A

CONTEXT-AWARE COORDINATION CONTROL IN DISTRIBUTED COLLABORATIONS

by

ALI SHIHAB SABBIR

A dissertation submitted to the Graduate School Faculty in Computer Science in partial fulfillment of the requirements for the Degree of Philosophy, The City University of New York

2005

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 3169976

Copyright 2005 by
Sabbir, Ali Shihab

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform 3169976

Copyright 2005 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

©2005

ALI SHIHAB SABBIR

All Rights Reserved

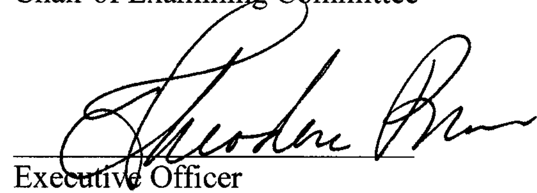
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This manuscript has been read and accepted for the Graduate Faculty in Engineering in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

02/14/05
Date

Chair of Examining Committee

2/15/05
Date



Executive Officer

Professor Michael Anshel

Professor Gwang S. Jung

Professor M. Umit Uyar

Dr. Kevin A. Kwiat

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Context-Aware Coordination Control in Distributed Collaborations
by
Ali Shihab Sabbir

Advisor: Professor Kaliappa Ravindran

In a distributed collaborative application, a key requirement is that all users see the same copy of a shared window object at any given point in time (WYSIWIS) to maintain cohesive context of the problem. We recognize that user actions are generated based on their understanding of the context of a problem, where context is defined in terms global view of shared window object. We note however that, actions generated from different sites in response to the same globally agreed upon context may not be compatible to each other since execution of one action may change the context in a way such that other actions become irrelevant. This notion of compatibility of action is the crucial distinguishing factor between a distributed collaborative system and other distributed systems where independent actions are serializable. We formalize distributed collaboration as a finite state machine model (FSM) and propose an epoch-based implementation of this model. We formalize the notion of compatibility in terms of FSM, and also propose a pragmatic, rule based implementation of compatibility enforcement. We then provide distributed protocols for a system-level realization of our model. Sample applications are also described to bring out the salient features of our model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Dedication

For Rita

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I am deeply indebted to my advisor Kaliappa Ravindran for introducing me to the concept of distributed computing and for being a mentor to me every step of the way. His patient elaborations have made me erudite and his constant encouragement kept my spirit high. I also thank my committee members, professor Michael Anshel, professor Gwang Jung, professor Umit Uyar, Dr. Kevin Kwiat and professor Ted Brown for providing insightful comments and suggestions.

I am grateful to my parents for believing in my abilities, always expecting the best but never demanding it. I thank my father for his quiet support. I am grateful to my mother for wanting me to have a Ph.D. and making it known.

This work would not have been possible except for the encouragement of my wife, Rita. Her unwavering faith in me made me strive harder. I thank her for enduring this long process with me, always offering support and love.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Abstract iv
Dedication v
Acknowledgement vi
List of Tables x
List of Figures xi

1. Introduction 1
   1.1 Context of a Problem 4

2 Formulation of Research Problem 12
   2.1 Motivation 12
   2.2 Research Problem 14
   2.3 Solution 14
   2.4 Contribution 15

3. Evolution of Collaboration 17
   3.1 Synchrony and Interactivity 17
   3.2 Centralized vs. Distributed Implementation 18
   3.3 Incorporation of Audio and Video 20
   3.4 Application Domains 21
      3.4.1 Telemedicine 21
      3.4.2 Distant Learning 21
      3.4.3 Online Gaming 22
      3.4.4 Cyber Art 22

4. Characteristics of Coordination Problem 23
   4.1 Interactive Sharing 23
   4.2 Real-Time Responsiveness 24
   4.3 Dynamic Participation 24
   4.4 Spontaneity of Actions 26
      4.4.1 Pessimistic Concurrency Control 26
         4.4.1.1 Locking 26
         4.4.1.2 Transaction to Mechanisms 27
         4.4.1.3 Turn-Taking 28
         4.4.1.4 Centralized Controller 29
      4.4.2 Optimistic Concurrency Control 30
         4.4.2.1 Rollback 30

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Our Model: A State-Machine Based View of User Interactions 31
- 5.1 Action Compatibility 37
- 5.2 Formalizing Action Commitment 41
- 5.3 Collaboration Driven by Rules and Goals 42
  - 5.3.1 A Sample Application: Distributed Tiling 43

6. Mapping of Collaboration Model onto Existing Programming Frameworks 47
- 6.1 Logical Synchrony and Message-Level Asynchrony 47
- 6.2 Extracting Event Dependency 49
- 6.3 Atomic Actions Based Frameworks 52
- 6.4 Atomic Message Broadcasts 53
- 6.5 "Atomic Write" Based Realization 56

7. Proposed Solution: Temporal Epoch Based Programming Paradigm 59
- 7.1 Formulation of Epoch 60
- 7.2 Choice of Epoch Duration 62
- 7.3 Application- Specific Choice of Time Constraints 65
- 7.4 Summary 67

8. Programming Tools to Incorporate the Multi-Media as Part of Epochs 69
- 8.1 Occurs_After: A Declarative Notation 69
- 8.2 Dependency Graphs 71

9. Timed Distributed Agreement Protocol 75
- 9.1 Assumptions 75
- 9.2 Structure of Agreement Protocol 76
- 9.3 Delay Specification by Protocol 78
- 9.4 Protocol on Deterministic Delay Bounded Networks 80
- 9.5 Data and Control Channel Separation 88
- 9.6 Protocols With Probabilistic Delay Bounded Data Channels 91
- 9.7 Performance Evaluation of the Model 99

10. Deciding on Local Presentation 104
- 10.1 Rule Space for Generating Compatibility Relations Among Actions 104
- 10.2 Constructing Rule Base for a Sample Application 104
- 10.3 Mapping of Compatibility Relations Onto Action Execution 107
- 10.4 Algorithmic Procedures to Determine Executable Sequences 108

11. Sample Applications 115
- 11.1 Distributed Canvas 115

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

11.2 Distributed Gaming 117
11.3 Multiplayer Card Game 117
11.4 Multiplayer Shooting Game 120

12. Implementation 123
12.1 Structure of API 123
12.2 Private Window 125
12.3 GUI for Parameter Specification 125
12.4 Processing of Temporal Constraints 126
12.5 Media-Level Presentation Committed Actions 128
12.6 Session-Level Group Membership 130
12.7 Play-Out of Media Messages 130

12. Conclusions 132

Bibliography 139

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

Table 1. Collaboration classified in terms of synchrony & interactivity 18

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

Figure 1. User view of a sample collaborative session 4
Figure 2. Progress of collaboration in terms of epoch 9
Figure 3. State space for collaboration: each dot represents a possible global state $\in \{q_a, q_b, q_c, \ldots\ldots\}$ 33
Figure 4. State transition view of user interactions 33
Figure 5. An illustration of spontaneity in user interactions 35
Figure 5. Layers of function in our model of distributed collaboration 49
Figure 6. Additional functions required to use ARGUS for implementing our model 53
Figure 8. Additional functions needed to use ABCAST for implementing our model 54
Figure 9. Use of TAW to implement distributed collaboration 56
Figure 10. User oriented view of a communication epoch 61
Figure 11. Management-oriented view of application-protocol interface 67
Figure 12. Timing representation of $((z,p_z)$, Occurs_After$(y,I_z,u_z))$ 71
Figure 13. A sample dependency graph of sequential actions 72
Figure 14. A sample dependency graph of concurrent actions 72
Figure 15. The flow of timing information in the system 78
Figure 16. Illustration of delay behavior of message channel 81
Figure 17. Protocol scenario for deterministic delay bounded channel (case of all eligible participants generating action) 84
Figure 18. Protocol scenario for deterministic delay bounded channel (case of a subset of eligible participants generating action) 86
Figure 19. Data and control channel separation 90
Figure 20. Illustration of delay behavior of data and control channels 91
Figure 21. Protocol scenario for probabilistic delay bounded data channel 98
Figure 22. Performance evaluation graph 101
Figure 23. Example: layout of a building construction zone in a city 106
Figure 24. A sample run of our algorithm (example of city planning) 113
Figure 25. Schematic diagram of software model on a workstation 124
Figure 26. GUI for parameter specification 126
Figure 27. A message cluster 128

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1. Introduction

Distributed collaboration is an interactive joint effort among a group of people toward a common goal. With the availability of graphic workstations interconnected by high speed networks such as Ethernet and fiber optic networks, application scenarios in which a set of users interact with one another over a network in real-time to achieve a common task are becoming prevalent. Shared text editor, internet-lectures, and remote consultations are few examples of collaborative applications. Besides human-level collaborations, many applications are evolving that involve intelligent agent programs interacting with one another to collectively execute a task. An example is the multi-robot navigation of a work area, say, a room, using the positional and speed data from the sensors mounted on various robots for navigation planning and execution. In a hybrid scenario, applications involving the use of both humans and machines for collaborative decision-making are also evolving (e.g., Tele-immersion). Regardless of the type of application, the information flows between the collaborating entities have real-time constraints, and are often realized by using multimedia data (such as images, video and voice).

A salient feature of distributed collaboration that makes it stand out from other traditional distributed applications is the interactive sharing of a *context* that drives the task execution. The participants in a collaboration (also referred to as users) act upon a shared window of objects that embodies the context of a global task executed by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

them. Here, the action taken by a participant is often based on the other participant actions that have already taken place on the shared window. This is due to the fact that actions are generated with a common goal of solving the task in mind, that is to say, the actions correspond to user strategies towards achieving the goal. Here, each user activity can potentially impact the way other users formulate their activities during a task execution. Accordingly, users may need to re-evaluate the solution strategy at different stages of their interactions. Furthermore, the various task steps may not be known beforehand during execution, say, due to a large dimensionality of the problem being collectively solved (this precludes an automation of the various task steps).

Given the above characteristics of collaborative problem solving, the distributed execution of a task by remote collaboration may cause the participants' perception of task progress, as seen through a globally shared window, to diverge. The view divergence may arise due to the user-level asynchrony and system-level communication delays, compounded by the lack of physical shared memory between users. Consequently, users may possibly end up generating conflicting actions --- which may in turn lead to a lack of cohesiveness in the collaboration session. This is unlike a face-to-face collaboration among users where the shared context is physically made available to the users at a single location, often through visual cues and other means that can be easily seen and/or sensed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Our research is along the direction of developing a *distributed programming* model that facilitates collaboration among users over a network while mimicking a face-to-face collaboration. To achieve the effect of face-to-face collaboration in a network setting, the system needs to enforce WYSIWIS (What You See Is What I see). That is, all the users have the same view of the shared window at various points in time as they work through various steps in the global task, despite the absence of physical shared memory between them. We shall formalize the model of WYSIWIS in a different light, namely in terms of a *shared context.* We shall see how thinking in terms of 'context of a problem' allows us to detect potential conflict among spontaneously generated actions. Perceptive consistency [Boui'04] is then a natural outcome of our model.

We view a collaboration as a set of actions taking place on shared window objects, where the objects are replicated at various participating sites (but are logically shared). Consistent problem view is interpreted as having the object state and data same in all the sites at various points of interactions among users. Figure 1 gives the pictorial illustration of a collaboration session. The globally shared window is realized in terms of the local copies of object state and data maintained at the participant sites (shown as rectangle boxes). The figure also shows a 'private window' in each workstation (shown in oval). The latter can be thought of as a 'thinking area' for the user at that site to compose and ready his actions in response to the current shared view of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

objects[1]. The possible divergence in the contents of 'private windows' captures the user-level non-determinism in generating actions (say, different humans responding differently for the same input).

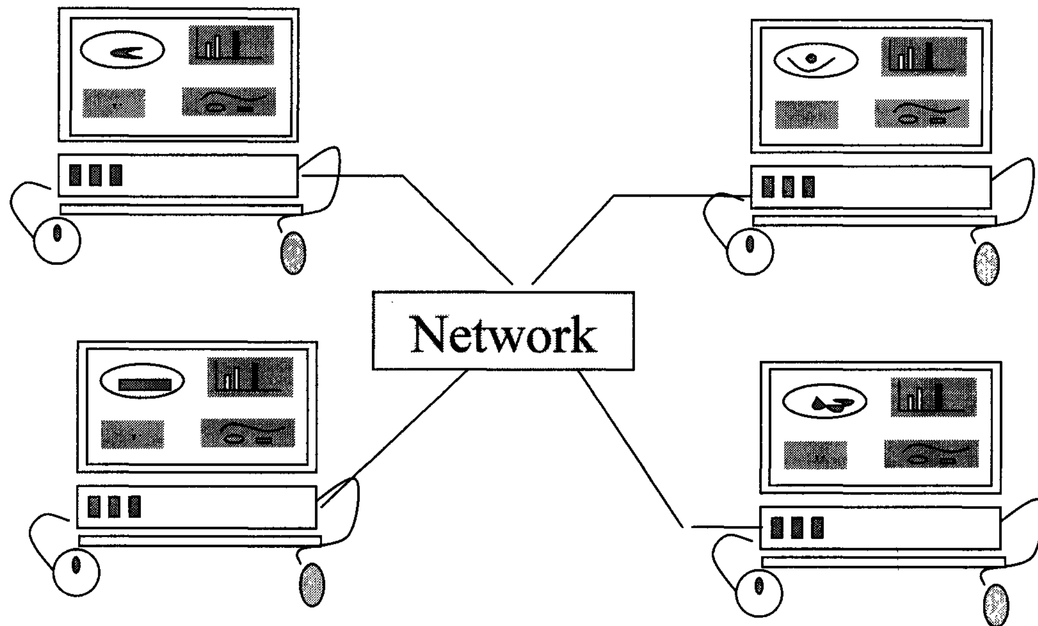


Figure 1: User view of a sample collaborative session

1.1 Context of A Problem

The context of a problem can be defined in terms of the object state and data through which users interact and the collaboration task itself (for the problem on hand) that is represented by the objects. For example, let us consider a scenario of collaborative tiling. We define the task goal (i.e., objective) as "putting tiles in an orderly manner

[1] Note that the 'private window' may or may not have a physical implementation at the participant sites.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(no overlapping) to cover a given space completely". Let us also assume the tiles come in different shapes. Now, at any point in time, a user action (putting a tile), will be dictated by his/her understanding of the task objective (namely, tiling the whole space without overlapping tiles) and also by the current shapes and positions of the tiles already placed. The combination of these two forms the context of the tiling problem. We understand that the context will keep in response to user actions, although the objective will remain unchanged. In the tiling example, the placement of tile by a user can change the strategy for future placement of tiles by the other users. When users are allowed to generate actions spontaneously and concurrently (i.e., they are not restricted by strict turn taking rules), it is possible that many users generate actions based on the same 'context of the problem' perceived at that time. For such scenarios, it is possible for any one of these (concurrent) actions to change the context in a way that the validity of other actions is called into question. This leads us to the concept of *action compatibility*, namely, how the generation of a user action in a given context can conflict with the generation of other user actions in reaching the overall task objective.

One of the main themes of our research is formalizing distributed collaborations in terms of the shared context of a problem. We shall see how the notion 'context of a problem' helps us identify user actions that are independent yet not compatible. Likewise, our notion accommodates multiple actions that are potentially non-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conflicting, i.e., are compatible with one another under certain execution sequences (even when the actions change the object state and/or data). By allowing compatible actions to take effect on the shared window (and aborting the other actions), we provide a generalized model of enforcing *contextual integrity* at various steps during a task execution.

The WYSIWIS requirement on shared window objects is necessary to maintain the session focus, because if a user sees a slightly different or out-of-date version of the object, the session's cohesiveness is lost [Elli'89]. We recognize, however, that there is scope for implementing a weaker form of WYSIWIS without violating the contextual integrity of a collaborative task execution. The reason being, it is possible to have independent compatible actions such that executing them in different order does not violate the perceptual consistency. We take advantage of such a weak implementation whenever possible to reap the benefit of the inherent parallelism at the user-level action processing and in the system-level object presentation control. This in turn can result in a more optimal use of the system resources: such as the communication network bandwidth and the processing cycles at user stations.

While adopting WYSIWIS as a basis for providing contextual integrity, our model allows multiple non-conflicting actions to commit (i.e., take effect on the shared windows) without violating the compatibility requirements. In this vein, existing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

works on distributed collaborations (such as [Sun'97]) realize a degenerate form of enforcing contextual integrity by allowing no more than one action commit at each task step. Accordingly, we claim that our model improves the cohesiveness of collaborations by minimizing the number of aborts at various stages of a task execution. This is a major contribution of our research effort.

From a distributed programming perspective, our model allows meeting the WYSIWIS requirement in the presence of users modifying the shared object asynchronously at various points in time. Here the notion of 'time points' refers to the occurrence of changes from one object state to another, as perceivable to the participating users, without violating contextual integrity. These 'time points' correspond to 'virtually instantaneous' actions on the objects, as perceived by users. Thus, our model provides a *virtually synchronous* execution of object-level actions (in the user-level view), despite the asynchrony in generation of user actions and the system-level communication delays. The granularity of these 'time points' is application-specific, with the window subsystem software allowing the users to only see consistent object states as the task progresses by virtue of execution of one or more user-level actions in a context-dependent manner[2]. For example, the 'time point' may

---

[2] The notion of 'virtual synchrony' as defined by K. Birman of Cornell University (in the earlier versions of ISIS system) [Birma'87] is at the level of individual messages exchanged between the processes of a distributed application. In contrast, our notion of 'time points' are coarse granular: first, it is action-based and not message-based, and second, more than one action commit can be accommodated in a single 'time point'.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

correspond to, say, 200 msec in a multiplayer shooting game, 100 msec in a distributed robot control application, and 10-15 sec in a 'designer team workbench' application. The thesis identifies suitable programming constructs to embody the above notion of virtually synchronous execution.

Based on the above formulation of research direction, the thesis embarks on identifying suitable distributed programming constructs for use by collaborative applications, and then on designing the underlying coordination control protocols needed to realize the constructs. The model allows the incorporation of *meta-rules* prescribing action compatibility as part of an application-supplied library, for use by the window subsystem. Sample applications are described to illustrate our model and the benefits accrued therein. The thesis also describes a prototype implementation of the model as a demonstration of the 'proof-of-concept'.

As part of modeling, the thesis provides a programming construct, *temporal epoch,* that integrates the passage of real-time and the action committals at various user sites. See Figure 2. The epoch construct is state-machine based in which the user-level action persistence over time is modeled as a state transition. The coordination control protocols determine the duration of an epoch by identifying the set of actions that can be committed in each round of user interactions, and then playing out the *messages* constituting these actions on the user windows (here, a message may carry multimedia

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

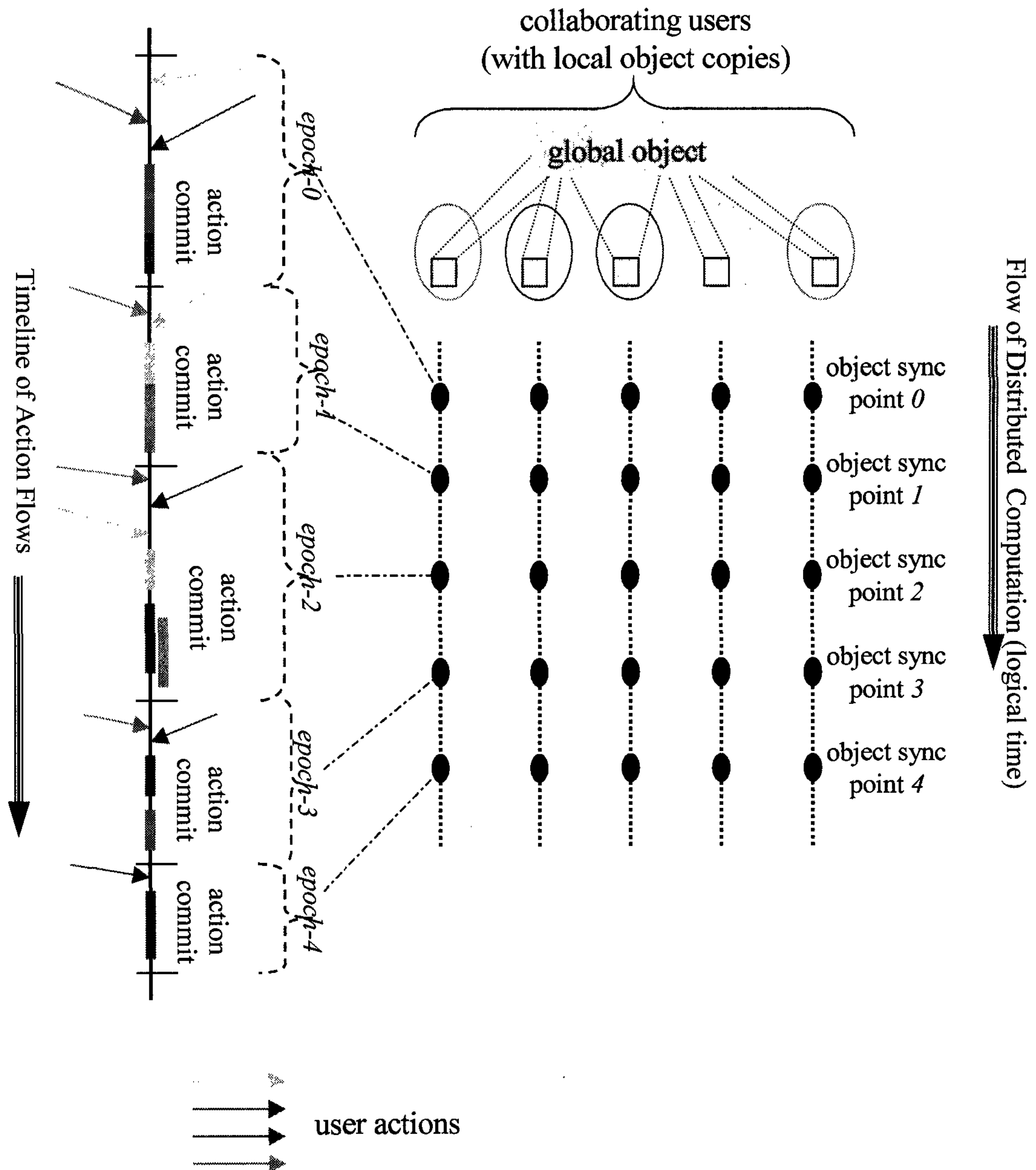


Figure 2: Progress of Collaboration in Terms of Epoch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

data such as video, images, and voice). That way, each user site advances its local instance of application-level clock by the same duration as every other user site, with the local copies object state and data updated identically at all the sites. Upon completion of object updates in an epoch, the window subsystem begins the next epoch by allowing users to generate the next round of actions.

