapTransformer).

Each condition was run for 12 hours (720 minutes). The results are summarized in Figure 8-4. In the “without compaction” condition, memory use climbed steadily over time. This is as expected, as events are continuously added to the event queue. Interestingly, even after 12 hours of disconnection, only 15 MB was required to store data from the drawing program. This hints that even with no compaction, small-group shared workspace groupware applications such as a drawing program can sustain lengthy disconnections with acceptable overhead.

**Figure 8-4: Memory use during disconnection**
In the “with compaction” condition, memory usage reaches a steady state after approximately one hour, at approximately 0.4 MB. This indicates that the aggregation to state strategy works well, where individual events are retained to allow progressive update (e.g., through double-speed replaying), while a full state snapshot allows aggregation of large numbers of events. The limitation of space to 0.4 MB also allows considerably faster reconnection than under the “without compaction” condition, as less data needs to be transmitted to the client.

This shows that with the use of parameterized plug-ins, DiscoTech can have modest and bounded storage requirements when handling disconnection. These results may depend on the kind of application being tested. Our drawing surface was limited to 500 x 500 pixels; a larger drawing surface would have required more storage space. Similarly, trying to save the full contents of a video chat on a server would require considerably more storage. Nevertheless, for small-group shared-workspace groupware applications, this test indicates that storage requirements are well within the capacities of modern hardware.

### 8.3.3 Limitations

In experiments one and two, we have run multiple clients in a single computer which might increase the CPU usage and can contribute to get higher feedthrough time. Another limitation is that in experiments 1 and 2, we have used a single run of the experiments to prepare the results. Although the reported single-run results are stable and expected, the average values of the multiple runs would add more stability in our obtained results.

Our experiments were limited to a local area network (LAN). Although we have shown that the DiscoTech toolkit adds negligible overhead in a LAN setting, it remains to be tested how the
overhead looks like in the wide area setting where the DiscoTech server and clients will reside in different cities.

8.3.4 Summary of three performance experiments

In order to empirically measure the performance overhead of disconnection handling in groupware, we have conducted three experiments. In experiments one and two, we have shown that disconnection handling does not have that much impact on feedthrough time both in connected and disconnected situations, which we expected. In experiment three, we have shown that by applying compactor plug-in, we can control the memory usage during disconnection.

However, we have noticed that feedthrough time is sensitive to different factors, such as computer hardware, CPU usage and bandwidth rate. Therefore, results could be different if the factors’ values are changed.

8.4 Summary and Discussion of Empirical Evaluation of DiscoTech

This chapter presented DiscoTech’s empirical evaluations in three areas in order to show its conformance to achieve the three quality goals: ease of use, expressiveness, and high performance. First, we showed that the toolkit is easy for developers to learn and use by reporting on tutorial sessions with four programmers. Second, we demonstrated the expressiveness of the toolkit by specifying its capability of building five different kinds of groupware applications and supporting policies from previous work (including latecomer support). Third, we provided empirical evidence that DiscoTech demands only modest overhead in feedthrough times for both connected and disconnected clients restricted to a local area network setting, and that memory use is appropriately constrained when people are disconnected. In short, the evaluation results show that DiscoTech
handles a wider range of issues than previous toolkits, without requiring undue effort, and provides a practical way to improve the real-world usability of synchronous groupware while demanding little performance overhead. Thus, by developing and evaluating the DiscoTech toolkit, we get the answer of our sixth research question (RQ#6: Does disconnection handling demand little overhead to the performance of a groupware application and little programming complexity for addressing different disconnection scenarios?, discussed in Section 1.3).
Chapter 9

Conclusion and Future Work

This chapter first summarizes the thesis. Then, it specifies its contributions, and limitations. Finally, it mentions some important future work and concludes the thesis.