Our generalized model of distributed collaborations in terms of temporal epochs (with the associated constructs and protocols) can be deemed as *context-aware.* Here, the objects in the shared windows prescribe a context for user-level actions in progressing towards a task goal. And, the application-specific rules governing the compatibility relationships between user-level actions allow the window subsystem to maximize the session cohesiveness, while the users strive to meet the task goal collaboratively. Thus, the window subsystem built using our model can adapt to the application domain through an appropriate prescription of the timing parameters, action semantics, and the rules governing the flow of actions. Overall, our epoch-based programming model is amenable for a *middleware-level* realization of distributed collaborations.

The remainder of the thesis is organized as follows. In chapter 2, we formalize the research problem and give a glimpse at our contributions. Chapter 3 is a brief history of the evolution of collaboration. Chapter 4 identifies the characteristics of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coordination problem. In chapter 5, we formally introduce our state base model of collaboration. Chapter 6 explores the existing techniques and required adaptation to use them in realization of our model. Chapter 7 introduces the temporal epoch based solution. Chapter 8 provides mechanisms that allow for the incorporation of multimedia as part of epoch. In chapter 9, we develop the distributed agreement protocol that enables the collaboration to progress in a synchronous and timely manner. Chapter 10 talks about rules that govern an application, inference of action compatibility from the rule base and generation of execution sequence for actions. In chapter 11, we provide formalization of some representative collaborative applications that can employ our framework. Chapter 12 is the description of a prototype application that has been implemented to prove the concept behind our research. And finally chapter 13 contains the concluding remarks and future direction in our research.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2. Formulation of Research Problem

The context sensitive WYSIWIS solution for a real-time environment differs from traditional distributed database synchronization in its dynamics. Any solution to the problem that limits the level of possible asynchrony, in the presence of contextual integrity, is not desirable. Consider for instance, requiring that an action be performed at exactly the same real-time at every participant (which subsumes a strict ordering of actions) thus guaranteeing WYSIWIS. Though possible with the use of centralized time-stamping schemes, the solution would not have allowed concurrent events to be executed by a workstation as soon as the message carrying these events are received from network. Accordingly, we need distributed concurrency control mechanisms that order the actions at various replicas of the object only to the extent necessary. This requires flexibility in specification of the object granularity available to various operations and temporal ordering relationships among operations on an object. The object semantics, in turn may impact the realization of the concurrency control protocol.

2.1 Motivation

There appears to be a difference of paradigm when we are talking about real-time user centric collaboration systems (the kind we are interested in) and systems like shared databases/files. As Ellis [Elli'91] points out, "the database systems strive to give each user the illusion of being system's only user, while groupware systems strive to make

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

each user's actions visible to others". Database systems are extensively researched and most of the concurrency control algorithms that are in practice explicitly or implicitly assume problems having characteristic properties of the latter class like serializability and/or revocability of actions. In a user centered collaboration the notion of revocability does not exist since human perception cannot be rewound to a previous state. In order to maintain WYSIWIS, we still need to be able to serialize user actions, but bearing in mind the potential violation of compatibility among independently generated actions. The notion of action compatibility has to be incorporated into the model of collaboration so that it is possible to agree on a set of actions rather than a single one. At the same time we need to develop a consistency model that allows asynchrony to exist to its fullest extent among compatible actions. For the real-time nature of collaboration, the capability of imposing real-time constraints on event scheduling becomes essential. We also need to develop a formal model for characterizing user-level actions in an environment of interactively sharing windows so that:

- The serialization protocol that underlies the window system can be employed across a variety of application scenarios involving window sharing.
- Analysis of user-level interactions is possible in an implementation-independent way.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Research Problem

The problem, simply stated, is to *provide a mechanism that guarantees contextual integrity among different entities in a collaboration session in a timely manner.*

The goal of this thesis is three fold:

i) We investigate the nature of real-time collaborations, identify the characteristic traits of such systems and come up with an abstract model to capture them in general.

ii) We identify the mechanisms needed to support real-time collaboration.

iii) We propose an architecture robust enough to incorporate the collaboration mechanisms in an application independent way.

2.3 Solution

Our solution to provide consistency is by guaranteeing a weaker form of WYSIWIS that can exploit asynchrony without violating contextual integrity. Realization of our solution has the following distinct steps:

i) We formalize a state transition model to capture user (human) actions in a collaborative session.

ii) We develop a distributed programming view suitable for conferencing applications.

iii) We introduce a transaction-like concept: 'message clusters', that can be used to bundle up messages representing user activity. We provide

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

constructs and implementation methods for realizing shared window systems in terms of message clusters.

iv) We introduce the concept of temporal epoch, unit of time over which collaboration takes place, which is used to implement the state transition seamlessly.

2.4 Contribution

We believe our research makes the following contributions:

- Our understanding of nature of *human oriented real-time collaboration* is enhanced. A comprehensive study of the existing techniques for solving problems related to CSCW (computer supported collaborative work) applications and groupware lead us to develop a state transition based formal framework. We formalize user interaction in the form of a finite state automaton.
- Existing literature defines a consistency model in terms of the following properties: Convergence, Causality-preservation and Intention-preservation (Sun'96). We recognize that even in the absence of causal or intention violation it is possible to violate the *context of intention*. We introduce a new concept '*context violation*'. We argue that when more than one user generate actions in response to some globally agreed upon context (i.e. the pre-condition of contextual integrity existed), it is possible that even

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

though these actions are causally unrelated (i.e. spontaneous), committal of one action will deem the other action unwarranted. In other words, execution of one action will lead to the *context violation* of the other action, deeming them incompatible. This concept of *compatibility* of actions is a novel one.

- We provide mechanism for specifying real-time constraints to be enforced on play out schedule of media streams. Thus our model has physical time awareness embedded in it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3. Evolution of Collaboration

Computer aided collaboration has evolved and has touched on many different disciplines: multimedia, distributed systems, networking and human interaction, to name a few. Accordingly, research on this field is accordingly varied and diverse. We can group the disciplines in terms of their architecture or application domain. A comprehensive study of the collaboration matrix can be found in the survey by Schooler [Scho'94]. Here we brief on the seminal ideas.

3.1 Synchrony and Interactivity

One of the most important distinguishing factors in collaboration is whether the collaboration is done over real-time or not. When users interact in a timely manner, e.g., carrying on a conversation over telephone, they are said to have a synchronous communication whereas when their interaction is done over a more relaxed time frame, e.g., email, they are considered to form an asynchronous communication. Examples of asynchronous conferencing (we note that conferencing is one form of collaborative computing) are structured messaging systems, e.g. Object Lens [Lai'88], multimedia electronic mail e.g. MIME [Bore'93], electronic bulletin board etc. Whereas examples of synchronous collaboration is multi user text editors such as GROVE [Elli'91], multi user draw editor [Hags'90], shared window system such as VConf [Lauw'90] and multi player network supported computer games. Classifying groupware, as a 2X2 matrix in terms of space and time was first proposed by Ellis

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(Elli'91). The fundamental factors for classifying remote collaboration are relatedness and time; which is represented neatly in the following table [Domm'97]

Time / Relatedness	Asynchronous	Synchronous
Non Interactive	Database Access	Simultaneous File/Resource Access
Interactive/Reactive	WWW, Email, Bulletin Boards	Conferencing, Application sharing

Table 1: Collaboration Classified in Terms of Synchrony & Interactivity

Our research interest lies in human oriented synchronous multimedia collaboration system.

3.2 Centralized vs. Distributed implementation

The underlying architectures for collaboration can differ in nature in terms of replication. In the un-replicated or centralized models we have the application maintained in a single site. Input is forwarded from site (often the site having the floor) wishing to change the state to the central site executing the application. Output, in turn, is broadcast to all participating sites. A fully replicated architecture on the other hand runs a copy of the application at each site. Input from the sites (possibly regulated by a floor control mechanism) is broadcast to all the participating sites and the output is generated locally.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The advantage of centralized approach over the decentralized one lies in the fact that there is no need of synchronization among the replicated (locally computed) output. Also, it is often easier to transform a single user application into one that is collaboration-transparent in a centralized architecture. The drawback, on the other hand, is that for a geographically distributed environment, communication delays may deem the approach unsuitable. For such scenarios centralized architecture may provide poor interactive response among the participants. In addition centralized architecture will produce more network traffic compared to a replicated one, since not only the input, but the output as well need to be distributed over the underlying network. In a LAN setting these disadvantages are not prominent due to low latency, but they are accentuated in wide area settings. Because of this, replicated strategy seems more suitable for WANs. A replicated strategy also makes sense due to the inherent redundancy in the architecture; it is less prone to problems associated with bottlenecks and single point of failure. Hybrid approaches attempt to take advantage of the best of the two worlds; for example, maintaining data consistency through a centralized data store, but supporting individualized view by graphical front ends has been proposed by Bentley [Bent'94].

Diverse communication environments, like the Internet, come with its own set of problems, irrespective of centralized, decentralized or hybrid architecture. Because of the variable delays or short-term service outage, as a possible consequence of routing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

failure, resynchronization may be necessary. In such scenarios a centralized architecture, having only one copy of data, may prove to be less taxing [Lauw'90].

3.3 Incorporation of Audio and Video

For remote conferencing, audio must be supported as a communication stream. Some earlier attempts to incorporate audio as part of collaboration involved coupling computers with telephones like Phoneshell [Schm'93]. More standard approach is to treat audio as data [Casn'92]. "A fundamental challenge of conversational audio in the packet realm is not only jitter free playback, but also the need to meet delay thresholds for interactivity [Scho'94]". Audio also is used as a control channel to resolve floor arbitration.

Just like audio, incorporation of video in a conference setting has become more and more prevalent. Breakthrough in image processing, discoveries in sophisticated compression and decompression techniques and at the same time advent of faster communication channel lead to the realization that video, like its audio counterpart, can be carried in digital form and transmitted across computer networks as a standard data type. Packet video conferencing systems have been developed for not only LANs [Vin'91] but also for WANs [Elli'93].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4 Application Domains

The concept of computer-aided collaboration, in the beginning, was closely tied to the concept of conferencing; later on conferencing became a tool for facilitating collaboration. Collaboration is being used in a diverse spectrum of disciplines like medicine, science, education, art, games, just to name a few.

3.4.1 Telemedicine

With the advent of telemedicine, physicians separated geographically can share their expertise in a diagnostic process. Collaborative radiology, neurology and surgery are becoming more common nowadays. System prototypes mix real-time tele-consultation with online medical records [Scho'94]. Telemedicine requires the ability to share high quality images in real-time, on the other hand, due to the privacy of medical information, sharing of information has to be secure.

3.4.2 Distant Learning

Due to the ubiquity of the Internet, many educational institutions now offer online courses. Students, instead of being required to physically be in a classroom, may participate via a web terminal. A flexible floor policy is required for dynamic student participation in discussions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4.3 Online Gaming

Network supported distributed Multi-player games is becoming ever more popular. High responsiveness with fine synchronization is usually demanded. Providing WYSIWIS in real-time is the challenge.

3.4.4 Cyber Art

A shared whiteboard kind of application is around for quite a while. Painters from different part of the world can take part in forming a collage. Preserving contextual and intentional integrity is important. Musicians and artists can perform together synchronously over the network in environments like cyber cafe. Overcoming scheduling conflicts in different time zones becomes a major problem [Scho'94].

In this chapter we gave a glimpse of the evolution of distributed systems. We also looked into the diverse application domains that fall under the conceptual umbrella of distributed collaboration. Next we investigate the problem of co-ordination among the collaborating entities, since irrespective of their domain and scope, all collaboration effort requires co-ordination and consequently employment of concurrency control mechanisms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4. Characteristics of Coordination Problem

Collaboration without coordination is lame. Any collaboration framework must provide coordination mechanisms. Collaboration performed over real-time imposes additional synchronization restrictions. First we identify the traits of the coordination problem and some techniques for solving them.

4.1 Interactive Sharing

A user activity may be based on one or more prior activities. For example, a user operation on a document during a joint-edit session may be triggered by the verbal comment of another user. The extent of interactivity among users in a session influences the program-level parallelism. Extracting event dependency is a challenging problem. There are two alternative models:

- Inferred Causality: Based on Lamport's [Lamp'78] notion of logical time, mechanisms like Vector Clock [Bald'02] have been developed that allow for the message level dependency to be inferred.
- Declarative Specs: Users explicitly specify dependency of generated actions [Ravi'93][Yava'91][Paza'94].

These models have their own set of advantages and disadvantages. Lamport's is more amenable for incorporation in a procedural paradigm (C, C++), whereas declarative approach requires a meta-layer to embody the specs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Real-Time Responsiveness

Real-time systems supporting these activities must not hinder the group's cadence [Elli'91]. High interactivity is usually achieved by requiring short response time and comparable notification times. User actions should take effect within prescribed real-time intervals at participant workstations to maintain the session cohesiveness [Gaje'95]. In a shared document editing example, it may be required to present three message segments a) cursor positioning, b) voice annotation and c) some icon removal from shared display, by a user u, in that sequence and say, over a 6 sec interval with a separation of 2 sec between each. Likewise, a user u′ may generate an action in response to u, say within 8 seconds, and this response action from u′ needs to be seen by all participants thereafter.

Thus from a modeling perspective, a temporal presentation involves extracting the timing relationship between various data segments collected at sources and determining the delivery times of these data segments for processing. There is no existing standard framework that allows for real-time awareness to exist in a collaboration setting. We will propose ours later.

4.3 Dynamic Participation

The inclusion of a new participant in a session (we define a session formally to be "a period of synchronous interactions supported by the collaboration system") may result

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in a reconfiguration of the system, say, by updating the session participant list in the display window of workstations. How the local object copy at the joining user gets initialized consistent with the global object state seen by current participants, needs to be determined as part of the session admission protocol. The protocol should ensure that any action on the object invoked by the new user be processed only in the context of updated session list. Likewise, users may abruptly leave the session, or may suspend their participation for a short period of time. How actions generated by participants prior to leaving a session are seen relative to the leave action itself, influences how other users perceive these actions. The protocol should ensure that the leave event be processed only after all actions generated by the leaving user take effect, else, other users perceive these actions as anonymous. Floor control mechanisms can be employed to maintain a cohesive group view. It is possible to implement a centralized floor controller or a distributed one. In a centralized version all requests for joining and leaving a group must be made through a central controller and the controller will grant permission in a timely manner (say, only in the beginning of a session). It also keeps all the participating entities up to date about the group membership. In a distributed implementation, we can run a multi-phase commit protocol on the decision to grant permission to join or leave a group. We assume, here, and for the rest of our discussion, that processes do not fail. Agreement in the presence of process failure is not our research interest.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4 Spontaneity of Actions

Users may generate actions spontaneously. Distributed systems need concurrency control mechanisms to resolve conflicts between participants' simultaneous operations. Extending existing techniques for resolving conflicts among users in a distributed database into the realm of real-time collaboration seems a natural choice. We investigate the most common techniques and their usability in the new setting. Classical concurrency control techniques fall into two categories: optimistic and pessimistic.

4.4.1 Pessimistic Concurrency Control

A pessimistic approach is to prevent conflicting events from happening in the first place. The advantage is that there is no conflict, hence nothing to recover from. On the other hand, looking for a potential conflict may take up a substantial amount of resources and dampen the collaboration spirit. Among the most commonly adopted methods are:

4.4.1.1 Locking

The oldest and most widely used concurrency control algorithm is locking. An object may be locked before it is modified (e.g., locking the graphical icon before deleting it). Deadlocks are usually avoided by performing two phase locking (first get the lock then lock). For collaborative settings, a display manager might visually indicate the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

locked resources. There are three major problems with locking [Elli'89]. First there is the overhead of requesting and obtaining a lock, this may include waiting for an object already locked (thus reducing the concurrency that is at the heart of a real-time application) which will have a degrading effect on response time. Secondly, there is the granularity problem. It is often very difficult to decide on the optimal level of granularity (i.e., exactly what needs to be locked? For example in a shared word processing exercise should we lock the word that the cursor points to, the sentence the word belongs to, or the whole paragraph?). Even if a judicious decision is made about using the hierarchical multiple granularity locking [Muns'96], the ensuing system overhead of obtaining the locks can be prohibitive and time consuming, thus making it a poor choice for real-time collaborative applications. Third, a problem stems from the determination of exactly when a lock should be requested and released. It is difficult to resolve the aforementioned issues satisfactorily for a broad class of collaborative problems, the solutions tend to work only for a special group of problems like shared files or databases.

4.4.1.2 Transaction Mechanisms

Sophisticated use of locks, in mechanisms like transactions, have allowed for successful concurrency control in non real-time systems such as Quilt [Lela'88]. In a real-time collaborative system however, implementing a transaction like mechanism can take a heavy toll on user responsiveness; also user actions are often mutually

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

exclusive thus they may not be serializable. For instance, when a set of users spontaneously try to update the document, it is quite possible that only one of the updates takes effect and all the others get aborted. Let us consider the following example of collaborative circuit designing: the premise being, a group of engineers, geographically separated, are working on development of a circuit. Say one of them wants to insert a capacitor whereas another wants to modify an existing resistor, it is possible that if the first action (insertion of capacitor) is committed it would deem the second action absolutely unnecessary -- if not detrimental to the design. That is to say that if the second person knew that somebody would suggest the capacitor in the first place, he/she would not have suggested the resistor change. Thus the two actions are not serializeable.

4.4.1.3 Turn-Taking

Turn taking mechanisms, such as intra-session floor control, can be viewed as a concurrency control technique. For a multi user setting it is particularly ill suited since it inhibits natural flow of information [Elli'91]. Turn taking can be employed for special applications like distributed card playing or other gaming scenarios where taking turns is part of the user behavior semantics. For scenarios where participants are allowed to spontaneously generate their actions and/or vie for shared resources, forcing them to take turn ruins the spirit of collaboration.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.1.4 Centralized Controller

The problem of synchronization could be resolved by having a central arbiter. By routing all user requests through the central entity, ordering of actions can be determined centrally; guaranteeing the same execution sequence at all sites and eliminating potential object discrepancy. Major weaknesses can be identified as follows [Diot'99]:

- Single point of failure.
- Scalability. There are two factors that would limit the scalability of such systems:
    - Since all data converge at the centralized server, the server becomes a bottleneck. Since a centralized server must collect all participants' data and forward them to all the participants, the processing speed of the central server can put an upper bound to the speed of data distribution. This is of a concern particularly in a real-time system.
    - With a server architecture, the amount of data transferred on the network is, in the best case, equal to the amount of data transferred with a distributed architecture. It is higher when native multicast is not being used.

Due to these problems a distributed implementation is desirable for time sensitive, responsive systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2 Optimistic Concurrency Control

An optimistic approach, on the other hand, works on the principle that conflicts are rare. Hence instead of trying to avoid them it tries to resolve them when it happens. The most commonly used method is rollback:

4.4.2.1 Rollback

In this mechanism, operations are executed immediately, but states and action causing the state transitions are retained so that at a latter point they can be undone, if necessary. Rollback is often employed in database systems. It is unsuitable in general for real-time systems especially for real-time collaboration. Particularly for human oriented collaboration rollback is not a viable option since human perception cannot be rolled back to an earlier state. Thus, feasibility of roll back is restricted to the communication layer. No presentation layer roll back is acceptable. Communication level roll back usually incurs substantial overhead since it requires the processes to keep track of all the possible intermediate states and also take care of garbage collection resulting from useless recovery information [Elno'96].

Having looked into the characteristic traits of the co-ordination problem and existing techniques for concurrency control, we put forth our model for capturing user interaction in a collaboration setting.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5. Our Model: A State-Machine Based View of User Interactions

Collaboration can be modeled in terms of objects that go through state transitions due to the execution of actions. Our ability to identify the distinct states an object goes through enables us to formalize a state transition model. User entities in collaborative applications may be modeled as interacting with one another to act upon the shared object. Generation of user actions can be thought of as in response to the 'state of shared object', enabling us to express actions as a function of the state. All the independent actions that are generated in response to a particular state of the shared object (global view of collaboration) can be grouped together, and can be thought of emanating over a single round of interaction, which we refer as epoch. The concept of epoch will be formalized later in chapter 7. Having characterized user collaboration in such a way helps us abstracting the whole collaboration effort in terms of a Finite State Machine (FSM).

Let *B* embody the object capturing a collaboration session. Then the state of the object at site *j* for a particular interaction round *i* may be noted by s_j^i. A consistent view of the object *B* at all sites can be maintained by making sure that

$s_j^i = s_k^i{}_{j,k=1,2,\ldots N} = s^i$, where *N* is the number of users in the session.

Then the global state of the collaboration session in this round: $O^i = s^i$

Now we can formalize the state transition model for a collaborative session in the form of a quintuple,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\mathbf{M = (}\boldsymbol{O,\Sigma,\delta,b,F}\mathbf{)}$$

Where,

$O = \{q_a, q_b, q_{c,}.........\}$ is a finite set of states

$\Sigma = \{a_j\}$ is a set of actions

$b \in O$ is the initial state

$F \subseteq O$ is the set of final states

And $\delta : O \times \Sigma \rightarrow O$.