9.1 Thesis Summary

Synchronous distributed groupware is widely used in our Internet-based era. Disconnection and reconnection are common problems for users of synchronous groupware. Disconnection causes information loss and results in different user-level problems (e.g., confusion and interpretation difficulties) upon reconnection. In order to overcome such user-level disconnection problems, disconnection handling is important, allowing a reconnected user to see missed events quickly but in an understandable manner. However, disconnection handling in synchronous groupware is difficult as it involves tackling of different disconnection scenarios and timeframes. In this thesis work, we provided codified solutions and a plug-in based toolkit for handling disconnection in groupware.

We have designed eight groupware applications (the chat tool, four shared workspace editors, Tank game, remote presentation tool and gesture-based collaborative game, discussed in Chapter 3) for eliciting motivating disconnection scenarios. We have determined what kinds of user-level problems might occur if disconnection handling is not supported in these disconnection scenarios while considering different lengths of disconnections. With these applications, we have also determined ten disconnection and reconnection behaviours that can be supported to mitigate the
user-level disconnection problems (summarized in Section 3.1.6). In order to adapt these behaviors in the applications, a wide range of strategies are required. However, developers usually have a lack of knowledge about the disconnection design space that covers strategies for handling disconnection. More effective strategies than store and forward [11, 12] and event replay [59, 60] are neither well known nor studied. Consequently, current groupware applications typically only support limited disconnection and reconnection behaviors (e.g., Skype can only show the missed chat conversation all at once, but it cannot show a summary of chat conversations if a user reconnects from a long-term disconnection). When disconnections last longer, developers need to manage the trade-off between performance and understandability requirements (discussed in Section 5.1). However, it is a difficult task to do because the trade-off occurs in different disconnection situations (e.g., when handling disconnection in awareness events, in model update events, and in events requiring both compaction and pacing, discussed in Section 5.3 and shown in Figure 5-1) and its solutions vary in each situation. No toolkits or development infrastructures exist to help developers implement the strategies for handling disconnection in a simple and flexible manner. Therefore, developers need to write a great deal of code to implement the strategies from scratch. Moreover, little evidence exists about the overhead of disconnection handling in groupware. Developers do not know how much overhead disconnection handling might demand to the performance of their applications, e.g., how much extra memory or feedthrough time will be needed to support disconnection handling.

In order to simplify developers’ tasks in handling disconnection, we have provided three codified solutions (such as the disconnection design space (Chapter 4), three disconnection design combos (Chapter 5) and the plug-in architecture (Chapter 6) and built a plug-in based toolkit called DiscoTech (Chapter 7) based on the solutions. The codified solutions capture and categorize
different strategies for handling disconnection and show how to combine and implement the strategies in a simple and reusable manner. We have shown that by using DiscoTech with eleven plug-ins (that implement the strategies, discussed in Section 7.4 and summarized in Table 7-1), it is possible to handle disconnection for five types of groupware applications (in total eight groupware applications, summarized in Section 7.9). We have also shown that disconnection handling using the toolkit does not incur substantial overhead (rather little overhead) to applications’ performance in terms of feedthrough time (discussed in Sections 8.3.2.1 and 8.3.2.2) in a local area network setting and memory usage (discussed in Section 8.3.2.3). We have conducted a user study with the toolkit by recruiting four programmers (discussed in Section 8.1). Their feedback shows sufficient indications that DiscoTech is capable of simplifying development of a disconnection-aware groupware application.

By developing the four codified solutions, we are able to simplify developers’ tasks for handling disconnection in different disconnection scenarios and timeframes, to reuse solutions from one scenario to another, and to show evidence about the overhead of handling disconnection in synchronous groupware. Thus, with this thesis work, we can answer our six research questions, discussed in Section 1.3.

9.2 Contributions to Knowledge

This research contributes novel ideas and knowledge to the field of collaborative software engineering. This work provides groupware developers with the codified design knowledge on disconnection to improve their understanding in handling disconnection in synchronous groupware, and the technology to implement disconnection handling in their applications. At the same time,
this thesis serves as a foundation for future research projects in disconnection handling. There are four major contributions as follows:

- **The five-dimensional disconnection design space:** It informs developers about a wide variety of strategies for manipulating stored events during disconnection. These strategies are important for developers to know in order to satisfy users’ requirements upon reconnection and to resolve performance issues.

- **Three disconnection design combos:** These combos help developers in selecting and combining a set of strategies from the disconnection design space in order to manage the trade-off between performance and understandability requirements in three disconnection design situations.

- **The plug-in software architecture based toolkit:** The toolkit simplifies disconnection handling for groupware developers by satisfying three quality attributes: expressiveness, ease of use and high performance.

- **Evidence about the overhead of disconnection handling in synchronous groupware:** This evidence provides practical data about the disconnection programming complexity, performance overhead and range of disconnection and reconnection behaviours that developers need to consider for handling disconnection.

There are two other minor contributions of this research. First, it increases the current understanding of the user-level disconnection problems. Second, several new disconnection-aware
groupware applications serve as example applications that can demonstrate how to handle disconnection in practice.

9.3 Threats to Validity

The primary limitations of this thesis include:

- While we have evaluated the designed architecture by designing and developing eight groupware applications, such a moderate sample size might not guarantee the completeness of the architecture. There is a wide range of groupware applications in today’s Internet-based era, and thus there is always a possibility that we might have missed important disconnection and reconnection requirements. In order to mitigate this issue, we have carefully chosen five diverse varieties of applications (discussed in Sections 3.1 and 7.9) from different domains, e.g., applications are selected from the domain of collaborative gaming, streaming media, online chatting, remote presentation and shared editing. Thus, we believe that such a diverse set of applications would work as a reasonable representative for capturing a wide range of disconnection and reconnection requirements for other similar types of applications as well.

- The user study of the DiscoTech toolkit was performed using a tutorial session where four programmers converted a single user application into a disconnection-aware groupware application using the toolkit. The study shows that DiscoTech is indeed useful in programming disconnection and reconnection behaviours in groupware applications. However, although the study was sufficient for demonstrating the practicality of the toolkit, a comprehensive user study with industrial participants can be conducted in future work.
In our performance evaluation studies, we have shown that the plug-in based DiscoTech toolkit incurs minimal performance overhead to groupware applications in terms of feedthrough time and memory use. However, our experimental settings were limited to small group sizes (maximum eleven clients), limited to moderate disconnection period (maximum twelve hours) and in a local area setting. Although the DiscoTech toolkit adds negligible performance overhead in the limited settings, it remains to further study its scalability with large group sizes (e.g., over thousands of users), lengthy disconnection periods (e.g., more than a year) and wide area network settings.

9.4 Future Work

Our initial investigation into handling disconnection based on users experience leads to a series of open questions. Areas requiring further research are presented in the sections below.

9.4.1 Future studies

The user study and performance experiments presented in this thesis were limited in application setup, participant population, local area network setting and study design. These factors can be explored in more detail in future studies. We have made the toolkit freely available ([20]) in order to open the study opportunities to the CSCW community.

In the DiscoTech toolkit, we have only applied the concepts of disconnection design combos, but we did not offer any API to use the combos directly. Therefore, DiscoTech can be extended for offering APIs for the design combos. Moreover, it will be interesting to study whether the API can further improve DiscoTech’s usability.
9.4.2 Supporting offline work

During disconnections, some groupware applications allow participants to work offline. Upon reconnection the changes are merged to applications’ current state. For example, when connection is lost, DISCIPLE [50] allows a mobile client to work offline, and then the changes are merged when the connection is re-established. This is an interesting feature to have for some groupware applications (e.g., for the Tank game and the shared coding editor). For example, during a disconnection, the Tank game can keep connected players’ tanks moving in a disconnected player’s game interface by using a dead reckoning algorithm [1, 16], rather than suddenly freezing their movements. The dead reckoning algorithm allows the disconnected game client to predict the next possible moves of the connected players’ tanks. Thus, with this feature it is possible to hide disturbances of frequent short-term disconnections from users and to improve the usability of groupware applications. DiscoTech can be extended to accommodate this feature as well.