The initial state O_0 for our model is an empty state.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

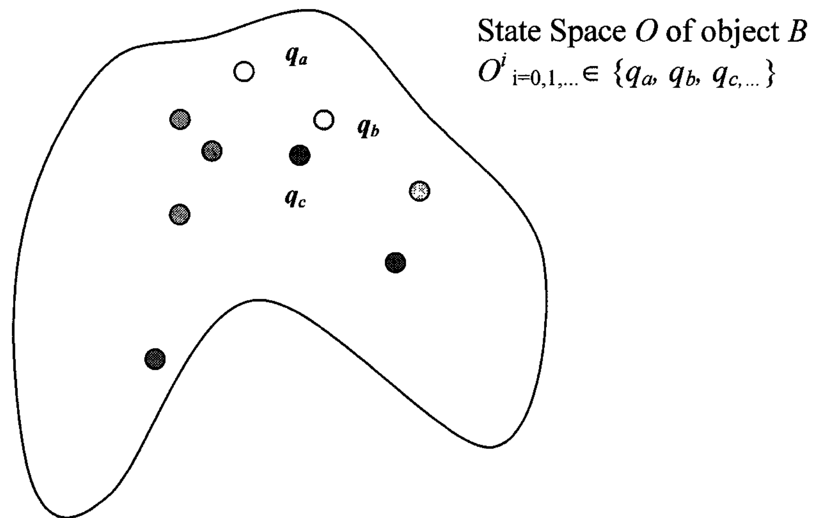


Figure 3: State Space for Collaboration: Each Dot Represents A Possible Global State $\in \{q_a, q_b, q_{c,..........}\}$

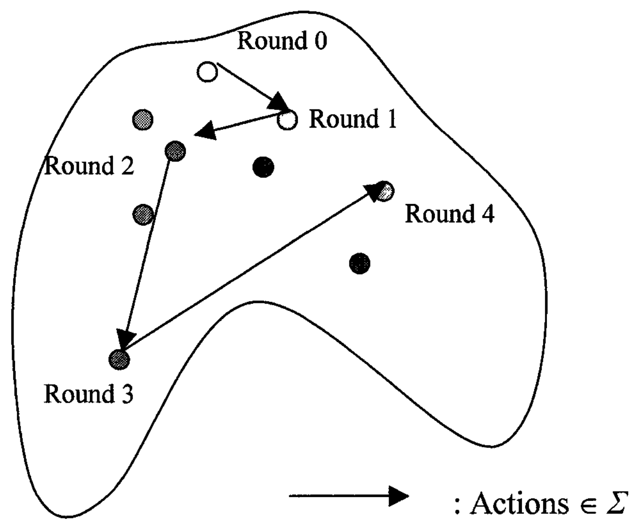


Figure 4: State Transition View of User Interactions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let g and f be functions denoting 'finite state automaton' relations that depict 'state transition' and 'action generation' respectively on a window object. In terms of g and f the state change occurring on window object at the end of a round e_j may be given as a functional relation on the object state O_{j-1} of previous round e_{j-1} and the set of user actions that update the window at the end of e_j, given as:

$$O_j = g\,(\{a_{i,j}'\}_{i \in \{1,\,2,\ldots k\}}\,,\, O_{j-1})\;\; {}_{j=1,2,\ldots} \qquad \ldots\ldots\;(1)$$

Where $a_{i,j}$ denotes the action generated by user i in round j and $a_{i,j}'$ denotes the committed actions at the end of e_j. The action generation by P_i for a given round j may be given as a functional relation on the object state on the window at the beginning of current round:

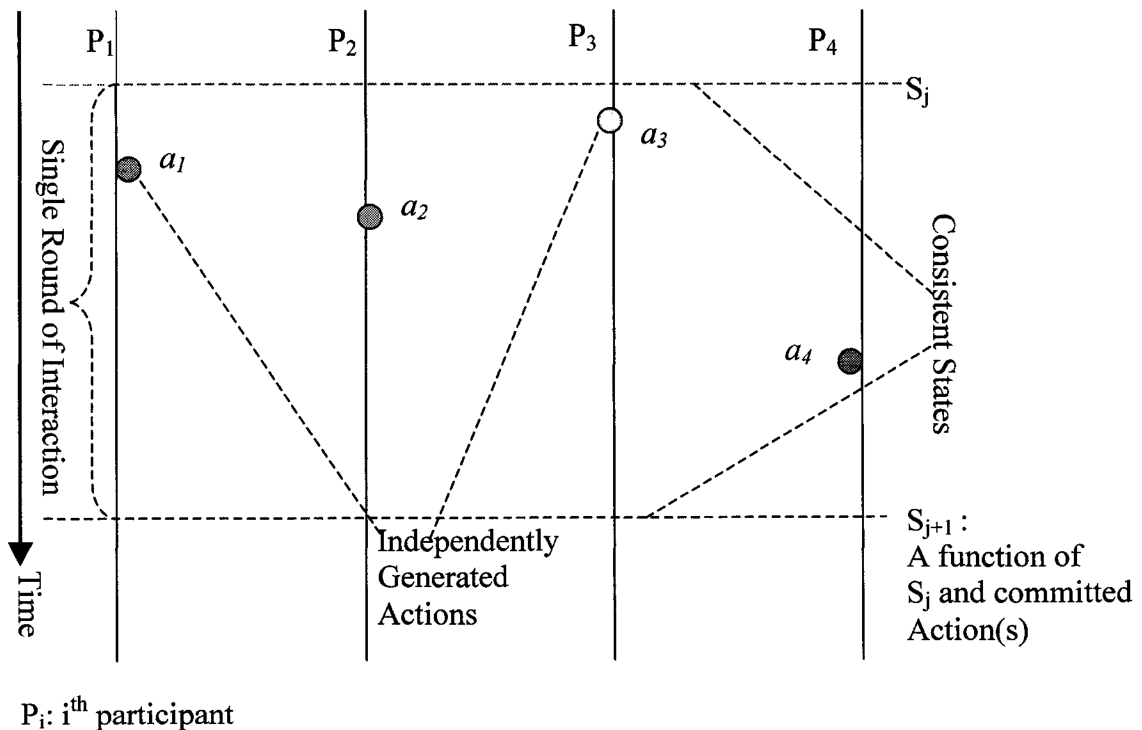
$$a_{i,j} = f\,(O_{j-1})\;\; {}_{j=1,\ldots,k.} \qquad \ldots\ldots\;(2)$$

The function g is the same for every participant, in order to maintain session cohesiveness. Having uniform g guarantees that all the participating entities change their view of global state deterministically and uniformly upon the execution of an action. Whereas, f can be different; capturing the potential diversity in the user action space. In other words, different users may generate different actions in response to the same state.

The relation (2) guarantees that actions at different sites are independent of each other, i.e., spontaneous. The actions initiated from multiple participants (for k>1), may, thus be viewed as potentially concurrent actions. Not all the concurrent actions may be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

compatible. A distributed agreement protocol commits k'' actions from a set of k' actions generated in a particular round ($1 \leq k'' \leq k' \leq N$) and aborts the rest, if any. The committal of an action involves presenting the media data units (messages) pertaining to the committed action on the media devices (window) configured to display it. The abort of an action involves discarding its component messages, without any effect on the user-level windows. The WYSIWIS requires that the various communication agents in a session reach agreement on set of actions to be committed and aborted. The novelty of our implementation is in enforcing a correspondence between the states of FSA and the rounds.



P_i: i^{th} participant

Figure 5: An Illustration of Spontaneity in User Interactions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

At this point we reiterate the fact that participants in our model will always respond to a state, not to other user actions. We made sure of it by taking the following step:

1. Global state remains the same as long as actions are generated.

From an implementation point of view, Step 1 is guaranteed by not playing out the events (executing the actions) as soon as they arrive. By making sure that during WOO (Window Of Opportunity), the period of time over which user actions are permitted to be generated, no actions are executed, we can keep the global state unchanged. In this light, our earlier work [Sabb'04] focused on a causality-oriented view of the collaboration problem.

Our model also provides a consistent view (WYSIWIS) at all sites by making sure the following steps are taken:

1. Global (the context of all actions) state is the same at all sites when actions get generated.
2. Same set of actions is committed at all sites over agreed upon time interval.
3. State transition happens deterministically and synchronously at all sites.

From an implementation perspective, step 1 is trivially granted in the beginning of a session, since everybody starts with, to use a metaphor, a blank slate. Equation 2 guarantees determinism by assuring the same actions induce the same state transition

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

at all sites. Thus global state remains to be true every step of the way provided we can guarantee the other two steps are taken.

Step 2 goal is achieved by running an agreement protocol.

Step 3 involves the ability to agree on the start of next round based on the duration and other timing constraints on committed actions.

5.1 Action Compatibility

By maintaining that actions emanating from different entities in a round are in response to the same global state we guarantee that they are independent, i.e., there is no causal dependency among them. Thus our model guarantees that:

Actions belonging to the same round are causally independent.

That brings us to the notion of compatibility of actions. Let us consider two actions that are generated in response to a globally agreed upon context (consistent view of the shared state), e.g., actions emanating from different entities in a given round. We argue that, it is possible for the execution of one of these actions (upon its committal) to change the shared state in such a way that the other action becomes unwarranted. We call this 'context of intention violation', ***context violation***, in short. Our notion of context violation is broader in its scope than intention violation defined elsewhere in the literature. Intention violation is defined in terms of intention preservation:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

"For any pair of operations O_a and O_b, if they are causally independent, the execution of O_a and O_b in whatever order must preserve the intention of each other."

Where intention of an operation is defined to be [Sun'96]:

"Given an operation *O*, the intention of operation *O* is an execution effect which could be achieved by applying *O* on the shared (document) state observed by the user at the time of issuing *O*."

Based on the above definitions, intention violation occurs when operations do not preserve 'intentions'. Intention integrity (absence of intention violation) is defined over a narrowly defined shared object state, which allows users to perform parallel operations over different objects without any dependency lookup. For example, in a shared draw-board kind of application, let us consider what happens if a user wants to add a tree into the drawing and another user wants to add a house. Intention violation will occur only if there is a violation of shared space, i.e., if both the users want to add their intended objects at the same place. By the very definition of intention preservation, an operation that involves adding a tree and the operation that involves adding a house produces the same end result (having a tree and a house) in whichever order they are performed, hence they do not lead to any intention violation. In reality, we argue, it is possible that, addition of the tree changes the context of the drawing in such a way that addition of the house is not warranted. In other words, had the user wanting to add a house knew a tree would be added to the drawing, he/she may very

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

well restrain from doing so. Our definition of shared state as the state of collaboration in its entirety allows us to deal with such context violation. In our FSM model every action changes the state, and actions are generated in response to states, that is, context of an action is the global state. Thus performing an operation (executing an action) in our model will always make us re-evaluate the other operations (actions) and their perform-ability (possible execution) in the renewed context. In light of these concerns we define compatibility of actions:

Two actions are compatible if they:

- Guarantee that execution of one will not change the shared state in such a way that the other's generation, in the renewed context, is not brought into doubt.

Compatible actions are such that they are all executable in the same round in which they are originated.

The above definition implicitly assumed the commutative property of compatibility since we did not mention the order of execution in our definition of compatibility of action. Since every action is going to make changes to the globally agreed upon state and thus change the context of collaboration it is quite possible that two actions *a* and *b* are compatible only if they are executed in a particular order. Referring back to our draw-board example it is conceivable that adding a tree does not influence the decision to add a house but not the other way around. Thus we formally define compatibility,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and denote it by the notation $\sim$ in the following manner, we say for two actions a and b:

$a \sim b$ (read as a is compatible with b) iff

- $a=f(q)$ and $b=f'(q)$, i.e. both actions were generated in the same context (Compatibility is not defined if contexts are different);
- $g(q, b) = q'$ and $a=f(q')$, a would have been generated even in the renewed context of b being executed.

Furthermore, an action c being compatible with the combined execution effect of b followed by a, denoted as $c \sim (a \sim b)$, implies the following:

- given that $c=f(q)$, $c=f(g(g(q,b),a))$.

We note the following properties are true for actions a, b and c:

- Non-commutativity: $a \sim b \mathrel{!\Rightarrow} b \sim a$
- Non-transitivity: $a \sim b$ and $b \sim c \mathrel{!\Rightarrow} a \sim c$
- "Happens Before": Given that actions a and b have occurred, and $a \sim b$ completely specifies the relationship between them, then

  $a \sim b \Rightarrow b \rightarrow a$, where $\rightarrow$ captures the "happens before" ordering relation as defined by Lamport [Lamp'78].
- Concurrency: Given that actions a and b have occurred,

  $a \sim b$ and $b \sim a \Rightarrow \| \{a,b\}$, where $\|$ indicates that the actions could have occurred in any sequence.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The asymmetric nature of action compatibility is captured in these properties. Furthermore, the properties allow mapping the compatibility relations between actions onto a temporal ordering of these actions at various sites during execution[3].

5.2 Formalizing Action Commitment

From an agreement protocol's point of view only compatible actions are committable. In this context we suggest using the following mechanism to decide which actions are committable at the end of a round. For the jth round we say an action a^i_j, is committable provided it satisfies the following condition:

$$\textbf{commit}(a^i_j) \Rightarrow [a^i_j = f(g(\{a^k_j\}_{k \neq i}, O_{j-1}))] \ \ldots\ldots\ldots \quad (3)$$

(Where, $\{a^k_j\}$ is the ordered set of already committed actions and f and g are functions defined in equations (1) and (2))

This means that an action a^i_j commits only if the decision to initiate the action is independent of all the other actions that have already been committed on the same, shared state. It is a powerful notation that has the capability of capturing the concurrency of independent events; since we do not require that the object state to remain the same for all the concurrently committed actions, we merely say that the state changes in such a way that it does not alter the user response to it. Consequently

[3] This is unlike Lamport's ordering of events based on logical clocks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we say that if the object state changes in such a way that a user might have responded differently then we discard that user's action.

Although equation (3) captures the notion of commitment, implementing the notion the way it is defined, is, if not impossible, computationally challenging. This equation would require the agreement protocol entity to know the f's of all the entities (however, due to determinism, g would be the same at all sites). Often f is not provided by the application in a closed form to begin with. Particularly in a collaboration setting, the ability to enumerate all possible scenarios and predict actions will be constrained by the dimensionality of the problem. The ensuing spontaneity leads to non-determinism (in terms of action generation), which implies our inability to compose f in a closed form.

5.3 Collaboration Driven by Rules and Goals

A more practical approach for determination of commitment of actions may be in terms of achieving a problem specific goal. If we define a goal for our collaboration, then it is possible to come up with a set of rules that help the participants collectively reach that goal. Actions will then be deemed committable if they do not violate these rules.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An application provides a list of objects (audio clip, video clip, draw clip, text message etc.), where an object is defined to be anything upon which an action can be defined. It also provides a set of actions defined on these objects (display, move etc.) and a set of rules in terms of objects and actions (e.g., do not display more than one audio clip at a time).

5.3.1 A Sample Application: Distributed Tiling

To clarify our ideas, let us consider a sample collaboration scenario of distributed tiling. The problem is to tile a given space with fixed size, but different shaped tiles.

To keep things simple (still interesting) let us assume there are two kinds of tiles: triangular and square. (Sizes of the tiles with respect to the space to be tiled are shown in the diagrams that follow). Let us also assume that, there is an infinite supply of tiles; i.e., a user will always have, at his/her disposal, any tile he/she wants to use.

Actions are placing tiles within a given boundary.

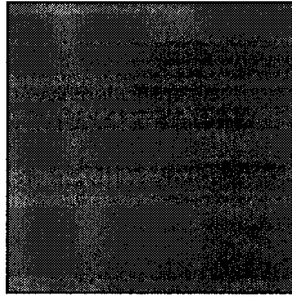
We can define the goal as:

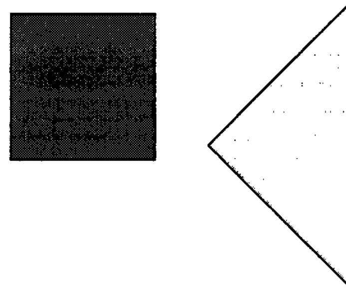
- The whole space covered with tiles with no overlapping.

We will come up with the rules as we go.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

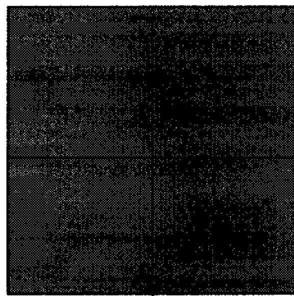
Scenario 1:

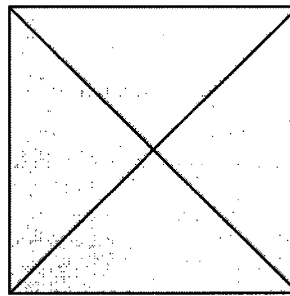




Space to be tiled Tiles

We first list the possible ways of tiling:

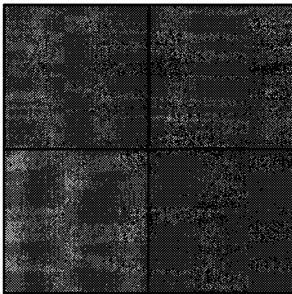


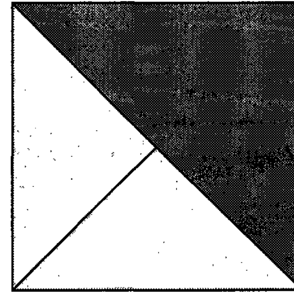


One possible rule for this scenario may be not to allow tiles of different shapes to be used. This rule will deem two actions, where one is the placement of a triangular tile and the other is the placement of a square tile, as incompatible hence not committable at the same time. Thus stopping from the following from happening:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

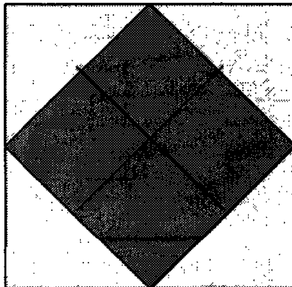
Which would have prevented us from achieving the goal. (We note at this point that intention preservation schemes would have considered the actions compatible and would not have been able to avoid the above situation), but would allow the following scenarios:





Scenario 2:

We see that the solution space for this collaboration problem can grow rapidly when we change the dimension of the space to be tiled, thus our simple rule will end up restricting many possible valid combination of actions from taking place simultaneously, one such scenario being:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is always possible to come up with more sophisticated rules that allow us to explore a greater portion of the solution space. But even simple rules allow us to exploit parallelism without compromising the goal.

For a collaboration setting where the goal is not well defined (e.g., distributed draw-board), and consequently, there is a lack of a well-defined algorithm for achieving that goal, we can be conservative and declare all actions to be mutually incompatible, thus restricting the commitment to no more than one action.

In this section we have described our framework in terms of compatibility and committability. Now we need to transcribe this collaboration model onto a programming framework for a distributed implementation, as described in the next chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6. Mapping of Collaboration Model onto Existing Programming Frameworks

Having formalized collaboration as a state machine model we investigate the scope of existing techniques in providing an environment with contextual integrity. First we note that due to the notion of action compatibility it is not possible to provide contextual integrity in the absence of some functionality that checks for the compatibility of spontaneous actions. With this constraint in mind, we shall examine event-based frameworks and atomic action based frameworks.

6.1 Logical Synchrony and Message-Level Asynchrony

Logical synchrony refers to the user-level view embodied in the lockstep progression of collaboration activities at various sites. It is a logical property of the system rather than the physical timing of the various activities. This is a powerful abstraction from a distributed programming standpoint since the user level view is that, each collaboration step appears to take place instantaneously. The logical synchrony is incorporated into the correctness assertion of a distributed algorithm that realizes collaboration among various sites.

For correctness reasons, an algorithm requires a maximum wait of the participants to accommodate the variability in user action generation time and the randomness in network delays. However, from a performance standpoint, its normal case of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operations often completes much sooner at a site than the required maximum waiting time, thereby allowing the site to initiate the subsequent steps in the algorithm. This is because the delays are more concentrated toward the lower end of the distribution. This manifests as a high degree of asynchrony in the presentation of actions at various sites, while preserving the logical synchrony in the user level view of the presentation. Thus a main goal of the distributed algorithm is to preserve logical synchrony while maximizing the latency-oriented performance. Our design of collaboration protocols meets this goal.

For algorithmic realization, we assume that the set of participants in the collaboration session is static and known beforehand. This assumption is merely to simplify the description of our model without restricting the scope of our model.

Figure 6 illustrates the architectural relationship between collaboration functions and the programming support mechanisms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

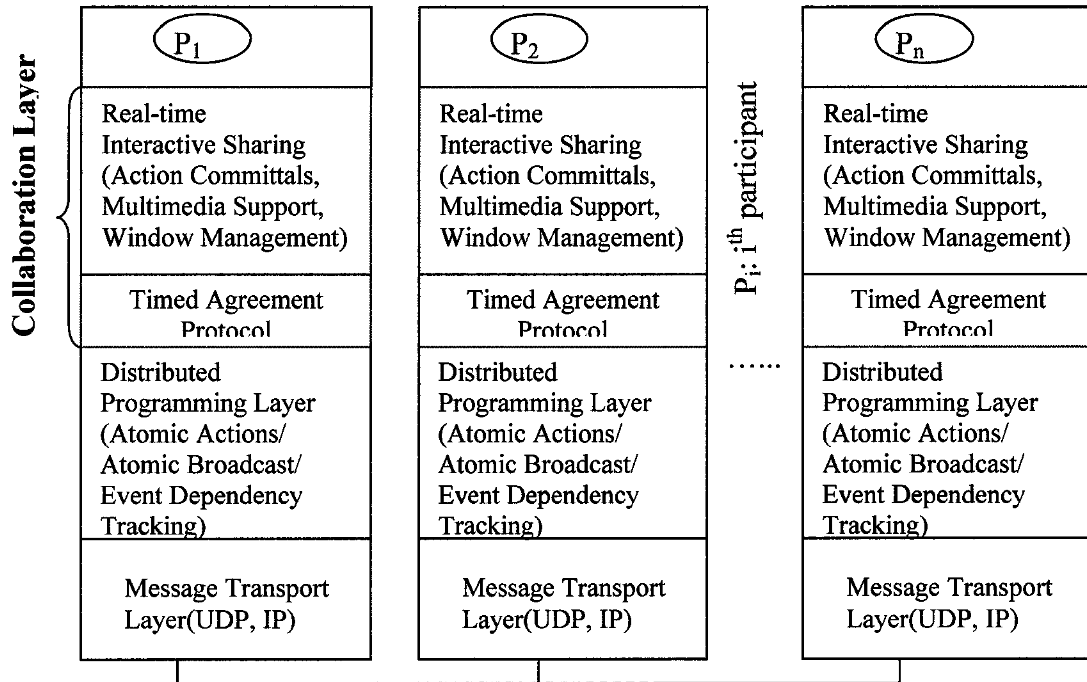


FIGURE 6: Layers of Function in Our Model of Distributed Collaboration

6.2 Extracting Event Dependency

The approach overwhelmingly used for concurrency control and synchronization in a collaborative setting is, avoiding conflict by detecting dependency among the actions. If generation of an action is influenced by another action (directly or indirectly), we say the first action to be causally dependent on the second. Figuring out causal dependency among actions emanating from sites in a distributed setting can be challenging. Lamport, in his seminal paper, "Time, Clocks and the Ordering of Events in a Distributed System" [Lamp'78], proposed the concept of inferred causality. Since

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

then, a whole consistency model has been developed based on Lamport's causality. According to Lamport:

Definition 1: Causal ordering relation "$\rightarrow$"

Given two operations O_a and O_b, generated at sites i and j respectively, then $O_a \rightarrow O_b$, *iff:*

1. $i = j$ and the generation of O_b happened before the generation of O_a. Or
2. $i \neq j$ and the execution of O_b at i happened before the generation of O_a.
3. Or, there exists an operation O_c such that $O_a \rightarrow O_c$ and $O_c \rightarrow O_b$

Definition 2: Dependent and Independent Operations

Given any two operations O_a and O_b

1. O_b is dependent on O_a *iff* $O_a \rightarrow O_b$
2. O_a and O_b are independent, i.e., concurrent, expressed as $O_a \parallel O_b$ *iff* neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$

Distributed systems researchers often use Lamport's causality notation '$\rightarrow$' to mean causality in general.