9.4.3 Supporting disconnection due to network latency

Simulating DiscoTech for handling disconnections due to network latency (discussed in Section 2.2.1) can be performed. As the network connection remains valid between two end points, the DiscoTech server is not aware of this type of disconnection and the receivers only experience interruptions due to variance in network latency (which is called jitter [45]). As a result, disconnections due to network latency are different from other types of disconnections caused by network outages, power failures or explicit departures.

Artificial delays can be injected into the system in order to simulate the disconnections due to network latency. In doing so, functionality of the Disconnection Identifier (discussed in Section 6.2.2.2) and the Reconnection Manager (Section 6.2.2.4) components might need to be extended.
For example, the Reconnection Manager should be able to receive notifications from the Disconnection Identifier about the connection status of a user based on the time gap between two consecutive messages (discussed in Section 6.2.2.2). There should be a way for the Disconnection Identifier to inform the Reconnection Manager about the types of disconnections.

9.4.4 Providing disconnection awareness

In future work, we can extend DiscoTech in order to support disconnection awareness that can prevent connected users from interacting with a disconnected user. If a user suddenly becomes disconnected, it is hard for connected users to know about the disconnection. For example, as discussed for the case of Tank game (Section 3.1.3), a player can become disconnected for a few seconds due to a network outage, and a connected user can shoot her tank. Since the tank will remain unresponsive to the shooting (because of the disconnection), the connected user might become confused. For example, she might think that there is a problem with her own game client and thus might restart it. This kind of confusion and subsequent actions are unpleasant and unnecessary.

Therefore, it would be useful, if groupware applications provide disconnection awareness with visual feedback, which will prevent connected users from interacting with a disconnected user. However, the visual feedback might differ based on lengths of disconnections and disconnection scenarios. For example, the Tank game might show a red circle around a tank (as shown in Figure 3-12(a) and discussed in Section 3.1.3 and elsewhere [43]) during a short-term disconnection (e.g., below ten seconds), whereas during a long term disconnection, it might remove the tank from the game interface. In contrast, other applications might use different other types of feedback to provide disconnection awareness, e.g., the paper reviewing application (discussed in Section 3.1.2.2) might
gray out a disconnected reviewer’s telepointer gestures. Therefore, in future work, it will be interesting to find out what sorts of visual feedback are appropriate for particular applications and lengths of disconnections and how we can use DiscoTech to program for disconnection awareness in these contexts.

9.5 Conclusion of the Thesis

Disconnection handling allows a user to understand both the state of the workspace and the current activities of other people upon reconnection. As there are different disconnection scenarios and timeframes, developers often experience trouble in handling disconnection. In order to simplify developers’ tasks in handling disconnection, we have provided codified solutions and a plug-in based toolkit in this thesis. The codified solutions provide knowledge to developers for understanding different aspects of disconnections in synchronous groupware. The toolkit justifies the practicality of the provided solutions by allowing the development of different groupware applications with minimum performance overhead and makes disconnection handling feasible for groupware developers by providing ease of use API. In future work, we plan to adapt more features, such as allowing a user to continue while disconnected, and adapting reusable solutions for providing disconnection awareness (discussed in Section 9.4) in the toolkit. We also plan to evaluate the toolkit with comprehensive real world scenarios.
References


70. Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. 

71. Pinelle, D., Dyck, J., and Gutwin, C. Aligning Work Practices and Mobile Technologies:
   Design for Loosely-Coupled Mobile Groups. In *Proceedings of Human-Computer


73. Preguica, N., Martins, L., Domingos, H., and Duarte, S. Integrating Synchronous and
   Asynchronous Interactions in Groupware Applications. In *Proceedings of International

74. Prechelt, L., Unger, B., Tichy, W. F., Brssler, P., and Votta, L. G. A Controlled Experiment
   in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on


   Applications using Replicated Objects. In *Proceedings of Computer Supported Cooperative


78. Roseman, M., and Greenberg, S. Building Real-time Groupware with GroupKit, A


Appendix A
Instructions for Using DiscoTech Toolkit

Table of contents

1. Introduction
2. Working with the client application
3. Working with the server application
4. Running the multi-user sketching groupware application
5. Adapting disconnection and reconnection behaviour
6. Conclusion

1. Introduction

You will create a groupware version of a sketching application, starting from a single user application. Then you will improve the application’s usability by improving its behaviour during and after network disconnections.

- You need to understand how the single user application works. These instructions and comments in the source code are provided to help you understand the source code.
- You need to create two projects: one for the client application (Windows form application) and another for the DiscoTech server (console application). In the client application, you need to use DiscoTechClient.dll and in the server application, you need DiscoTechServer.dll to provide the access to the DiscoTech API.
- You need to treat the existing single user application as a client application.
You need to refactor the single user application’s source code using the DiscoTech toolkit’s API so that it serves as a groupware client application.

You need to know two things to do the refactoring: (1) how to send messages using DiscoTech’s `SendMessage` API, and (2) how to interpret received messages in the application using the `OnReplayEvents` event-handling method.

Please keep these things in mind:

- Your client application needs to both send and receive messages because when you use a groupware application, you send messages over the network specifying changes made by the user of this client, and you receive messages over the network specifying changes made by other clients.

- If you use any data to display something in the drawing window or shared workspace, you need to send the data to remote users so that they can see the same thing. For sending data you will use the `SendMessage` method in the DiscoTech API. For example if you draw a line in your window, you need to send the line points to the remote clients so that these clients can draw the line. To represent a line, you need two coordinates \((x_1, y_1, x_2, y_2)\), and you need to specify that you are drawing a line. Using the API, you might specify: `SendMessage(“DrawLine”, x1, y1, x2, y2)` where first argument specifies the type of command, and the remaining four parameters are string values specifying the line’s endpoints. The signature of the `SendMessage` API call is:

```java
public void SendMessage(string MessageTag, string Data1, string Data2, string Data3, string Data4);
```

Note that, programmers can use ten-overloaded `SendMessage` API for sending maximum eleven parameters, and that each parameter must be a string.
When a remote user performs a command, the local client application needs to receive that command (via the network) and execute it locally. For example, consider that a remote client has sent the line drawing message sent in step 2 above. The local client application receives the message using the OnReplayEvent event handling method, which has a parameter `ReplayEventArgs ra`. "ra" has a field called `ReceivedEvent`. `ReceivedEvent` stores all the parameters that were passed to the SendMessage method on the remote client. It stores the data as the same order as they were sent. For example, if `re` is the instance of `ReceivedEvent` then it will contain the following values:

```
re.MessageTypeTag = "DrawLine", re.Data1 = x1, re.Data2 = y1, re.Data3 = x2, re.Data4 = y2
```

Within `ReplayEventArgs`, you need to interpret the message so that you can display the line as specified.

- The `OnReplayEvent` described above is part of the application. An additional step is required to tell the DiscoTech framework how to pass on remote messages to the client application. The simplest behaviour is to pass on messages to the client application as soon as they arrive over the network. Alternatively, you can use the `DoubleSpeed` replayer to replay messages at double-speed following reconnection, allowing quick catch-up.

- You need to run the server before running the client application. The server receives messages from clients and forwards them to other clients. If a client is temporarily disconnected, the server queues all messages during the disconnection period so that they can be sent once the client reconnects. The server can use the compactor plug-ins to reduce the volume of the stored information. You can add multiple compactor plug-ins (such as `AgedDataTruncation` and
AwarenessDataTruncation) to compact information differently during different timeframes of the disconnection period.