- Lamport's notion of causality: Since Lamport's is an inferred causality mechanism. It is possible for two concurrent actions a and b to be inferred as causally dependent. Thus possibly declaring $a \rightarrow b$. Since compatibility is defined only on concurrent actions, b escapes compatibility checks and its execution is automatically carried out once a is executed. Since these two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

actions are in fact concurrent, it is possible that *b*!~*a*. Thus the use of Lamport's inferred causality leads to a potential *context violation*.

- Declarative Specs for Causality: In this framework, dependency among actions is explicitly specified. So, if two actions are declared to be dependent, they indeed are. One such notation, Occurs_After, is discussed in fair amount of detail in section 9.1. This framework avoids misrepresentation of concurrent actions as causally dependent, thereby eliminating the potential correctness violation that otherwise arises due to inferred causality. However, we have to guarantee that compatibility checks are done on the same set of actions at every site. This naturally leads to the implementation of an agreement protocol similar to ours. However, the declarative specs require maintaining extensive communication states and their garbage collection at run-time.

Thus we see that, although event dependency offers an elegant programming framework, it is not directly suitable for implementing our collaboration model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.3 Atomic Actions Based Frameworks

Programming frameworks like ARGUS were developed based on a distributed atomic action primitive that embodies a transaction style grouping of user actions. The two main assumptions of such a model are [Lisk'88]:

- Actions are serializable: The effect of running a group of actions is the same as if they were run sequentially in some order.
- Actions are total: An action either completes entirely or it is guaranteed to have no visible effect.

In our model, user actions are spontaneous and are not known beforehand. Thus, forming a distributed transaction becomes difficult since we do not know the boundary of the transaction. Also, in our model, the assumption about the serializability of actions does not hold true in general. The compatibility check among actions will restrict the scope of serializability. Referring to figure 4, suppose the ARGUS system commits actions $<a_1, a_2, \ldots, a_k>$ in that sequence. This sequence cannot be presented to the user level window due to potential compatibility violations. So we need a sub-layer in the window system that performs a committability check on $<a_1, a_2, \ldots, a_k>$ before determining their presentation to the user level window (Figure 7).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

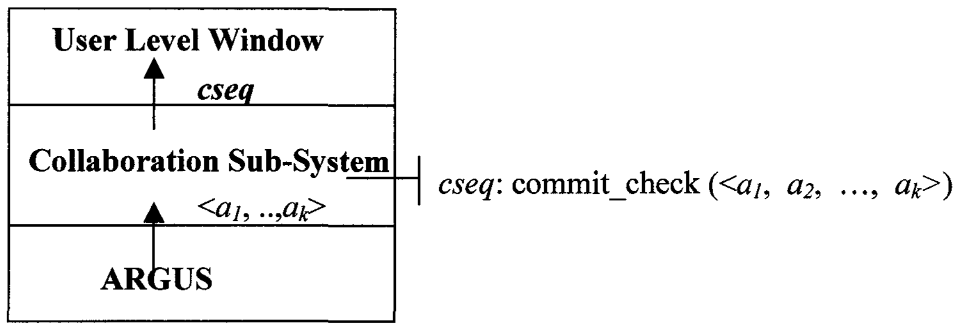


Figure 7: Additional Functions Required to Use ARGUS for Implementing Our Model

Furthermore in ARGUS, synchronization is done through specially created atomic objects and synchronization for such atomic objects is done by means of locks[4]. As mentioned earlier (section 4.4.1.1), the coarse-grained locking provided by ARGUS does not allow exploiting the fine-grained concurrency among multi-media window objects.

Thus existing action based frameworks are less amenable for use in our setting. Now we examine the use of atomic broadcast as a programming paradigm for our model.

6.4 Atomic Message Broadcasts

Based on Lamport's logical clock, Birman [Birm'87], as part of the ISIS system, proposed two communication constructs for group communication:

[4] In collaborative application setting all actions on the windows are treated as write operations in ARGUS system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a) Causally Ordered Broadcast (CBCAST) and

b) Atomic Broadcast (ABCAST).

ABCAST delivers messages atomically and in the same order at all sites. CBCAST, on the other hand, is less restrictive in terms of message delivery; messages that are causally independent (as defined earlier) can potentially be delivered in different order.

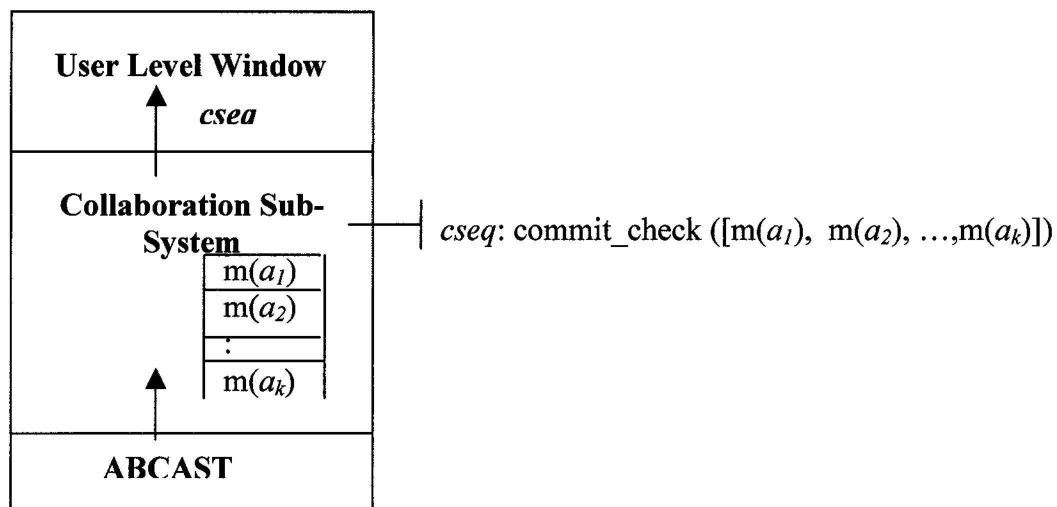


Figure 8: Additional Functions Needed to Use ABCAST for Implementing Our Model

As we have already pointed out, CBCAST is not useful due to the potential violation of context ensuing from inferred causality. ABCAST, on the other hand, totally orders the spontaneously generated messages in the current round of interaction. Typically, the messages carrying the actions are placed in a totally ordered message queue at every site, with the upper layer functions (deterministically) transforming the delivery

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequence onto an application-specific message selection[5]. In our case, the collaboration sub-system may possibly run compatibility checks on the sequence of messages [m(a_1), m(a_2), ...,m(a_k)] to determine the committable sequence *cseq*. Since total ordering guarantees that the message sequence is identical at every site, an agreement protocol that operates on this sequence can determine an identical *cseq* at every site for presentation to user level window (Figure 8).

We note, however, that for the employment of an agreement protocol to determine a committable ordered set, although each site has to have the same set of messages, they do not have to be stacked up in the queue in the same order. Since an agreement protocol is going to invoke compatibility look up functionalities, that in turn is going to determine the set of compatible messages and their order of presentation, any effort that goes into ordering causally independent messages before delivering them to the agreement protocol is wasted. Thus the problem in employing ABCAST in our model is not its correctness, but its performance. In other words, ABCAST is an overkill; putting unnecessary strain on the network layer in demanding the same delivery sequence for messages.

---

[5] For instance, Lamport's mutual exclusion algorithm selects the first message in a totally ordered queue of request messages to grant mutually exclusive access to a shared resource [Abad'94]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.5 "Atomic Write" Based Realization

We explored an alternate programming paradigm, *timed atomic write* (TAW), which implements a weaker form of ordering the actions when compared to the ABCAST [Ravi'04]. The TAW primitive allows group communication among users to realize 'data writes' on a shared buffer under timing constraints, and determine an unordered set of the 'data writes' to be agreed upon in each round of user interactions. Where the application-specific ordering of 'data writes' is not expressible in a closed form, the ordering is deeply buried into the application-level computations that reside on top of the TAW layer. The TAW-level decision on the set of writes is oblivious of the application semantics. Thereupon, the TAW requires employing filtering rules on this set of 'data writes' for application-specific processing --- such as enforcing an application-level ordering of the writes.

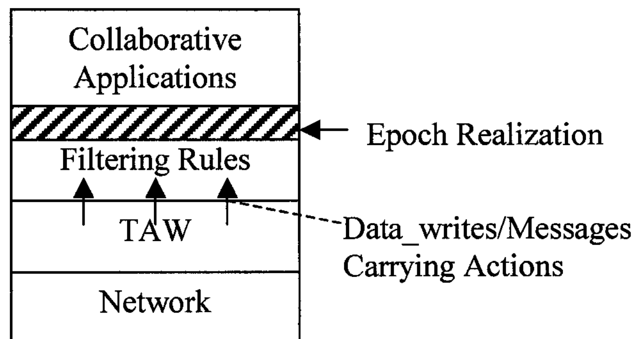


Figure 9: Use of TAW to Implement Distributed Collaboration

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The TAW primitive caters to a general class of 'data fusion' applications, with distributed collaboration being one of them. For the collaborative application settings considered in this thesis, the 'data writes' correspond to user actions, with the filtering rules prescribing the compatibility checks on the actions for final committal to the user-level windows. Here, the notion of epoch is deeply buried into the application-level processing on the actions.

The use of TAW primitive in distributed collaborative applications where the compatibility relationships among user actions do not have a closed-form representation has two ramifications. First, committing no more than one action will solve the ordering problem, albeit, in a degenerate manner. This is because committing exactly one action enforces contextual integrity without any need for compatibility checks --- as is the case with existing models. Second, an ordering of the actions based on a user-assisted resolution of conflicts among candidate user actions (say, through a GUI)[6] may still allow committing more than one action in an epoch. In either of the cases described above, the use of TAW will be as effective as the generalized epoch-based implementation described in this thesis.

Though TAW is a useful primitive, we believe that a generalized representation of epochs, which can be explicitly captured with application-supplied parameters, will

[6] The GUI structure to allow user involvement in conflict resolutions may require the window system to expose each user action to the other competing users and query about potential conflicts vis-a vis their actions --- say, with a Yes/No button in the GUI for a candidate action being queried about.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

offer more programming flexibility. Furthermore, a middleware realization of epochs can allow a re-use of the window system software across different application domains.

Having determined that the currently available programming paradigms are not flexible enough to accommodate our model of collaboration we embark on describing our solution to the collaboration problem: both at the programming level and protocol level, in the remaining part of the thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7. Proposed Solution: Temporal Epoch Based Programming Paradigm

The necessity to group together actions that share the same context follows naturally from our model. The notion of compatibility is relevant only on actions that share the same context. When two action-generating participants have different views of the problem space, there is not much point talking about their compatibility. Thus before we can check for compatibility of actions, we have to make sure that they were generated in response to the same agreed upon global state (context). Now, since execution of action is going to change the context, any action generated in the renewed context should not compete for execution with actions that were generated in response to the previous context. Hence, before we let the participants respond to a state, we have to make sure that this state reflects the execution effect of all compatible actions which were generated in the previous round of interaction. In an effort to group the actions that share the same context (and consequently are spontaneous), we introduce the notion of *epoch*. Informally an epoch is the period of time such that all the actions disseminated in this period are guaranteed to share the same context. From a programming standpoint, an epoch appears as a single 'time point' along the logical time axis[7].

---

[7] Epochs, in a sense, quantize the continuous flow of time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

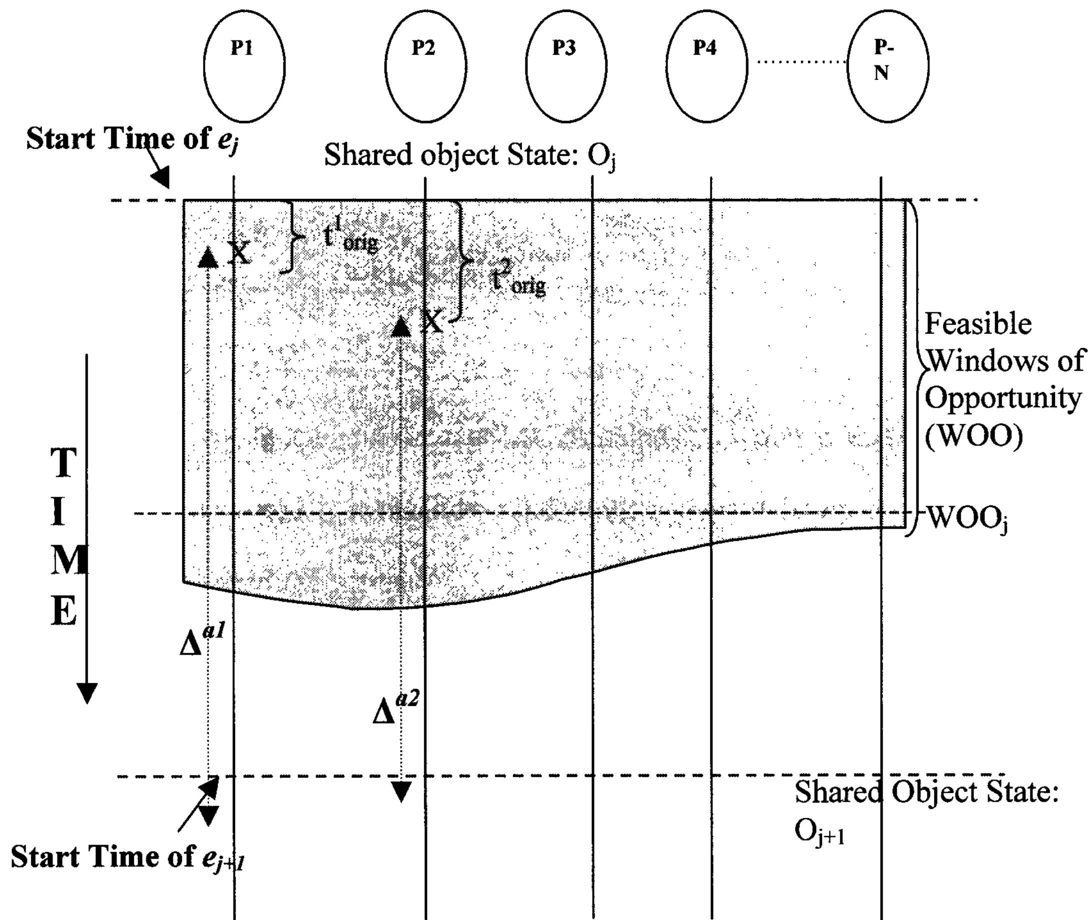
7.1 Formalization of Epoch

We can formalize the model of user-level interactions, as seen by application programmers. The model is based on determining the spontaneity in the generation of user-level actions, and how an ordering of these actions cause consistent state changes in the shared window. Our proposed construct, *epoch*, is the time interval over which the user actions emanating from one or more sites get generated and presented on the shared window for dissemination. A user may generate an action at the beginning of an epoch, as allowed by the application. Here, an action may very well contain more than one sub-actions as in multi-media based collaborations (e.g., video, audio annotations). The collection of such actions when committed defines what constitutes an epoch.

Let $\{P_i\}_{i=1,2,.....N}$ be the set of participants in a collaboration session. During an epoch, k of the participants $\{P_1, P_2, \ldots\ldots, P_k\}$, are eligible to initiate actions on the shared window, as allowed by the floor control policy, where $1 \leq k \leq N$ (Figure 10). Let Δ be the time duration over which the action generated by a user remains relevant. Every site has the mechanism to deterministically compute the beginning of an epoch, and the local copy of the globally shared window objects are updated at the end of each epoch guaranteeing a consistent view of the objects. When an epoch completes, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

necessary actions get committed at all user sites[8]. Upon a commit decision, the presentation of actions is realized by scheduling a timed-delivery of all the component messages carrying these actions.



X: Generation of user-level action Δ^{ai}: relevance time of action *ai* generated by P_i

Figure 10: User Oriented View of a Communication Epoch

---

[8] When none of the *k* actions commit, we view as if the epoch has progressed by executing a NULL action.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It may be noted that the maximum allowed time skew in data generation by users is given by: $[WOO - t_{orig}^{1}]$. This notion of time skew can be employed by the presentation protocol to reap the benefits of asynchrony inherent in the data generation activities.

7.2 Choice of Epoch Duration

For reasons of interactive sharing and/or fast interaction, the serialization protocol may commit only a subset of user actions at the end of e_j, with the remaining actions aborted. This allows all the users to 'slide' to the next epoch e_{j+1} in a renewed context provided by the updated state O_j. In real-time interactive games for instance, $(t_{j+1} - t_j)$ should be within the human limits of persistence time intervals (say, <50 msec). For slow mode interactive applications however, such as 'circuit design' by engineers, the interval $(t_{j+1} - t_j)$ can be of the order of seconds. So, users whose actions got aborted in e_j, will have the opportunity to modify their actions with respect to the renewed contexts in subsequent epochs (subject to the floor control enforced in those epochs).

Given the inherent asynchrony in user level action generation, we need to limit the skew in action generation times in a way that an epoch completes within the application perceivable latency. For this purpose, we introduce a notion of *window of opportunity* (WOO), which is the time interval over which each eligible user is allowed to come up with his/her action. WOO should account for the 'thinking time'

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(time required to come up with a response mentally) and the actual time to generate the action in a procedural form. At the same time, WOO should have the flexibility to accommodate for the variation in user responsiveness. The session manager (implemented centrally or in a distributed manner) is equipped with the application specific knowledge to prescribe WOOs. And these WOOs may be different in various epochs. Based on the WOOs, the session manager can determine the *maximum allowable latency*, T_{max_lat}, as:

$$T_{max_lat} < [WOO - t_{orig}^{\,1}],$$

which can then used by every site to determine the epoch duration. For network-supported computer games where fast responsiveness is required (such as Final Fantasy [Li'04]), we can have a short WOO. Whereas, for collaboration efforts of engineers, WOO can be comparatively large, since it is more important to come up with thoughtful suggestions rather than proposing the first thing that comes to mind. The fact that the transition from one epoch to the next involves the execution of at least one action allows us to model the discreteness of medium[9].

Ability to determine the beginning of an epoch deterministically becomes crucial for maintaining session cohesiveness. Computation of epoch duration has to take into account the times spent in the following steps:

- Collection of actions at various sites;

---

[9] In a discrete medium, the state changes in objects are event driven, rather than induced by mere passage of time. Whereas in a continuous medium, state changes may be induced by both events and passage of time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Execution of agreement protocol to determine the set of actions considered for commit;
- Determination of committable actions;
- Determination of the delivery schedule of component actions.

We can use the following formula for computation of the start time of next epoch (a detailed rationale for the formula will be given later in section entitled 'Agreement Protocol').

(Assuming we are currently in j^{th} epoch)

$$\mathbf{t_{j+1} = t_j + \min\{t^i_{orig}\}_{i=1,2\ldots} + T_{max_lat} + T_{dap} + T_{pres}} \qquad \ldots..(4)$$

Where,

$\{t^c_{orig}\}$: Origination times of the set of committed actions (committal of an action will be talked about in detail later)

t_j : Start time of jth epoch.

t^i_{orig} : Origination time of action emanating from user *i*.

T_{max_lat} : Maximum allowable latency in generation of response.

T_{dap} : Time spent by distributed agreement protocol to decide on a set of actions.

$T_{pres:}$ Cumulative presentation time of all the actions.

Note the set of actions committed are such that their relevance time Δ have not yet expired.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A word of clarification about epochs is in place here. It is not that the epoch duration is fixed beforehand and the action committals are then determined to fall within this time duration. Instead, the Δ, k, and D_Q parameters influence the epoch duration that is dynamically determined by the coordination protocol, which is in consonance with the time granularity meaningful to the application. The epochs may be viewed as imaginary lines along the time axis, with the spacing between successive lines depicting the distinctly perceivable logical clock ticks in the application.

7.3 Application-Specific Choice of Time Constraints

Our model of collaboration seamlessly accommodates different levels of timing constraints (Δ) on the actions generated. Based on the value of Δ, the epoch duration and the network delay parameters (D_Q in the deterministic case and the D_{Q1} and D_{Q2} in the probabilistic case) are appropriately chosen by the protocol. In applications dealing with emergency situations where the users interact among themselves under rigid time constraints (such as interactions among air traffic controllers or hazard management team members), the parameter Δ is set to a small value specific to the problem domain. In such hard real-time applications, the actions have to be committed within a rigidly specified bound Δ. However, in soft real-time applications, an action may still have some value even after the expiry of its Δ. In the latter cases, the application will prescribe a deadline larger than Δ, for use by the underlying

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

protocol. Regardless of the problem domain, our protocol enforces a deadline parameter supplied by the application (through an appropriate programming interface).

We provide below some representative parameter values for a hard real-time application where air traffic controllers are trying to schedule many flights through a given airspace over a certain time interval. We assume that the interactions among the traffic controllers take place through point-and-click operations on a visual display of the airspace. A controller action can have a deadline of 5 seconds, i.e., $\Delta = 5$ sec. $T_{max_lat} = 1$ sec (say). Reasonable values for the other parameters of the model are: k=3, N=10. The time window over which the actions are collected by the protocol (i.e. WOO), can be about 2 seconds. It means that upon the generation of an action by a traffic controller, the protocol allows a window of 2 seconds for the actions of other traffic controllers to be considered under the current epoch. Thereafter, the protocol determines the actions that actually commit over a subsequent time window of, say, 1 second. Thus the epoch duration for this application is 4 seconds. When the number of flights to be scheduled is small, i.e., the airspace is not congested, T_{max_lat} can be, say, 3 seconds and WOO can be, say, 4 seconds. Such operating parameters for the protocol are dynamically prescribed by a session-manager built into the application.