- You can also build your own custom compactors. For this you need to add the CompactorPlugins project.

2. Working with the client application

- Unzip the DISCOSourcecode Folder
- Go to ..\DISCOSourcecode\GWAppsWithDISCO\SketchingAPP\SketchingAppClient
- Open FreeSketchingApp project and compile the application from the IDE to check whether it works or not.
  - Press left mouse button and move the mouse. You will see blue lines. Close the running application.
- Go to the source code and take some time to understand how the source code works. Some hints are:
  - Go to in SketchingForm_MouseMove method.
  - When you move the mouse with the left mouse pressed, a line is created with current mouse position (e.X and e.Y) and previous mouse position (_lastX and_lastY). This line is stored in a dictionary variable (_Sketches) which is keyed by the user name (the local user name is stored in _myName variable).
  - The first starting point of the line is stored when you press the left mouse down, which is stored in _lastX and _lastY variables (see SketchingForm_MouseDown method).
  - The SketchingForm_Paint method is called periodically which draws the lines from the dictionary using drawGSkeecthes method.
In the client application (FreeSketchingAppClient), add the reference using the reference tab. Browse the DiscoTechClient.Dll in the path

- ..\DISCOSourcecode\GWAppsWithDISCO\SketchingAPP\SketchingAppClient.

See Figure 1 below.

![Figure 1: Adding reference](image)

- Write using DiscoFramework in SketchingForm.cs;

- At the constructor of your application add the following lines of code. You need to have the IP of your own server.
  - IPAddress serverIP = IPAddress.Parse("128.233.109.193");
  - StartDiscoTech(serverIP);

- Copy the following lines of code and paste under the constructor method. Fix errors if you see in the source code for the new line.
#region Start DiscoTech

private void StartDiscoTech(IPAddress serverIP)
{
    //Enabling DiscoTech’s Queuing architecture
    DISCOFramework.Instance.WithMessageStoring = true;
    //Initialize DiscoTech with the server IP and port number.
    DISCOFramework.Instance.Initialization(serverIP, 8010);
    //Subscribing to "ConnectionSucceed" event to see whether the
    //connection to the server is established or not.
    DISCOFramework.Instance.ConnectionSucceed += new
    ConnectionSucceedEventHandler(Instance_ConnectionSucceed);
    //Subscribe to "ConnectionFailed" event to see whether the
    //connection attempt to the server fails or not.
    DISCOFramework.Instance.ConnectionFailed += new
    ConnectionFailedEventHandler(Instance_ConnectionFailed);
    //Subscribe to "ReplayEvents" event to receive events from
    //DiscoTech at normal connected situation
    DISCOFramework.Instance.ReplayEvents += new
    ReplayEventHandler(OnReplayEvents);
}

#endregion

#region DiscoTech Handlers

private void Instance_ConnectionSucceed (object sender, EventArgs e)
{
    MessageBox.Show(“Connection is established successfully”)
}

private void Instance_ConnectionFailed (object sender, EventArgs e)
{

205
MessageBox.Show("Server is not connected: please try again");
}

System.Threading.Semaphore semaphor = new System.Threading.Semaphore(1, 1);

public void OnReplayEvents(object sender, DiscoFramework.ReplayEventArgs re)
{
    semaphor.WaitOne();
    try
    {
        // you need to write code here to interpret the receive event according
        // to the application requirements.
        semaphor.Release();
    }
    catch (Exception ex)
    {
        semaphor.Release();
    }
}

#endregion

- Add the following lines of code in signInToolStripMenuItem_Click method

    // set the user name to the local client name
    DISCOFramework.Instance.LocalUserName = "Debby";
    // Connect the client to the server
    DISCOFramework.Instance.ConnectToServer();

- Go to SketchingForm_MouseMove method. Use the SendMessage API for sending mouse positions over the network. For this application you need to send two types of messages: one is “DrawingSketch” (when you are sketching) and the other one is “TelepointerAwareness”
(when you are not sketching but moving the mouse). The “TelepointerAwareness” message is for awareness purpose.

There are five overloaded sendMessage methods. For example, if you want to send a single data item over the network, you can use the following API:

```csharp
    public void SendMessage(string MessageTag, string Data1)
```

If you want to send two Data items, you have to use the following API

```csharp
    public void SendMessage(string MessageTag, string Data1, string Data2).
```

With the MessageTag parameter, you can specify the name/type of the message in order to distinguish one message from another. Also using this MessageTag you can specify the type of a message, e.g., awareness message or line draw message. You to need to tag the word “Awareness” in the MessageTag string. For example for sending the sketching data, you can write the following lines of code:

```csharp
    DISCOFramework.Instance.SendMessage("DrawingSketch",
    e.X.ToString(), e.Y.ToString(), _lastX.ToString(), _lastY.ToString());
```

For sending the Telepointer move data, you can write following lines of code:

```csharp
    DISCOFramework.Instance.SendMessage("TelepointerAwareness",
    e.X.ToString(), e.Y.ToString());
```

- To receive messages from other remote users, you have to go to the following code segment.

```csharp
    public void OnReplayEvents(object sender, DiscoFramework.ReplayEventArgs re)
    {
    }
```
ReplayEventArgs has a field named ReceivedEvent which has following properties:

- StringTypeTag
- Data1, Data2, Data3, Data4, Data5
- SenderName
- SenderIP
- TimeStamp

In this method segment, you have to write code in a way that will display the same things as the remote users doing. Take a moment and think how you will do that. One clue is that you can add the sketching point to the dictionary where you store all the local sketching points. But you need to use different user/client name this time. The client name would be the sender name of the received event. For example, you need to write following lines of code in OnReplayEvents method:

```csharp
if (re.ReceivedEvent.MessageTypeTag.Contains("DrawingSketch"))
{
    AddSketchData(Int32.Parse(re.ReceivedEvent.Data1),
                   Int32.Parse(re.ReceivedEvent.Data2),
                   Int32.Parse(re.ReceivedEvent.Data3),
                   Int32.Parse(re.ReceivedEvent.Data4), re.ReceivedEvent.SenderName);
}
```

For drawing the telepointer cursor, you can track what kind of message you are receiving. For example you can declare a variable named _remotemsgType as follows:

```csharp
private MessageType _remotemsgType = MessageType.None;
```

You also need to declare three more variables to keep the current remote telepointer position and user name. In the “declare variables” region declare the following variables:

```csharp
private int _remoteX, _remoteY;
private string _remoteUserName;
```

Then add the following code in OnReplayEvents method:
else if (re.ReceivedEvent.MessageTypeTag.Contains("TelepointerAwareness"))
{
    _remotemsgType = MessageType.TelepointerMoveAwareness;
    _remoteX = Int32.Parse(re.ReceivedEvent.Data1);
    _remoteY = Int32.Parse(re.ReceivedEvent.Data2);
    _remoteUserName = re.ReceivedEvent.SenderName;
}

In the SketchingForm_Paint method, add the following lines to display the remote telepointer cursor.

if (_remotemsgType == MessageType.TelepointerMoveAwareness)
    drawMousePointer(_remoteX, _remoteY, Color.Blue, _remoteClientName, e);

- In SketchingForm_FormClosing event method add the following line

    DISCOFramework.Instance.CloseDiscoTech();

3. Working with the server application

- Go to ..\DISCOSourcecode\GWAppsWithDISCO\FreeSketchingAPP\SketchingAppServer
- Open the SketchingAppServer project
- Set the server IP to your current machine. For example

    IPAddress serverIP = IPAddress.Parse("128.233.109.193");

4. Running the multi-user sketching groupware application

- Run the DiscoTech Server.
- Run the two client applications. Go to Login menu and click Sign In for connecting to the server.
- Start sketching. You should be able to see the same to your other client. Congrats you have built the sketching groupware application successfully! Now you need to add various plug-ins to adapt various disconnection behaviours. Follow section 3.