Figure 11 provides a management-oriented view of the application-protocol interface depicting the flow of collaboration-oriented parameters.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

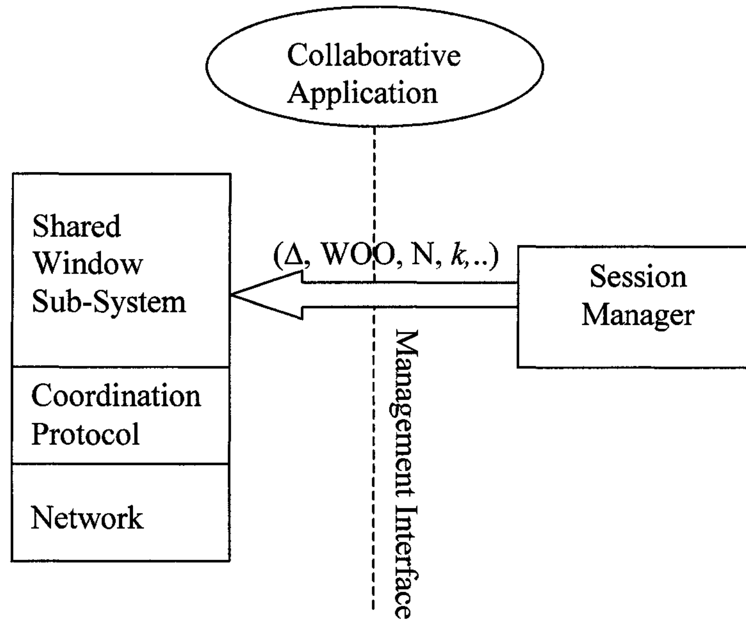


Figure 11: Management-Oriented View of Application-Protocol Interface

7.4 Summary

In this chapter, we introduced our model of epoch driven programming of distributed collaborations. A procedural realization of the model has the following phrases:

- *Timed Agreement Protocol*: It allows determining the committable actions under application specific and system specific timing constraints.
- *Message Level Abstraction*: It consists of message level support mechanisms and programming tools to incorporate multi-media aspects of actions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- *Object Based Window Systems*: It involves application specific segmentation of window objects, the use of private windows to realize action generations and device specific play-out of the multi-media objects.

These project phrases, which synopsize our contributions, will be described in the following chapters.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 8. Programming Tools to Incorporate the Multi-Media as Part of Epochs

Each user action manifests as a collection of multi-media data. From a programming stand point these data are modeled as *messages* that are exchanged between users. Since message is the unit of abstraction in a distributed programming system, modeling the multi-media data as messages allows us to readily use the existing programming tools in our system development. For example a video clip is abstracted as a message in our model, with the visual cues generated by the clip abstracted as a state change in the user. Thus message based programming tools are part of the underlying support mechanisms for our epoch-based model.

8.1 Occurs_After: A Declarative Notation

Collaboration in multimedia settings demands the capability to order messages in real-time. Displaying audio (say) 3 seconds after the video in a lip-synch scenario can be annoying, or an application may demand that some text message be displayed within 1 second of some highlighting, otherwise not to present any of them. We propose a declarative specification of message ordering dependencies. A user entity specifies the temporal ordering of a message z relative to a message y with the following primitive:

$$((z,p_z),\ Occurs_After(y,l_z,u_z))$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Here, p_z is the time period over which the effect of a message z persists, i.e., if the message z is generated at time t_z, then its causally preceding message, say, y should have started getting delivered at time t_y such that:

$$t_z > t_y + p_y .$$

See Figure 6.

The persistence time p_z can be measured in terms of the play-out time of message z, (denoted as δ_z) and the amount of time over which the effect of z lingers (denoted as ζ_z) -- as given by:

$$p_z = \delta_z + \zeta_z ,$$

where, δ_z, $\zeta_z > 0$.

The parameter l_z is the least amount of time a user must wait to display z after the presentation of its causally precedent message(s), which is y in this case.

The parameter u_z is the most amount of time a user can wait to display z after displaying y.

It may be noted that $0 < l_z < u_z$, which captures the discreteness in the presentation timing of media data.

When *Occurs_After(NULL)* is specified, z can be processed without any constraint, i.e., immediately upon arrival from the network.

The *Occurs_After* is a programming notation to explicitly construct the causal ordering relation. The causal order constraints are carried in messages for use by the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

underlying protocols to enforce play-out of the messages in an appropriate sequence and over specified real-time intervals. See figure 12 for a representation of the timing relationship captured by the *Occurs_After* notation.

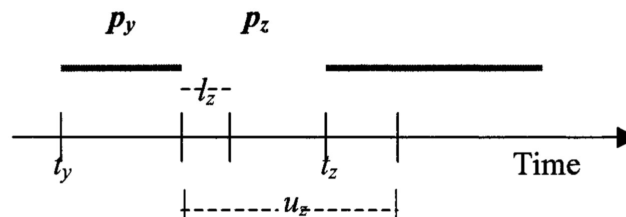


Figure 12: Timing Representation of $((z,p_z)$, *Occurs_After*$(y,l_z,u_z))$

8.2 Dependency Graphs

As we have mentioned, an action can be decomposed into sub actions. For example a user action, in a multimedia conference setting may be a composite of a video, an audio and a draw clip. These composite messages may have a real-time dependency relationship defined on them. Thus cumulative presentation time of an action i: $T_{pres,i}$ can be computed by forming a dependency graph of the constituent messages forming the action and then computing their aggregate presentation time. Dependency graph for an action of three messages $\{x,y,z\}$ having restricted by dependency relationships

$((z,p_z)$, *Occurs_After*$(y,l_z,u_z))$

$((y,p_y)$, *Occurs_After*$(x,l_y,u_y))$

and $((x,p_x)$, NULL)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

may look like:

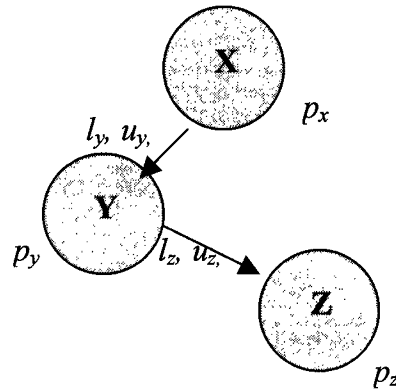


Figure 12: A Sample Dependency Graph of Sequential Actions

Whereas the same action, restricted by the dependency relationships:

$((z,p_z), Occurs_After(x,l_z,u_z))$

$((y,p_y), Occurs_After(x,l_y,u_y))$

and $((x,p_x), NULL)$

may result in the following dependency graph:

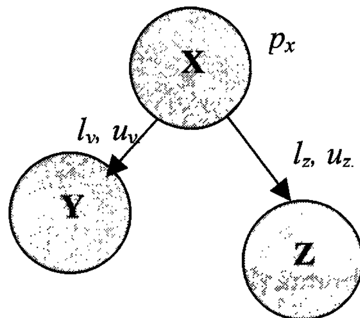


Figure 14: A Sample Dependency Graph of Concurrent Actions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each edge in a dependency graph is associated with two parameters: *l, u* whereas each node is associated with *p*. One possibility for the computation of component action duration is:

$$\mathbf{T^{j}_{pres,i}} = \boldsymbol{p+l+\mathrm{rand}(u-l)} \ldots\ldots (5)$$

Where, $T^{j}_{pres,i}$ corresponds to the presentation time of jth component message constituting a sub action of action *i*; and rand(x) is a function that randomly picks up a number in the range (0,x).

Based on the dependency graph, we can compute the presentation time of an action by finding the farthest (in terms of presentation time) leaf from the root and summing up the presentation time of all the nodes on the path, thus

$$\mathbf{T_{pres,i}} = \Sigma\, \mathbf{T^{j}_{pres,i}} \text{ where the sum is performed over j, j=1,2..} \ldots\mathbf{(5a)}$$

We understand that $T^{j}_{pres,\ i}$ may be different on different sites due to the flexibility in the play-out schedule of constituent messages allowed by *l* and *u*. Which may result in variation in computation of aggregate T_{pres}, and consequently will lead to different values of T_j (start time of an epoch). We could resolve this by eliminating the randomness in the computation of message (component action) presentation time, (equation2); for example if we have

$$\mathbf{T^{j}_{pres,i}} = \boldsymbol{p+l} \quad \ldots\ldots \mathbf{(5b)}$$

Or, $\mathbf{T^{j}_{pres,i}} = \boldsymbol{p+(l+u)/2} \ \ldots\ldots \mathbf{(5c)}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Then, $T^{j}_{pres,i}$ is deterministically computed at all sites, but the presentation controller entity at the participating sites loses the flexibility in scheduling of events, resulting in less optimal use of resources. To resolve this conflict of interest we propose the following compromise: for the computation of presentation time of events (i.e. computation of the length of edges in dependency graph) we use equation (5). But for the computation of t_j, we use the formula,

$$\mathbf{t_{j+1} = t_j + min\{t^{a}_{orig}\} + T_{max_lat} + T_d^{max} + T_{dap} + T'_{pres}} \quad ....(4a)$$

where, $\mathbf{T'_{pres} = \Sigma T_{pres,i}}$, sum is performed over i's of the farthest leaf from the root

$$\mathbf{T_{pres,i} = \Sigma\, T^{j}_{pres,i}} \quad \mathbf{(5a)}$$

$$\mathbf{T^{j}_{pres,i}} = p+u \quad ...... \ \mathbf{(5d)}$$

In this chapter we described the properties of message level abstraction needed to support our epoch model. These properties are specified in terms of message-level timing and ordering. In the following chapter we describe the timed agreement protocol.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 9. Timed Distributed Agreement Protocol

Success of our model rests on our ability to develop an agreement protocol that enables the participating entities to agree on a set of actions to commit under stipulated timing constraints. The real-time nature of collaborations expects certain timing guarantees from the underlying network system. These guarantees can be deterministic or probabilistic, depending upon the choice of underlying network and the application characteristics. The type of network guarantees impacts the structure of agreement protocols in determining the epoch duration (equation 4). We describe the details of these protocols in this section.

9.1 Assumptions

Our agreement protocol is based on the following assumptions:

- Processes do not fail.

Since we assumed static sessions [chapter 6] (i.e., process group membership does not change)[10], our assumption that processes do not fail is not a limitation[11].

- All the protocol agents have globally synchronized clocks.

This can be achieved by NTP or GPS clock.

- The network supports message-level multicast [Deer'88].

---

[10] This is achieved by guaranteeing that participants join only in the beginning of a session and they are not allowed to leave in the middle of a session.

[11] It is possible to reach agreement in the presence of process failures [Lynch, N]. However, for the purpose of this thesis, it is an orthogonal problem.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The multicast facility is available in many contemporary networks (e.g, IP Multicast). However, the protocol level requirements and/or the type of networks determine the message delivery semantics[12].

We assume an API that allows the timing parameters of user-level actions to be made available to the protocol agents executed at various user sites. Furthermore, we assume that the delay behavior of the network --- whether deterministic or probabilistic --- can be prescribed by the protocol[13].

9.2 Structure of Agreement Protocol

Our agreement protocol rests on the premise that there exists a bound on the skew on the time of generation of user actions. That is, if t_{first} is the generation time of the very first action and t_{last} is the generation of the last action (generation location may be different), then in order for the last action to be valid it has to satisfy the latency condition:

$$0 < (t_{orig}^{k} - t_{orig}^{1}) < T_{max_lat}$$

---

[12] We assume a multicast abstraction that allows a sender to receive its own message back atomically. This allows uniformity in the communication structure whereby the protocol actions at the sender of a message m will be governed by the same message-driven processing as at any receiver of m.

[13] It may be noted that the following protocol elements are implemented:

- Each session participant is associated with a unique identifiers;
- Session participants form a group that can be addressed by invoking its group identifier.

We omit these details in our protocol description, for brevity.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An agreement protocol that is needed for our model, should, in general have the following steps:

- The user actions are multicast to the group of participants using the message-level transport primitives of the underlying network;
- Each message carrying an action is associated with a time-stamp indicating its time of generation.
- Agree on an unordered non-empty set of actions, S, that is received by all the participating entities;
- Check for compatibility and generate an ordered set of committable actions: $S' \subseteq S$;
- Enforce the timeliness constraints on S' (based on Δ) to determine the set of actions that can actually be delivered to the user level windows.

The agreement protocol also has the information about the maximum number of users, k out of a possible n, that are eligible to generate action in a particular epoch. This information comes from the floor controller (When this information is not known k is set to the value of n), and may be prescribed as part of a termination condition for the protocol. So with the knowledge of k, an algorithm executed at various sites could reach agreement in a highly asynchronous manner (when compared to a case where the knowledge of k is not available), without compromising the logical synchrony requirement.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

How the various protocol steps are realized is intricately interwoven with the delay behavior of the underlying network. In this thesis, we consider two representative models of networks whose delay behavior can be controlled in some way by protocol agents.

9.3 Delay Specification by Protocol

The protocol mechanisms to agree on a set of actions S are tied to the level of delay guarantees provided by the network. We assume that an agreement protocol will be able to prescribe a delay bound D_Q on the delivery of messages transported through the underlying network. The prescription of D_Q should represent the time granularity of actions, i.e., $D_Q < \max\{\Delta^a\} + c$, where c is a small positive value. Here the parameter D_Q may itself be derivable from the application level specs of the timing constraints on messages namely $\{l,u,\zeta,\Delta,p\}$.

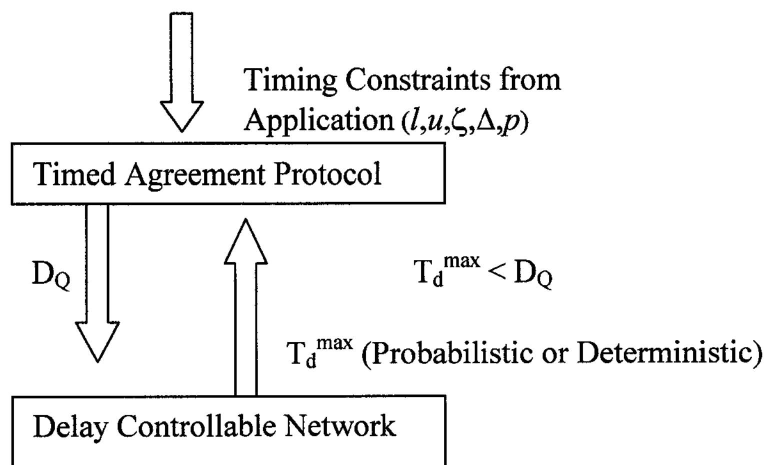


Figure 15: The Flow of Timing Information in the System

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The above premise assumes that the network is 'delay-programmable', i.e., the network has the ability to enforce a delay bound T_d^{max} such that $T_d^{max} < D_Q$, where the network parameter T_d^{max} may be derived from D_Q by the protocol layer. In this light, our earlier work on multi-media synchronization employed delay-controllable networks, with the attendant advantages of exploiting user tolerance to media data delays and skews to optimize network resources [Ravin'04].

With a delay-controllable network, the data exchange phase of an agreement protocol can be structured as follows. A site P_i that originates an action a multicasts a data message $m(a, t^a_{orig})$ to all the sites in the session. Upon receiving such data messages, a site P_j decides on how long to wait for other data messages. If $m(d_a, t^a_{orig})$ is the first data message received by P_j , then P_j opens a time window for receiving other data messages:

$$T_{rcv}(j) = t^a_{orig} + D_Q + T_{max_lat},$$

treating a as the earliest action in P_j's view. This receive window allows P_j to receive the various remaining actions generated in the current epoch, and proceed to the next phase of the agreement protocol to determine the start of next epoch. Determining the start time of next epoch requires knowledge of the time at which the various sites agree on the common set of actions received by them and the play-out times of these actions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The level of delay guarantee by the network can be either deterministic or probabilistic (see figure 15). The type of delay guarantee influences the structure of the agreement protocol. Since message flows, with the possibility of missed deadlines, have to be dealt with differently than the guaranteed on-time message flows.

We now describe two protocol mechanisms for reaching agreement: first, for deterministic delay bounded networks, and second, for probabilistic delay bounded networks. Note that in both the protocols, we model the 'passage of time' as a value being agreed upon.

9.4 Protocol on Deterministic Delay Bounded Networks

Agreeing on the set of actions *S* is made easier if there exists an enforceable deterministic delay bound D_Q on the delivery of messages. Note that the above assumption subsumes a loss-free message delivery by the network (see the delay behavior in Figure 16).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

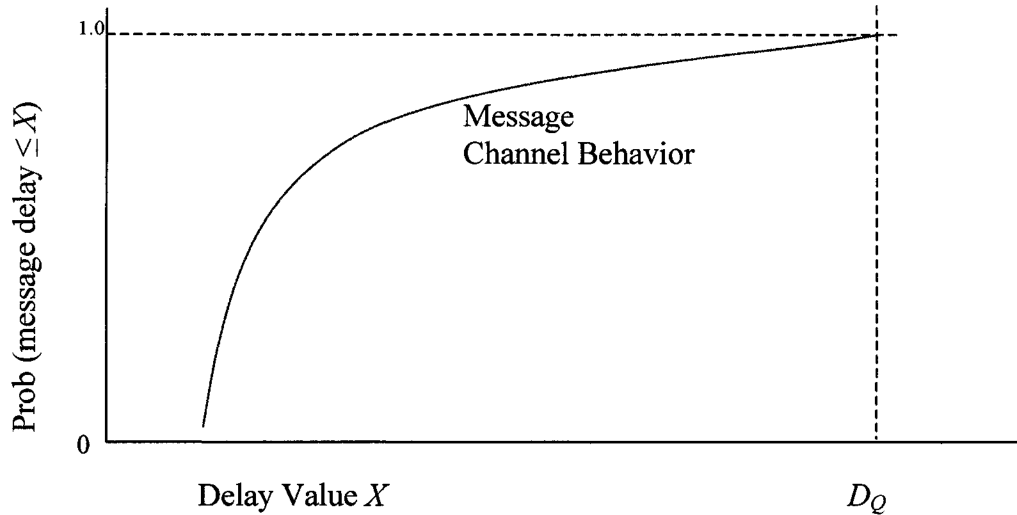


Figure 16: Illustration of Delay Behavior of Message Channel

At every site, the local agent waits for all the generated actions to arrive. Due to the delay bound enforced by the network, the earliest action (in terms of its generation time) will reach every other site (due to bounded delay on channels) before time t, where

$$t = \min\{t^{a}_{orig}\} + T_d^{max}.$$

Due to the maximum allowed latency (a session level parameter that puts a bound on the skew on the generation time of actions), the latest action must have been generated before: $\min\{t^{a}_{orig}\} + T_{max_lat}$,

and will be received by all the sites within T_d^{max} thereafter. Thus, all the generated actions are guaranteed to be received before:

$$\min\{ t^{a}_{orig} \} + T_{max_lat} + T_d^{max}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This worst case time of arrival can be estimated at all sites as soon as they receive the earliest action --- which as shown earlier will be no later than min{ t^{a}_{orig} }+$T_d^{\max}$.

Since the message channels are delay bounded by $T_d^{\max}$, each site may compute the termination time of the current epoch as:

$$\max(\{ t^{1}_{\mathrm{orig}}, t^{2}_{\mathrm{orig}}, \ldots, t^{k''}_{\mathrm{orig}} \}_{1,2,\ldots,k'' \in S'}) + T_d^{\max} + \alpha,$$

where, α depicts the time needed to play-out the various media data in the committable set of actions S' and any non-deterministic time that needs to elapse after the play-out as part of the current epoch (such as thinking time). How S' is determined is described later in chapter 10.

The algorithm executed by each site j for assembling S (j=1,2,…N) is given by the following pseudo-code:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Protocol Entity at Site *j*:

```
next_epoch_start=0;// Initialization

forever{
        //maximum number of eligible action is k (c.f. section 7.1) ;
        current_epoch_start = next_epoch_start;
        wait for an action c_1 to arrive over multicast channel;
        S = {c_1};
        action_recived=1;
        min_gen_time= c_1. t^a_orig;
        timeout=min_gen_time+D_Q + T_max_lat;
        while(current_time<timeout && action_received <k){
          if(timeout)
                  break;
          if(action c arrives){
             action_recived=action_recived+1;
             S = S ∪ c;
             if(min_gen_time > c. t^a_orig){
                min_gen_time = c. t^a_orig;
                //Re-adjust timeout with a smaller value:
                timeout = min_gen_time+ T_max_lat + D_Q;
             }
          }
        }
        max_gen_time=min_gen_time+ T_max_lat ;
        compute compatible set of actions S' ⊆ S ;
        schedule media play-out ({c'}_∀c'∈S') on various devices;
        next_epoch_start=current_epoch_start+max_gen_time+D_Q+ α;
        wait_until(next_epoch_start);//idle to model the passage of time
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

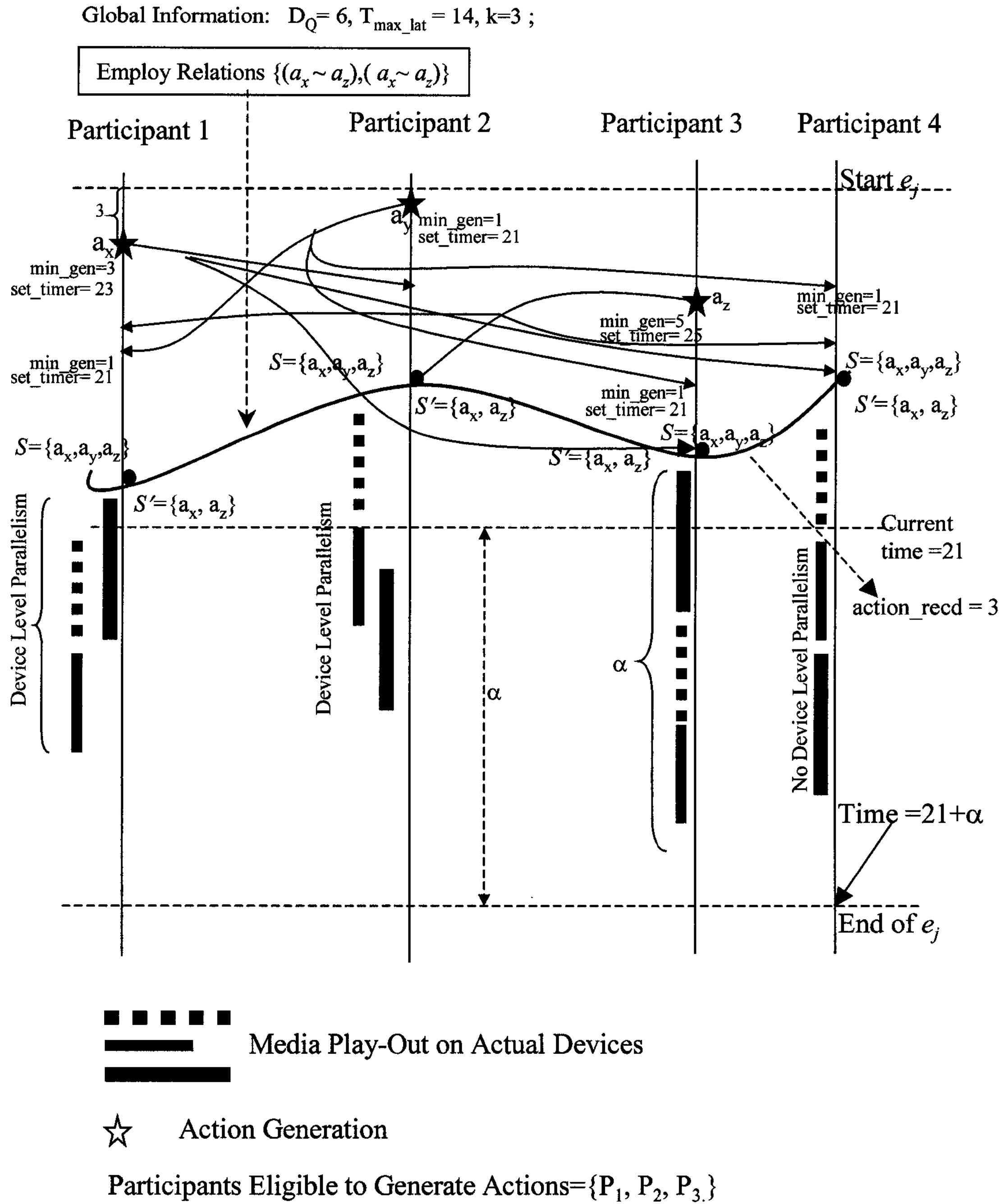


Figure 17: Protocol Scenario for Deterministic Delay Bounded Channel (Case of All Eligible Participants Generating Action)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It can be seen that all the sites will declare the completion of the assembly of S at the same time: $min\ \{t^a_{\text{orig}}\}+T_{\text{max_lat}}+D_Q$ relative to the start time of current epoch. The term $min\ \{t^a_{\text{orig}}\}+T_{\text{max_lat}}$ indicates the time by which all actions from the potentially eligible participants would have been generated. Since $T_d^{max} < D_Q$, the participants are guaranteed to reach agreement at exactly the same time as above. Thus, with the availability of deterministically delay-bounded networks, the agents can locally decide on the set S without an explicit exchange of messages with other agents.

Figure 17 and 18 are two examples of simulated protocol behavior using some sample values for action generation time, D_Q, and $T_{\text{max_lat}}$, for both the cases maximum number of participants eligible to generate action, k, is 3. We consider two different scenarios: one where all the eligible participants generate actions (Figure 17) and one where only a subset of them generates action (Figure 18).

For the scenario depicted in Figure 17, with the knowledge of k, it is possible that each of the participants 1 through 4, receive all 3 the actions a_x, a_y, a_z (via multicast) much earlier than the stipulated timeout of 21 units. Of these three actions only a_x, a_z are in the committable set, which are then played out. A key point is the variability in the media presentation times at various sites.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

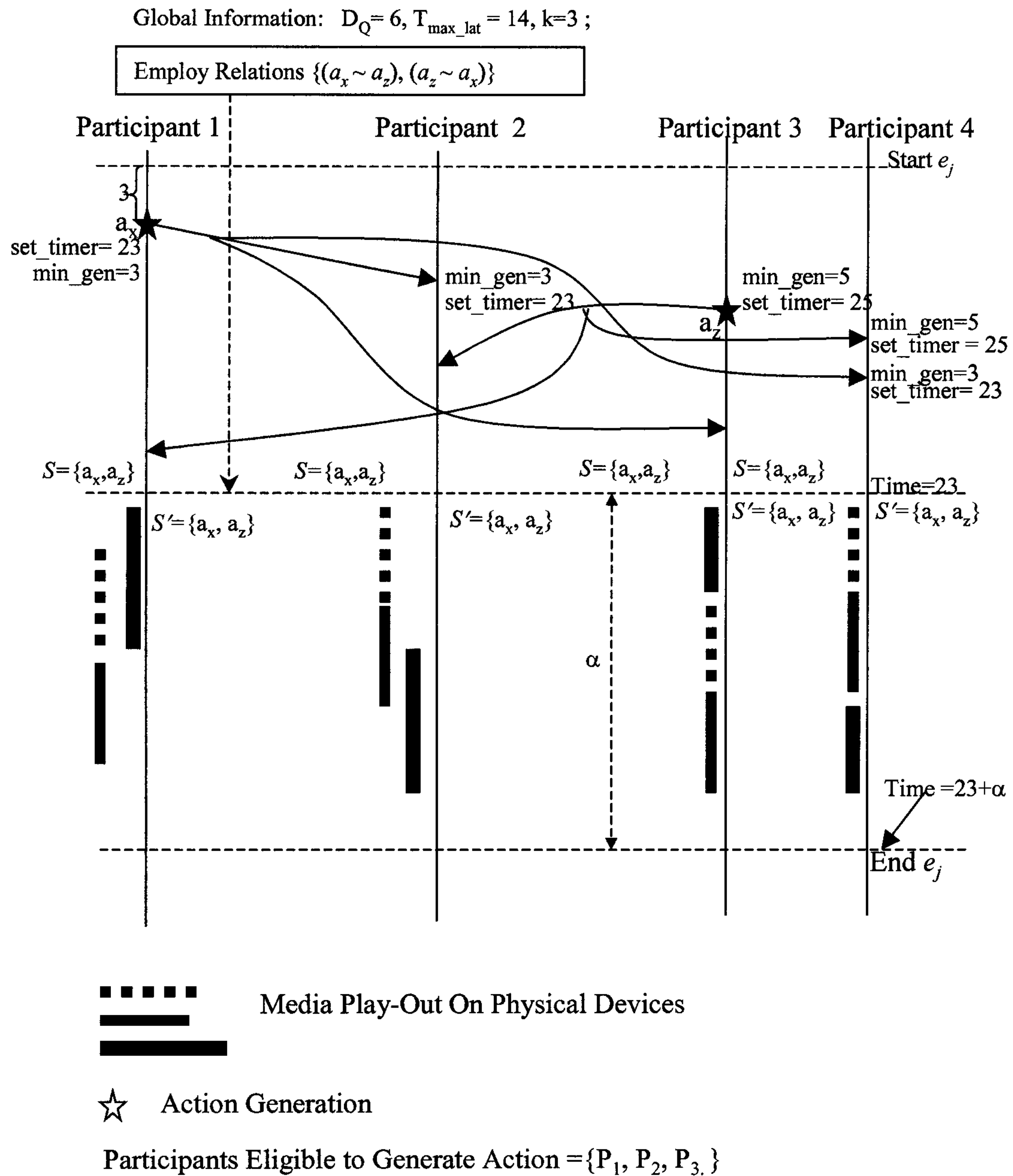


Figure 18: Protocol Scenario for Deterministic Delay Bounded Channel (Case of Only a Subset of Eligible Participants Generating Action)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For the scenario where only a subset of the participants generates actions (Figure 18), each participant has to wait for their timer to expire before starting playing out actions.

Applications in which the user actions have rigid time constraints, such as the air traffic control described in section 7.3, benefit from the use of deterministic channels. The following protocol-level parameter values seem appropriate for the air traffic control application. The network delay D_Q should be much smaller in comparison to the user-level response time of 1 second. A reasonable value of D_Q can be 100 msec. To accommodate the timing constraints, the protocol may set k′ as, say, 1 or 2, based on the asynchrony in the generation of actions by the traffic controllers. Such a choice of parameters is possible due to the context awareness built into the protocol.

From a distributed termination standpoint, the protocol falls under the class of synchronous algorithms. This is because the system bounds such as network delay (T_d^{max}) and user-level asynchrony (T_{max_lat}) are known and the clocks at various user sites are synchronized.

Although it is possible to develop a deterministically delay-bounded network, their scope is somewhat limited due to scalability issues. Since the messages that carry actions can be large (audio clip, video clip), providing a rigidly controllable delay

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

bound on their delivery can be expensive, if not impossible, in terms of network resource consumption (such as bandwidth and message buffers). In this light, it may be noted that the semantics of timed multi-media actions allows a certain level of tolerance to excessive delays in their presentations. Accordingly, we can relax the delay constraints on media data presentations within the tolerance limit allowed by the user actions. A higher value of D_Q allows sending a message over a transport path with longer message queues and/or containing lower bit-rate communication links. In general, message level asynchrony captures the user tolerance to latency in data presentation.

9.5 Data and Control Channel Separation

Incorporating the above idea in a presentation protocol, we overcome the problem of excessive resource consumption by providing for two types of channels:

- *data channels* that carry the messages containing user data;
- *control channels* that carry protocol level messages for agreement purposes.

We require the control message channels to provide deterministic delay bounds, but relax the requirement of delay bounds for data channels. Unlike deterministic network channels where there is single delay bound, namely, D_Q, our formulation of probabilistic-networks has two delay bounds: D_{Q1} that is probabilistically enforced on data channels and D_{Q2} that is deterministically enforced on control channels. See

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

figure 19 for illustration[14]. Since protocol control messages carry only meta-information about the data exchanges (rather than the data itself), they are substantially smaller in size. Thus providing for control and data channel separation is an engineering optimization to reduce the network resource consumption while keeping the application-level data delivery within tolerable limits.

The separation of data and control channels has been proposed elsewhere (see for example in implementation of RTCP (RFC 3550) to support multimedia broadcast or [Minz91] to support tele-communication signaling). We extend these ideas to achieve our goal of user-level media synchronization under resource constraints. We describe such a protocol next:

---

[14] The deterministic delay bound D_{Q2} on the control channel in our formulation of probabilistic networks corresponds to the delay bound of the D_Q in deterministic networks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

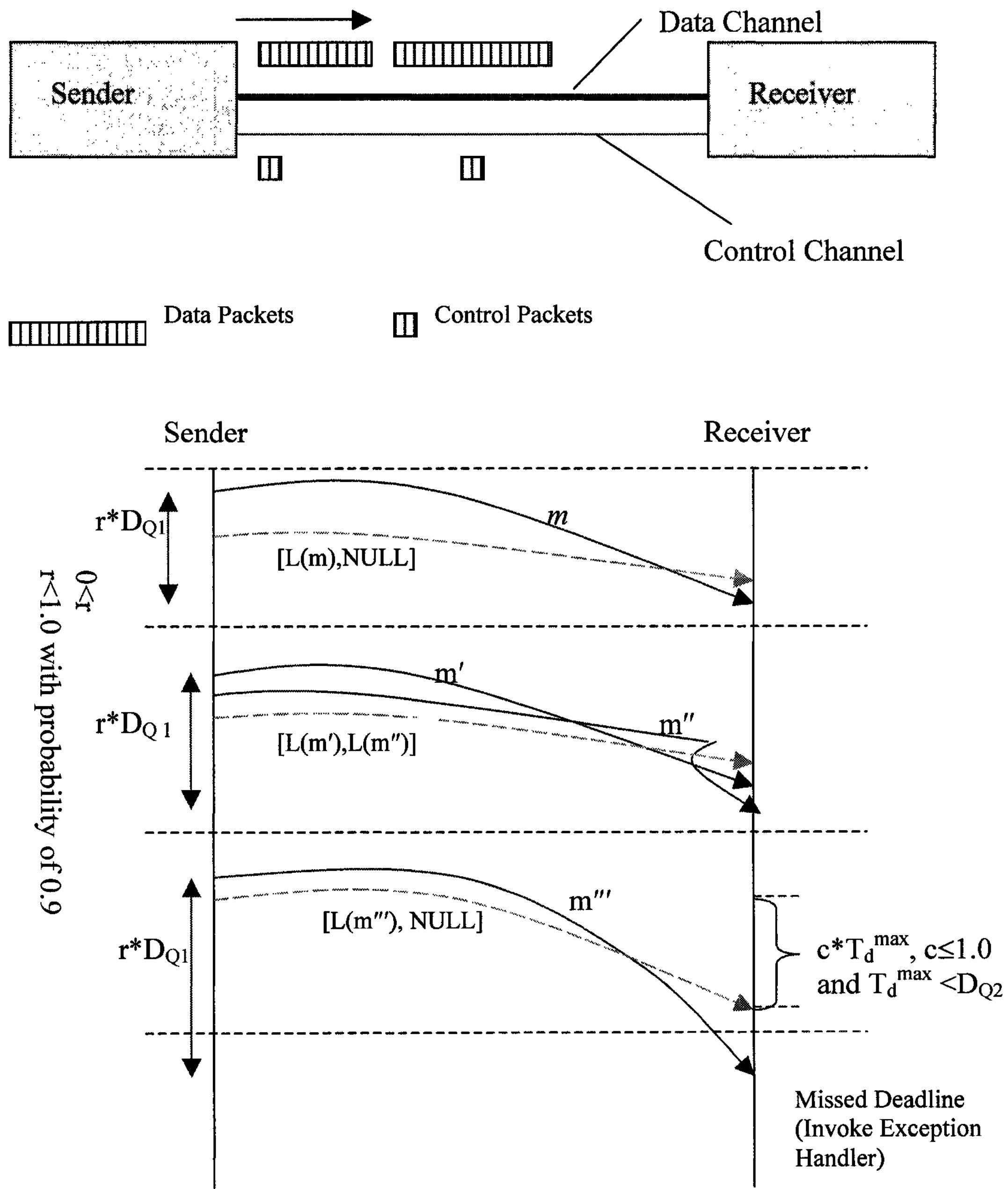


Figure 19: Data and Control Channel Separation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

9.6 Protocols With Probabilistic Delay Bounded Data Channels

In this section, we provide an algorithm to construct *S* under the premise that the users are connected by probabilistically delay bounded data channels [Felb'01] and deterministically delay bounded control channels. Here the reliability semantics of data channels is that any message arriving at a protocol entity beyond a prescribed delay bound D_{Q1} is deemed as lost in the network. Here, the delay prescription D_{Q1} may be in terms of a required percentile delivery guarantee ---say, 90% of packets need to be delivered by the network before expiry of a time limit D_{Q1} (shaded area in figure 17 shows the possibility of messages missing the deadline D_{Q1}). Since control purport to facilitate the delivery of data, it is reasonable to expect that the delay bound T_d^{max} on control channels satisfies the condition: $T_d^{max} < D_{Q2}$. In general, $D_{Q2} < D_{Q1}$. Figure 20 illustrates the delay behavior of data and control channels.

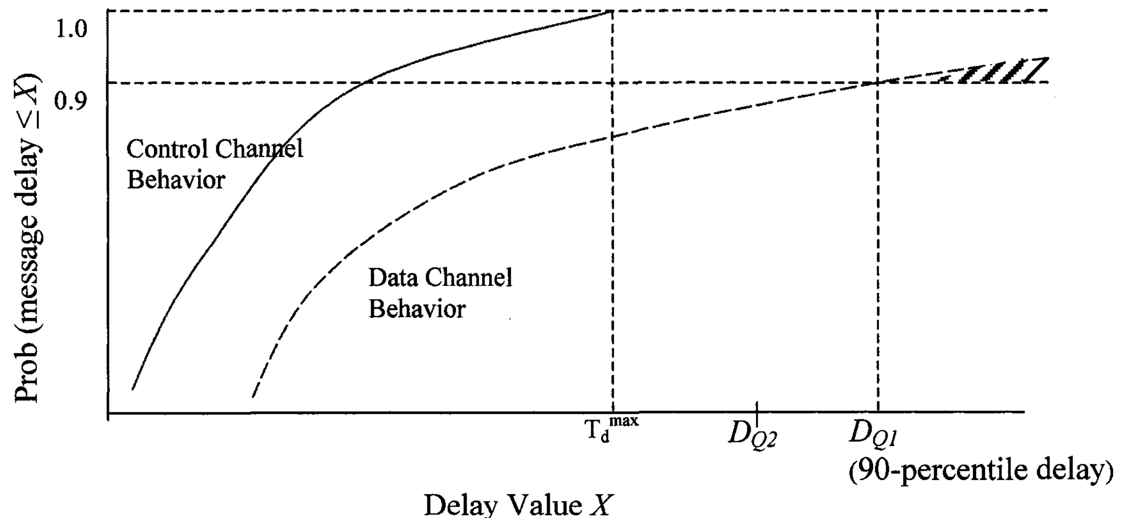


Figure 20: Illustration of Delay Behavior of Data and Control Channels

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Under the channel behavior described above, the control information about various actions generated before t_{orig}^{1}+ T_{max_lat} will be received by all the sites before t_{orig}^{1}+ T_{max_lat} + T_d^{max}. However, due to the probabilistic delays on the data channels, there is no guarantee that these actions will be received on time at all the sites. Given the percentile value associated with the prescription of D_{Q1}, the protocol entity executing at a site can determine, with a reasonable level of accuracy, the likelihood of one or more actions arriving before the expiry of D_{Q1}. Accordingly, the algorithm described in the previous section should now be modified to deal with the situation that the locally received set of actions S, may not be the same at every site. Thus, the various sites should agree on the maximal common set of action received by them. It is from this common set of actions the committable set of actions S' will be determined.

The decision at any given site about closing its receive window can be deferred to until close to D_{Q1}, in order to get a larger set S of the generated actions. Upon reaching such a decision point, the site starts its local instance of the protocol to agree on S with all the other sites.

This algorithm also falls under the category of synchronous distributed algorithms, since the control channel has an enforceable delay bound of D_{Q2}. The initial part of the algorithm that allows a site to collect data messages from other sites over a certain time interval is the same as the algorithm described in section 9.4. However, the fact

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that the data channels have probabilistic delay behaviors imposes some changes to the algorithm we described in that section for deterministic data channels. The changes are required to take into account the possibility of different sites receiving different sets of data messages (including a null set) within a time interval based on the D_{Q1} parameter. We solve this problem by incorporating gossip techniques in our protocol [Tana'02].

Consider the first receipt of a data message $m(d_a, t^a_{orig})$ at site P_j in the protocol described in section 9.4. The receive window $T_{rcv}(j)$ now allows P_j to receive all the remaining actions generated in the current epoch only with a certain probability. Given that it is possible for P_j to not receive all the actions (or any action at all) during T_{rcv}, however conservatively set, P_j needs to determine which of the actions that P_j has received have also been received by the other sites. For this purpose, we require each site to multicast a vector listing the actions it has received during its T_{rcv}.

Accordingly P_j multicasts, over the control channel, a N-element vector V_j upon either the expiry of T_{rcv} or the receipt of actions from all k eligible participants. Each element of V_j indicates whether an action from the corresponding participant has been received by P_j or not. The exchange of these vectors among participants allows the latter to agree on a common set of actions that have been received at every site.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Suppose V_j is the first vector that a site P_i receives, say, at time t_i. It is possible P_i has not received any data message during the interval $(0,t_i)$. Since $t_i > \max(\{t^x_{orig}\}_{\forall x \in Vj})$, P_i sets:

$$T_{rcv}(i) = t_i + D_{Q1} ,$$

and continues to await the receipt of data messages. This condition provides liveness in the algorithm by having P_i terminate its wait within a finite amount of time relative to the generation of one or more actions.

Suppose $T_{rcv}(i)$ expires without P_i receiving any data messages. Then, V_i indicates a null set. This means that no action will be committed in the current epoch.

Determining the start time of next epoch requires knowledge of the time at which the various sites agree on the common set of actions S received by them and the play-out times of these actions. Due to the asynchronous nature of the exchange of vectors we timestamp the vectors with their time of generation. Since the control channels are delay bounded by D_{Q2}, each site will compute the termination time of vector exchanges as:

$$\max(\{V_i.TS\}_{i=1,2,..N}) + D_{Q2} .$$

The committable set S' is determined from the set S agreed upon as above (described later in chapter 10). Then the start time of next epoch is given by:

$$\max(\{V_i.TS\}_{i=1,2,..N}) + D_{Q2} + \alpha.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Our protocol executed at each site *j* for probabilistic delay bounded channels[15] is given by the following pseudo-code (*j*=1,2,..N):

[15] Since a message is the unit of transport through the network, it is possible that some component messages of an action, say *a*, might be excessively delayed to cause only some sub-actions of *a* to be executed at site *j*. This partial execution of *a* may sometimes be permissible by the application (e.g., a video clip describing an object may miss the deadline but the top-level action may still commit using the context information conveyed by other media such as voice clip). Such a partial commit can be handled in an application-supplied exception handler that is invoked by the protocol agent at site *j* upon detecting the missed messages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Protocol Entity at Site *j*:

```
next_epoch_start=0;
forever{
   //maximum number of eligible action is k (c.f. section 7.1)
   current_epoch_start=next_epoch_start;
   VINT={1,......,k} //Common set of actions received at all sites
   S = {Φ};  //Actions received
   V={Φ};    //Participants whose actions are received
   vcounter=0;  //Number of received vectors
   VTS=0; //Largest timestamp of received vectors
   timeout=L   //L is an arbitrarily large value
   while(current_time<timeout && |S| < k && S≠ VINT){
      receive message m from participant P_i
         if(m ≡ action c ){
            S = S ∪ c;
            V= V ∪ {i};
            if(|S|= =1)
               timeout = t^c_orig + T_max_lat+ D_QI ;
               //a reasonable wait time for all data message to arrive
         }
         if(m ≡ vector V'){
            vcounter=vcounter+1;
            VINT = VINT ∩ V';
            VTS=max ({VTS,V'.ts});
            if(|S|= =0)
               timeout= current_time + D_QI ;
               //T_max_lat is already accounted for in P_i's generation of V'
         }
   }
   multicast V over control channel;
   while(vcounter < N){
      wait for message on control channel
      if(vector V' arrives){
         vcounter = vcounter+1;
         VINT = VINT ∩ V';
      }
   }
   S = VINT;
   compute compatible set of actions S'⊆ S
   schedule media play-out ({c'}∀c'∈ S')
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    next_epoch_start=current_epoch_start+VTS+ D_Q2 + α;
    wait_until(next_epoch_start)
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