5. Adapting disconnection and reconnection behaviours

- **Speed-up Playback.** If you want to display events quickly, you need to use the *Speed-up* plug-in (Table 7-1). You need to add the following lines of code in the "StartDiscoTech" method:

```csharp
// Events from index 2 events will be replayed quickly
int startingeventindex = 2;
ReplayerPluginAbstractClass rp = new SpeedUpReplayer(startingeventindex);
rp.ReplayEvent += new ReplayEventHandler(OnReplayEvents);
DISCOFramework.Instance.AddReplayerPlugin(rp);
```

To see whether it works or not, unplug your network cable for one client and continue working with the other client. After 5-10 seconds, you re-plug the network cable and Login. You will see the replay of your drawing events at a double speed in the reconnected client.

- **Awareness Data Truncation.** If you want to remove telepointer awareness messages, you need to use the *Truncator* plug-in (see Table 7-1) with relevant parameters settings. Write following lines of code in the **StartDiscoFramework** method of SketchingAPPSever:

```csharp
// start and end times of the timeframe
int startTime = 10, endTime = 20;
Compactor cAwarness = new Truncator(startTime, endTime);
cAwarness.MessageTag = "TelepointerMoveAwareness"
DISCOFramework.Instance.AddCompactor(cAwarness);
```
The first and second arguments are the timeframe of the compactor. Between the disconnection periods of 10 sec. - 20 sec. this compactor plug-in will be executed. The ‘MessageTag’ property will allow you to discard only the telepointer awareness messages. To see it is working, disconnect and reconnect the client as mentioned above.

- **Aged Data Truncation.** If you want to keep only last 20 seconds of data, you need to use the Truncator plug-in (see Table 7-1) with appropriate parameters settings. To add the compactor, you need to write following lines of code:

```c
int ThresholdTime = 20;
Compactor cAged = new Truncator(ThresholdTime, 21, 60);
DISCOFramework.Instance.AddCompactor(cAged);
```

Here the first argument is for setting the threshold time (20) and the second and third arguments are for setting the timeframe of the compactor. Between the disconnection periods of 21 sec. - 60 sec. this compactor will be executed. To see it is working, disconnect and reconnect the client as mentioned above.

- **Averaging Data.** If you want to average telepointer positions, you need to use the Averager plug-in (see Table 7-1). For this, you need to add the CompactorPlugins project. To add the compactor, first you need to add this project in your solution (see Figure 2). It is in the `\DISCOSourcecode\CustomPlug-ins\CompactionPlugins`. 
Then in the reference tab of your application you need to add the project CompactorPlugins. In the CompactorPlugins project, add the reference DiscoTechServer.dll (in the path ..\DISCOSourcecode\CustomPlug-ins\CompactionPlugins)

The *averager plug-in* aggregates telepopinter positions within a particular time gap. Write following lines of code for using the average plug-in:

```csharp
float intervalThreshold = 1.00F;
startTime =0, endTime = 10;
Compactor cTAG = new
Averager(intervalThreshold,startTime,endTime);
DISCOFramework.Instance.AddCompactor(cTAG);
```

Here the first argument is for setting the interval (one second) and the second and third arguments are for setting the timeframe of the compactor. Between the disconnection periods of 0 sec. - 10 sec., this compactor will be executed. To see it is working disconnect and
reconnect the client as mentioned above. However, you can write the Aggregate method based on your requirement if the exiting aggregation method doesn’t work for you.

6. Conclusion

This appendix chapter shows how a programmer will convert a single user application into groupware application using DiscoTech. It also shows how the programmer would adapt different disconnection and reconnection behaviours in the shared sketching groupware application using different plug-ins.