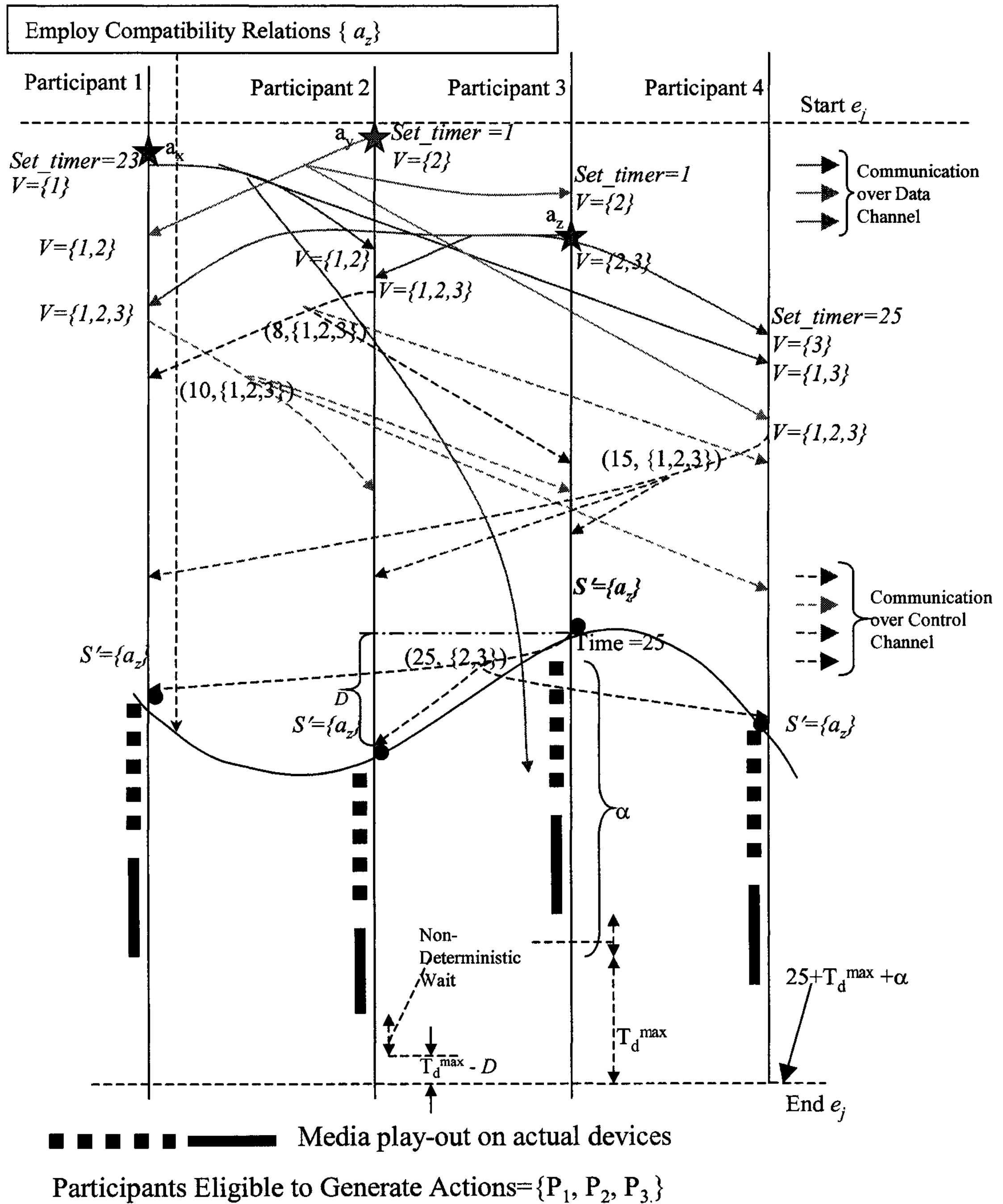


Figure 21: A Protocol Scenario on Probabilistic Delay Bounded Data Channel

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since all the vectors are sent over control channels, a site cannot terminate its protocol any sooner than VTS+ D_{Q2} (relative to current epoch start). In figure 21, we give an illustration of the execution of the algorithm with sample values for action generation time, T_{max_lat}, D_{Q1}, D_{Q2} and k.

9.7 Performance Evaluation of the Model

A salient feature of our model is our ability to commit more than one action in an epoch. This feature should be captured in a performance analysis of the protocol. The metric we use is the number of committable actions that get aborted due to excessive network delays and/or the user-level asynchrony. Due to the inherent difficulties in measuring the user-level asynchrony, we limit our analysis to the impact of network-induced delays on the successful completion of committable actions. So, without loss of generality, our performance analysis assumes that all user actions are generated at the same time in the beginning of an epoch.

With a deterministically delay bounded data channels, the application can prescribe a suitable delay bound D_Q such that all the actions generated reach the participants sites before the expiry of Δ. In other words, the network parameters can be engineered in such a way that a non-committal of an action is not due to excessive network delays. Thus, the interesting cases for our analysis arise in probabilistic delay bounded data channels.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Our goal is to represent the randomness in network delays by a probability distribution that closely approximates a shared infrastructure network. Typically, the delay bound D_{Q1} represents the 90-percentile value as determined from the given distribution. For simplicity, we assume that all the k eligible users generate actions in an epoch. Let $P_D(D_{Q1})(x)$ represent the probability density function of the network delay D whose 90-percentile value is D_{Q1}. An action *a* commits iff the message *m*(*a*) from the sender reaches all the participating sites within $(\Delta - D_{Q2})$ time units. This is because, the agreement protocol executed over the control channel is guaranteed to complete in D_{Q2} time units after *m*(*a*) has arrived at various sites. The probability that *m*(*a*) reaches a site in a timely manner, denoted as *L*, may be given as:

$$\int_0^{\Delta - D_{Q2}} P_D(D_{Q1})(x)\, dx$$

So the probability that *m*(*a*) reaches all the sites is $(L)^N$. Assuming[16], for simplicity, that all the k actions are committable the mean number of actions that will commit in the epoch is $k * L^N$.

[16] Our analysis is conservative (and somewhat rudimentary too) in the sense that the action *a* is deemed as aborted if *m*(*a*) reaches any of the sites during interval $(\Delta - D_{Q2}, \Delta)$.This is despite the possibility of being able to still commit *a* in cases where the agreement protocol can complete before expiry of Δ. In this light, our simplification of the analysis is merely for enhancing readability (but without compromising generality).Nevertheless, our protocol does in fact actually handle user-level asynchrony and network delay variability in a way to minimize the number of aborts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We use a normal distribution to represent $P_D(D_{Q1})(x)$ with an appropriate choice of the mean. The choice of normal distribution is to keep the mathematical analysis tractable. As D_{Q1} is changed, the mean of the distribution is changed in a way to provide a widely spread out set of delay values. We believe this closely approximates real networks where a larger D_{Q1} basically means that packets suffer a wide range of delays --- which often arises due to lax scheduling of packets in the transport paths.

The graph in figure 22 illustrates the performance results for a sample set of parameters N=5, k=4 by varying Δ between 75 and 125 msec and choosing D_{Q1} to lie in the range from 50 to 150 msec.

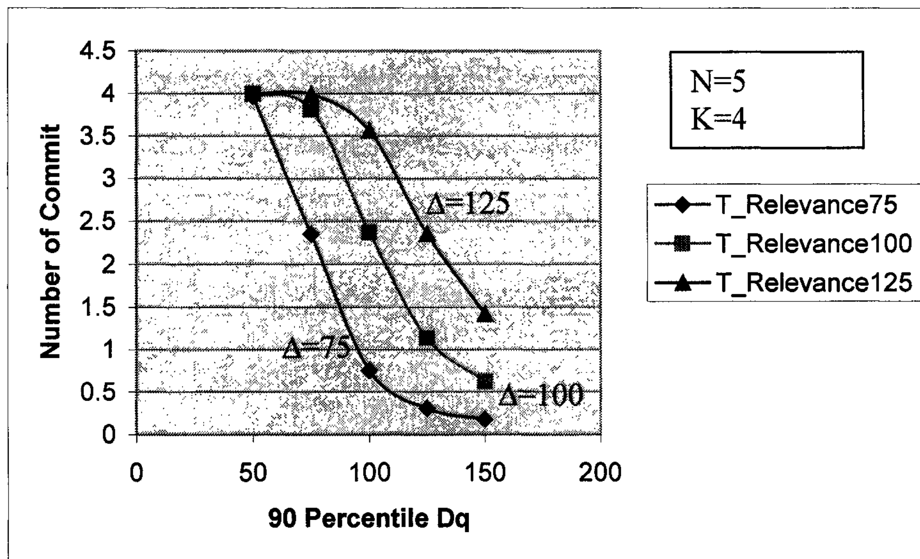


Figure 22: Performance Evaluation Graph

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As can be seen from the graph, the case of D_{Q1}= 50 msec yields a higher number of commits relative to the cases of higher values of D_{Q1} for a given value of Δ. In other words, smaller the value of D_{Q1} relative to Δ, higher is the number of commits. This corroborates our earlier observation that $D_{Q1} << \Delta$ is the optimal operating point for the system.

In summary, we described protocol mechanisms for reaching agreements under two types of networks: one with deterministic delay bounded message channels (section 9.4) and the other with deterministic delay bounded control message channels and probabilistic delay bounded data message channels (section 9.5). These protocols trade off between network resources and user-level quality of presentation. We have studied both the protocols by system-level prototyping (aided by semi-formal analysis), which brings out the salient features of our model of distributed collaboration.

As pointed out earlier, our agreement protocols fall under the class of synchronous distributed algorithms. Our focus on the synchrony property of the underlying system is to bring out salient features of our model of distributed collaboration in a simple way. We do not consider partially synchronous systems (i.e., a network that enforces a T_d^{max} but the protocol layer does not know about it --- which means that D_Q (or D_{Q2}) cannot be specified) and fully asynchronous systems (i.e., both the data and control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

channels exhibit a probabilistic delay behavior) in our protocols. Our non-consideration of such systems that do guarantee any synchrony is not a limitation of our model, but is merely to avoid getting engrossed into the protocol-level complexities induced by such systems [Afek'94][Kemp'02][Dwor'88][Fisc'85] --- which are outside the scope of this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 10. Deciding on Local Presentation

After all the sites have agreed on the same set of actions S, we need to carry out compatibility checks on these the actions in order to determine a committable set of actions S'. A presentation schedule is then determined on the actions contained in S'.

10.1 Rule Space for Generating Compatibility Relations Among Actions

Given a collaboration application, the rule space maintained by the session manager is a set of relations prescribing the possible conflicts between various actions potentially generated by the participants. These relations are prescribed in terms of the allowed action space and the shared window state at the start of current epoch.

Given two possible actions $A_1(x)$ and $A_2(x)$ allowed in the current region of state space X, a rule prescribing a conflict takes the form:

$A_1(x)$ conflicts_with $A_2(x)$ $\forall\, x \in X$.

This "conflicts_with" relation maps onto compatibility relations among actions $a_1 \in A_1(x)$ and $a_2 \in A_2(x)$. The mapping manifests in the form of not prescribing a compatibility relation: $a_1 \sim a_2$.

10.2 Constructing Rule Base for a Sample Application

Consider an example of construction activities in a city are regulated by the zoning restrictions. Here, a restriction on the licensing of building constructions may be that

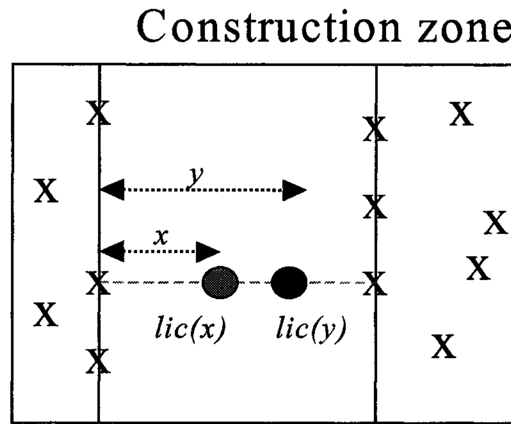
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

two buildings cannot be closer to each other by less than 100 ft when the number of buildings in a zone is less than 100. A rider may be that when the number of buildings in a zone exceeds 100, the city engineer-in-charge decides on the licensing on a case by case basis.

Here, a collaboration-oriented computer model of this application in one dimension may represent the spacing between the nearest two buildings as BSPC and keeps track of this as a state variable and the number of buildings (NBLD) in a zone as the application-level state. See Figure 23. The licensing restrictions may be prescribed as conflict rules, namely:

"License requests for buildings that do not shrink the inter-spacing between all the buildings to lower than 300 ft are non-conflicting".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



BSPC

NBLD=13

X: existing building

(lic(400), ~ lic(600)), (lic(600) ~ lic(400))

$g(\{BSPC, NBLD\}, lic(a)) = \{BSPC\text{-}a, NBLD+1\}$ for $NBLD < 100$

Figure 23: Example: Layout of A Building Construction Zone in a City

Let lic(x), lic(y) denote a license request action to construct a building at x relative to the nearest building at, say, the eastern end of the zone. The rule base $\Re$ for this application may then be stated as (we use notation *a~b* to mean that action *a* does not conflict with action *b*):

$$\forall x, y | [\ (x > 300) \wedge (BSPC\text{-}x > 300) \wedge (y > 300) \wedge (BSPC\text{-}y > 300) \wedge (\max(x, y) - \min(x, y)) > 300 \wedge (NBLD \leq 98)]$$

$$\Leftrightarrow (lic(x) \sim lic(y)) \wedge (lic(y) \sim (lic(x))$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The State transitions relations are:

g(lic(x), (BSPC, NBLD)) = ((BSPC-x), (NBLD+1)). In these relations, the functions like '+', '-', and '>' are application-specific, supplied through the API.

10.3 Mapping of Compatibility Relations Onto Action Execution

S' basically prescribes an execution sequence on a set of (compatible) actions $a_1, a_2, \ldots\ldots a_{k'''}$ denoted as $S' = \langle a_1, a_2, \ldots, a_{k'''} \rangle$. For example, $S' = \langle a_1, a_2 \rangle$ means that action a_1 gets committed on the user window before a_2. To determine S', we first consider a set C of possible execution sequences of actions extracted from S satisfying the compatibility conditions. We then select the longest execution sequence from C.

The production rules for extracting the possible execution sequences from S can be prescribed from the compatibility relations on the actions generated in the current epoch. Without loss of generality, we assume that each compatible relation prescribed by the session manager (at the start of epoch) is on a pair of actions, say, in the form, $a_i \sim a_j$. We believe this is a natural way of prescribing the compatibility relations for use in the algorithm, given that the rule base from which these relations are derived is based on the prescription of conflicts between pairs of actions under various scenarios.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

10.4 Algorithmic Procedures to Determine Executable Sequences

Denote the set of compatibility relations on the actions S={a_1, a_2,....., $a_{k''}$} that was determined by the agreement protocol, as:

$$\Re\ (\{a_1,a_2,...,a_{k''}\}) \equiv \{x\sim y\}$$

Where x and y can be elementary actions drawn from S or can be multiple actions contained in S executed in a certain sequence. The compatibility relations are generated by the session manager based on the rule space associated with the state at the start of the current epoch.

The algorithm begins by checking the compatibility of various pair-wise actions drawn from S={a_1, a_2,....., $a_{k''}$}. The properties pertaining to the compatibility relations described in section 5.1 are used in identifying the various possible execution sequences of different lengths. The algorithm to generate the sequences can best be described with a sample (representative) set of compatibility relations. Consider, for instance, $S=\{a_1,a_2,a_3\}$ governed by the compatibility relations $\{(a_1\sim a_2), (a_2\sim a_1), (a_1\sim a_3), a_3\sim (a_1\sim a_2)\}$. This will yield the following possible execution sequences:

$$\{<a_1>, <a_2>, <a_3>, <a_1,a_2>,<a_2,a_1>,<a_3,a_1>,<a_2,a_1,a_3>\}.$$

An exact algorithmic procedure to determine the execution sequences is described in the following section.

When more than one execution sequence meets the selection criteria, it is possible to deterministically select one of the sequences for a commit. Such a sequence may be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

based on a static ranking of the sites. Denote rank (P_i) =i and action a_i is generated by process P_i . Consider, for instance, two sequences $<a_1, a_2, a_3>$ and $<a_4, a_1, a_2>$. We select the sequence $<a_1, a_2, a_3>$, if the rank of P_1 is, say, higher than the rank of P_4. In a general case, multiple execution sequences of the same length may satisfy the selection criteria from which one needs to be selected.

We now describe an algorithmic procedure for sifting through various candidate execution sequences derived from S and then determining a committable sequence there-from. The following pseudo-code captures our algorithm:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
type ordered_set compute_exec_seq(S, ej, ℜ);
/* ej is the current epoch state; ℜ is the rule base */
   tempest := compute all possible ordered subsets of elements drawn from S;
   if(|tempest| = 1)
      return (element_of(temp_set));
   ∀y∈tempset
      assign a sequence number sno(y) to element y;
   for (i=1,…..,|tempest|)
      valid[i] := false;
      /*valid[i] = true means ith element of tempest is a committable
        sequence; valid[i] = false indicate otherwise */
   for(each x∈tempset)
      express x in the form <a1,….al>1≤l≤k" ;
      valid[sno(x)]:=true;
      if(l>1)
         int_state := g(ej , a1);
         for(i=2,…,l)
            if(∃r ∈ ℜ| r pertains to int_state)
               parse rule r to determine compatibility of ai with <a1, …, ai-1>
               if(ai is not compatible)
                           valid[sno(x)] := false;
                           break;
               else
                  /* (ai ~ (ai-1 ~ … ~(a2 ~ a1)), …),
                    i.e. it appears as if (∃f associated with r| f(int_state) ≡ ai) */
                  if(∃ a state transition function g associated with r)
                     int_state := g(int_state, ai);
                  else
                     valid[sno(x)] := false;
                     break;
            else
               valid[sno(x)] := false;
               break;
      if(valid[sno(x)]=true)
         if(presentation_time(<a1, ….al>) ≥ Δ) valid[sno(x)] := false;
         break;
   eset := {y∈tempset | valid[y(sno)] := true};
   if(eset ≠ Φ)
      largest_set :={ z∈ eset | (|z| ≥ |y| for y ∈ (eset – z))};
      eseq := compare_seq(largest_set);
      return (eseq);
   else return (NULL);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
type ordered_seq compare_seq(largest_set){
    if(|largest_set|==1) return element_of(largest_set);
    else
        largest_set = largest_set – {x|x is the first element of largest set} –
                                   {y|y is the second elementof largest set} ∪
                                   compare(x,y);

    compare_seq(largest_set);
}
```

The procedure ***compare*** employs the information on site ranking as given by the pseudo-code:

```
type ordered_set compare(seq1,seq2){
    // |seq1|=|seq2|
    seq1=<U1^i> i=1,2,..k''
    seq2=<U2^i> i=1,2,..k''
    if (Ur^i = aj) then weight (Ur^i)= j
    for(i=1,2,.. k''){
        if(weight(U1^i)>weight( U2^i)) {return seq1; break;}
        else if (weight(U2^i)>weight( U1^i)){return  seq2; break;}
    }
}
```

As can be seen, besides the compatibility relations among actions, two additional pieces of information are also used in the algorithm when generating the committable execution sequence. First, a check is made on whether the intermediate states reached during the course of execution of various actions in a candidate sequence fall outside the state space where the rules that applied to the current epoch are no longer valid. In that case, the candidate sequence is truncated at that point. Second, the timing constraints Δ may preclude some of the actions in a candidate sequence from taking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

place due to the passage of time in an execution of the earlier actions. In this case also, the candidate sequence is truncated at that point.

An observation about the possible use of our TAW primitives, alluded to in section 6.5, is in place here. An action, say, *a*, delivered by the TAW layer is already validated for timing constraints under an ordered delivery scenario. Consider the case of a subsequent decision by the epoch layer above to abort *a* due to violation of the timing constraints when *a* needs to be ordered relative to other actions. Such a case may arise in our generalized epoch based implementation as well ---c.f. section 10.4. This is because, the delivery semantics enforced by our timed agreement protocol (described in chapter 9) is identical to that of TAW primitives and subsequent compatibility checks are carried out only on the set of actions delivered by the protocol at all user sites.[17]

---

[17] A case of action *a* not being delivered at all by the TAW layer can only be due to the violation of timing constraints (because of the unordered delivery semantics). This case may also arise in our generalized epoch-based implementation, since *a* will not meet the timing checks carried out by our algorithm in section 10.4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 24 provides a pictorial representation of a run of our algorithm using the earlier example of city construction planning.

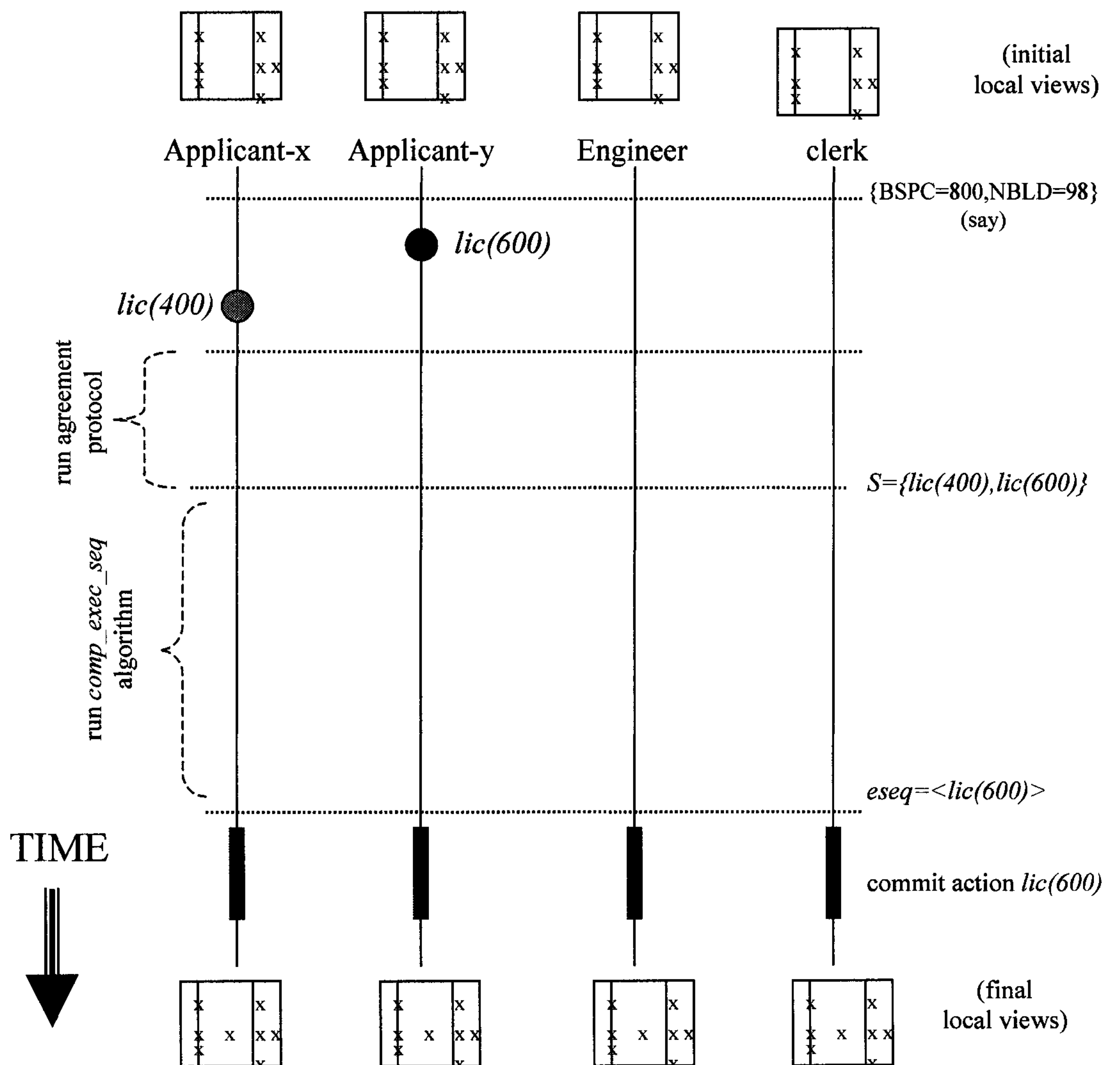


Figure 24: A Sample Run of Our Algorithm (Example of City Planning)

The ranking of sites and the timing constraint Δ is a stable piece of information made available to all the participating sites. Since this information is used in a deterministic

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

manner by all the sites and since the state transition relations are deterministic, there is an implicit agreement on the committable execution sequence generated by various sites.

After so determining a committable execution sequence, the session agent at a site determines the presentation schedule for these actions on the user window[18].

In this chapter, we described the algorithmic procedures needed to have all the sites agree on the execution sequences of compatible actions. These procedures employed various pieces of meta-information supplied by the session manager (such as site ranking and time deadlines)

We now provide a discussion of media-specific segmentation of a window object for multimedia application and introduce the concept of private window as an enabler for message assembly to embody an action.

---

[18] Note that compatibility among the component sub-actions is implicitly guaranteed since they are from the same user. So we do not carry out compatibility checks between the sub-actions of a top-level action.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 11. Sample Applications

In this chapter, we look into some distributed applications that are structured as per our proposed collaboration model. As mentioned earlier (section 9.2), our collaboration layer expects the following pieces of meta-information from the application layer:

- Object specifications.
- A set of permissible actions defined on these objects.
- A set of rules, in terms of objects and actions.

We examine how the above meta-information can be prescribed in some representative applications.

11.1 Distributed Canvas

Let us consider an application that allows participants to draw on a (logically) shared canvas. For such an application:

Set of objects: $O = \{\text{canvas}, \{d_i\}\}$

$\{d_i\}$ is set of segments on which user actions will have effects.

The possible set of actions:

$A = \{\text{insert } (d_i, \text{pos}_i), \text{move } (d_i, \text{pos}_i), \text{del } (d_i), \text{paint } (d_i, \text{col}_i), \text{paint } (\textit{canvas}, \text{col}_i)\}$

Where pos is the spatial coordinate on the canvas where the execution of action is intended. And col represents the desired color.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Possible set of rules:

$R = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$

R_1: {insert (d_i, pos_i) DOES_NOT_CONFLICT_WITH insert (d_j, pos_j)} for $pos_i \cap pos_j = \Phi$

It means that multiple insertions are compatible if they are spatially disjoint.

R_2: {insert (d_i, pos_i) DOES_NOT_CONFLICT_WITH move (d_j, pos_j)} for $pos_i \cap pos_j = \Phi$

It means that insertion of an object and transfer of another is permitted if the final destination is different.

R_3: {insert (d_i, pos_i) DOES_NOT_CONFLICT_WITH paint (d_j, $color_j$)} for $d_i \neq d_j$

It means that insertion and coloring of objects are mutually compatible if they are different objects.

R_4: {paint (d_i, $color_i$), DOES_NOT_CONFLICT_WITH paint (d_j, $color_j$)} for $d_i \neq d_j$

Coloring of objects Re mutually compatible, as long as two users do not want to color the same object.

R_5: {move (d_i, pos_i), DOES_NOT_CONFLICT_WITH paint (d_j, $color_j$)}

Insertion and coloring of objects are mutually compatible.

Aforementioned rules makes sense intuitively, we can come up with more sophisticated rules that tries to capture the aesthetic integrity:

R_6: paint (d_i, $color_i$) CONFLICTS_WITH (*canvas*, $color_j$)

R_7: insert (d_i, pos_i) CONFLICTS_WITH (*canvas*, $color_j$)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When an action of changing the color of the canvas (canvas can be thought of the backdrop against which all the objects get displayed) we should not modify the colors of other objects. The rationale being, the decision to choose a particular color for the backdrop (canvas) may be influenced by the color of the existing objects but, according to our rules, not by their positions, since there is no rule restricting movement of objects while coloring the canvas, hence changing of color of an object may lead to context violation. The same argument can be made about the incompatibility of object insertion and canvas coloring.

11.2 Distributed Gaming

Our model is very well suited to support distributed gaming. Let us consider two sample games:

11.3 Multiplayer Card Game

Multi player card games are among the simplest from a synchronization standpoint because of the inherent serialization embedded into gaming activity. Let us assume that the rules of the game dictate that participants take turns in displaying their cards in a pre-determined order (clockwise or anticlockwise). To make things interesting, let us assume that participants are allowed to fold. These fold actions do not conflict with each other.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We notice that each player takes their turn in playing their hands, thus the playing of each card constitutes an epoch. So, a round will involve going through up to N number of epochs (N being the number of players). The number of epochs can be less than N for a particular round if any player decides to fold, which can be done in parallel to any other action.

Global state O is a composition of card states and player states, thus

$O = \{O_c, O_p\}$

Card states: $O^c = \{o^c_j\} = \{\Phi, o^c_1, o^c_2, \ldots\ldots, o^c_k\}$

Where, $o^c_i = \{\{c_1, c_2, \ldots c_i\}\}$, and $c_i \in D$, where D represents the deck of cards;

(o^c_i represents the set of all possible ways of playing i cards)

let o^{cp}_i represent one particular instance of playing i cards, i.e. $o^{cp}_i \in o^c_i$

Player states: $O^p = \{ps_1, ps_2, \ldots\ldots, ps_N\}$

Where N is the number of participant and $ps_i \in \{0,1,2\}$;

0,1 and 2 representing three possible states: played, folded, not_yet_played accordingly.

Set of actions: $\{a_j\} = \{\text{play}(c,i), \text{fold}(i)\}$;

Where, $c \in D$ and $i \in \{1,2,\ldots,N\}$

Initial state $s = \{\{\Phi\}, \{2,2,\ldots 2\}\}$;

To start with, no card has been played and everybody is in not_yet_played state.

Final set of states ($F \subseteq O$): With the assumption that players can quit at any time, any valid state can be the final state, i.e. $F = O$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Transition functions δ :

$O\,X\,\text{play(c,i)} \equiv \{\, o^{cp}{}_{i-1} \,,\, o^{p}{}_{i-1}\}\; X\,\text{play(c,i)} = \{o^{cp}{}_{i-1}\cup c,\, o^{p}{}_{i}\}=\{o^{cp}{}_{i},\, o^{p}{}_{i}\} \in O$

where, $ps_i \in O^p = 0$ (0 represents played state)

$O\,X\,\text{fold(i)} \equiv \{o^{cp}{}_{i-1} \,,\, o^{p}{}_{i-1}\}\; X\,\text{fold(i)} = \{o^{cp}{}_{i-1} \,,\, o^{p}{}_{i}\}$

where, $ps_i \in o^p{}_i = 1$ (1 represents folded state)

We recognize that at the end of a round a score can be computed based on the played hands, and that score can be carried over to the next round to form a more realistic card-playing scenario.

Our formalization of the game application involves defining a set of objects, actions and set of rules.

Set of objects: $O = \{d_i\}$ where $d_i \in D$, i.e. our objects are cards.

Set of actions: $A = \{\text{play}(d, \text{i}), \text{fold(i)}\}$

And set of rules: $R = \{R_1, R_2\}$

Where,

R_1 : {play(d, i) DOES_NOT_CONFLICT_WITH fold(j)}

Any instance of actions *play* and *fold* are mutually compatible, i.e., they can be executed in parallel.

R_2 : {fold(i) DOES_NOT_CONFLICT_WITH fold(j)}

All fold actions are mutually compatible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We assume a strict turn taking protocol is maintained by the application layer, i.e. only one *play(d,i)*action gets generated per epoch (that is why we do not have any rules dictating the relationship between *play(d, i)* and *play(d,j)*).

Based on the above definitions, an agreement protocol of the collaboration layer can come up with a committable set of actions for each epoch.

11.4 Multiplayer Shooting Game

Let us consider a simple multi-player shooting game where users shoot at each other to achieve the 'last man standing' status. A window is basically the local copy of a globally shared game state representing the player's involvement on display screen. The content of a window is basically the spatial location of various entities corresponding to different users. Thus the window state can be captured in grid format where each grid indicates the presence of an entity. Such a grid can be captured in the form of a matrix $M = [m_{ij}]$. Thus having an entity a at grid (i,j) is capture by $m_{ij}=a$ Thus the game state

Game State: $O = \{\Phi, o_1, o_2, \ldots\ldots\}$

where o = M,

Set of actions: $A=\{a_1,a_2\}$

$a_1 \equiv$ move(x, $pos_{i,j}$)

Where, move(x, $pos_{i,j}$) means moving x to pos having coordinate i,j .

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$a_2 \equiv$ shoot(x, y)

Meaning: x is shooting at y

And finally the set of rules: $R = \{R_1, R_2, R_3\}$

$R_1 \equiv$ {move (x, pos^1) DOES_NOT_CONFLICT_WITH move (y, pos^2)} provided $pos^1 \neq pos^2$

Move actions are mutually compatible provided spatial integrity is maintained.

$R_2 \equiv$ {shoot (x, y) DOES_NOT_CONFLICT_WITH shoot (z,l)}

shoot-actions are mutually compatible. (This rule has some interesting consequences on the game state; it is possible that last two players standing shoot at each other and both being successful in hitting the other, ends up being dead. That is, we are allowing 'no man standing' as a valid final state of our last man standing game).

$R_3 \equiv$ {shoot (x,y) CONFLICTS_WITH move (y, *pos*) }

Shooting of an entity and moving of the same entity is not compatible. The rationale of this rule is, if an entity is shot, it cannot move, which makes sense intuitively.

$R_4 \equiv$ {shoot (x, y) DOES_NOT_CONFLICT_WITH move (z,pos)$_{y \neq z}$ }

Shooting of one entity is compatible with moving of another.

$R_5 \equiv$ {move (x, pos) DOES_NOT_CONFLICT_WITH shoot(y, x)}

Moving of an entity and shooting of the same entity is compatible. The rationale of this rule is, if an entity is has moved, it cannot be treated as being shot. (This results in a missed shot).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

State transition rules δ:

For any shoot action to be valid we must have the following condition true at the time of the generation of action:

$o_i\, X$ shoot(x,y) $\equiv M^i\, X$ shoot(x,y) $\Rightarrow \exists k,l\ m_{k,l}^{\ i} = x$ and $\exists p,r\ m_{p,r}^{\ i} = y$ (x,y must exist on the display)

having satisfied that shoot action induces state transition in the following manner:

$o_i\, X$ shoot (x, y) = M^{i+1} such that $m_{p,r}^{\ i+1} = \Phi$ (y is removed from the display)

Any move action must satisfy the following condition at the time of generation:

$o_i\, X$ move (x, $pos_{q,r}$) $\Rightarrow m_{q,r}^{\ i} = \Phi$ (in order to move to pos, pos must be empty)

State transition induced by move action is:

$o_i\, X$ move (x, $pos_{q,r}$) = M^{i+1} , $m_{q,r}^{\ i+1} = x$

Again, based on these rules, an agreement protocol of collaboration layer can come up with a committable ordered set of actions for each epoch.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 12. Implementation

To see the effectiveness of our proposed architecture, we have developed a multimedia presentation software that embodies our collaboration framework. The software is capable of generating video, audio and draw-clips. It binds different media messages emanating from a single user, with temporal ordering constraints; effectively implementing our notion of composite actions. We used native IP multicast as our group communication primitive, and physical time synchronization was attained through NTP. The design of API involves choosing a suitable service interface through which our control paradigm is made visible to the application developers. Some of the major functional elements that have been incorporated in our architecture are described next (See figure 25).

12.1 Structure of API

Specifying the parameters of timed actions, such as start time of an action, its persistence duration, its response time and its causal dependency on precedent action(s) are specified through API. In addition to these parameters, each message carries media specific parameters such as the encoding format (JPEG, MPEG), display mode (mono/stereo) etc. These parameters guarantee a meaningful display of media messages, and are affixed in messages at the time of action generation by the end users.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

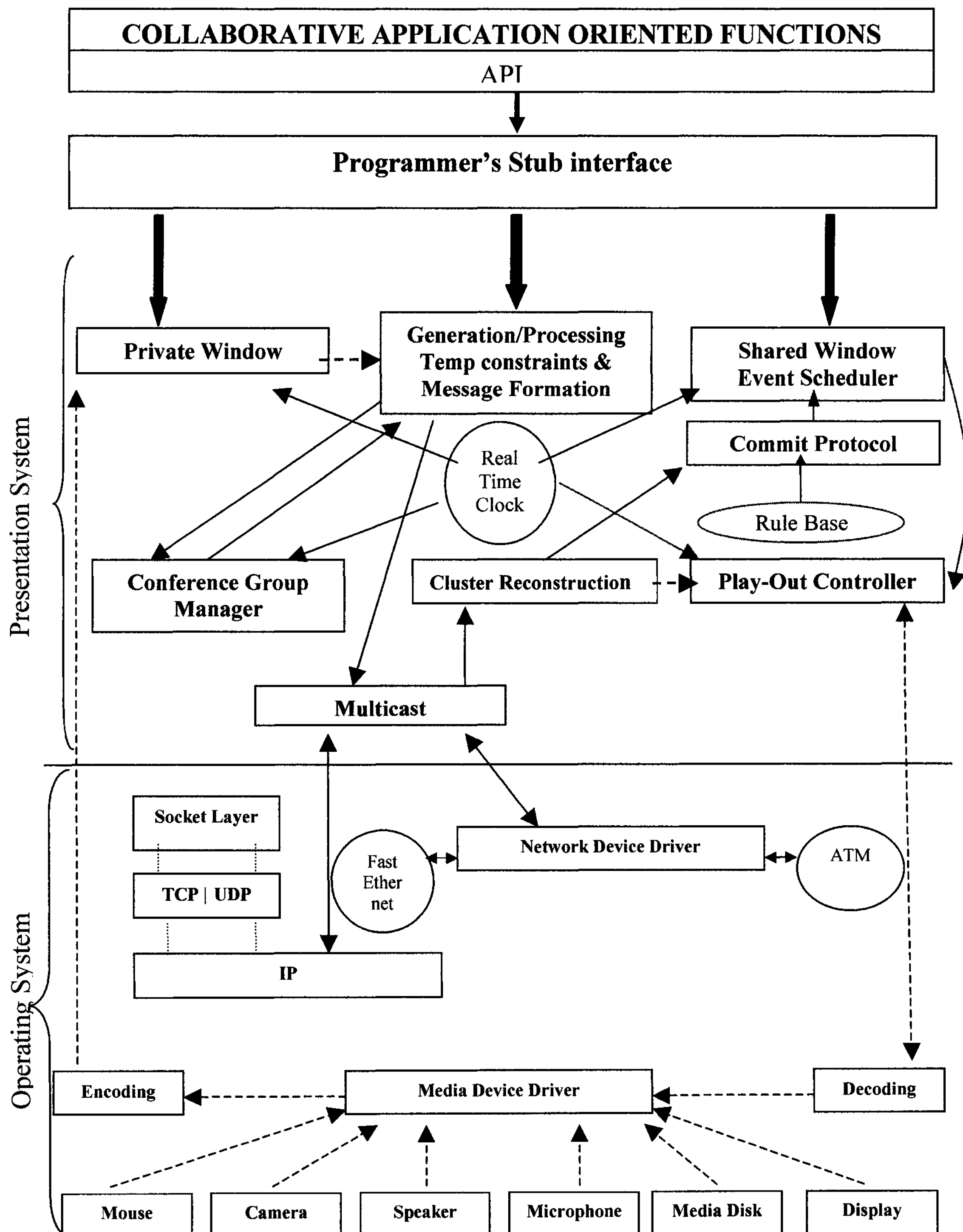


Figure 25: Schematic Diagram of Software Model on a Workstation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

12.2 Private Window

We have pointed out that user actions are generated in response to the global context. An object manipulated by users has its physical implementation as a collection of media specific windows on the user display. To maintain a uniform view of the object state, we do not allow the local generation of a new action to interfere with the displays in the current global view. We keep the message-level processing required to generate a new action as private. This is achieved by creating a separate working area where the user composes his new action based on the global view. This often involves assembling different media clips as part of a response in a multimedia collaboration session. This is the place where a user assembles the various components of an intended action and binds them together with meta-information that captures the dependency among the components.

12.3 GUI for Parameter Specification

We provide a graphic user interface (Figure 26) that enables the specification of relationship between the messages constituting an action. This basically allows us to implement *occurs_after* relationship, as described in section 8.1, in terms of a temporally constrained specification of the relationship that exists between any message and its logical predecessor. For example, an action consisting of a video clip, an audio clip and a draw clip we can enforce *occurs_after* for every pair of sub-actions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Current Msg: Previous Msg:

Least Waiting Time Most Waiting Time

Figure 26: GUI for Parameter Specification

by first selecting the current message from the drop down window, then selecting the corresponding previous message from another drop down window. The real-time constraints are provided through the sliding windows 'least waiting time' and 'most waiting time', allowing only non-negative values.

12.4 Processing of Temporal Constraints

A cluster is a message-level abstraction to carry the various sub-actions of a user-level activity on the shared window. As we have mentioned, user action in a multi-media setting may consists of several messages, e.g., video clip, text information, an audio clip, highlighting as part of a single update action on a document. A message cluster is a programming construct that binds together all the messages that constitute an action along with the meta-information depicting their causal and temporal dependencies. A cluster is basically a set of temporal ordering relationships,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

prescribed using *occurs_after* constructs, on a collection of messages. We denote a message cluster comprising of messages $\{m\}$ as:

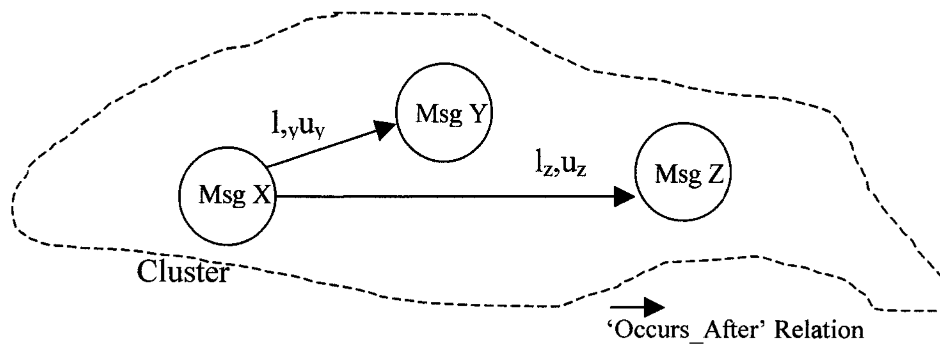
$$
\begin{aligned}
R(\{m\}) \equiv \{ & \; occurs_after(x_1,y_1), \\
& \; occurs_after(x_2,y_2), \\
& \quad \vdots \\
\} & \\
\text{for } x_1, x_2, y_1, y_2, \ldots & \in \{m\}
\end{aligned}
$$

Since a message is the basic unit supported by the window system in terms of which user activities are constructed, the ordering relationship among them is incorporated into the corresponding messages. In the earlier example of shared document editing, text highlighting and voice annotation constitutes a message cluster. These messages cannot be interleaved with messages of activities emanating from other users. The real-time persistence duration of a message cluster *mc* can be derived from that of constituent messages. This may be used to capture the real-time responsiveness of user-level interactions. A message cluster defined as

$\{((y,p_y), Occurs_After(x,l_y,u_y)), ((z,p_z), Occurs_After(x,l_z,u_z))\}$

has the following graphical representation:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



p_i: Persistence duration of message l_i: Least waiting time u_i: Most waiting time

Figure 27: A Message Cluster

The relevance time of a user action Δ, as defined earlier (section 7.1), is used in determining the presentation time of the corresponding messages in the cluster. The Δ itself impacts the relevance time of the component messages of the cluster. As mentioned earlier, Δ can be used by the protocol-agents, executing at user sites, to decide on the presentation times of the component messages[19].

12.5 Media-Level Presentation Committed Actions

Several modules in our software implement the agreement protocol. First, we reconstruct the clusters by parsing the dependency and binding relations carried in the messages received. Each message carries a field identifying the cluster it belongs to and another field indicating total number of messages within its cluster. So, a receiver

[19] For related ideas, we refer to [Tachi'97]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can assemble all the messages pertaining to the same cluster. Also, a dependency graph is created for every cluster to compute the presentation time of the action it incorporates.

After the clusters are reconstructed, they are processed by the commit protocol module. This is where the distributed agreement protocol is implemented ---c.f. section 9.4, section 9.6.

Once an action is deemed as committable, the component messages in the corresponding clusters are processed by a window event-scheduler. The window module is in charge of the actual delivery of media data carried in messages to the media-specific devices attached to the local workstation.

Since our model allows only one action per epoch per user, the presentation schedule on the intra-cluster messages is determined by their causal and temporal constraints. For intra-cluster messages that are independent (no established causal precedence relationship), it is possible for two different sites to play them in different order. This weaker implementation of WYSIWIS preserves 'perception integrity' without compromising the benefits of asynchrony inherent in the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

12.6 Session-Level Group Membership

A session-level group is bound to a multicast address, using which messages can be sent and received. In order to allow dynamic participation of users during a collaboration session, the group membership mechanism can allow participants to join and leave the group while the session is in progress. In our current implementation, we do not allow members to join or to leave a group in the middle of a session (i.e., the members can join the group only at the beginning of a session). This however does not compromise the generality of our model since the issues of dynamic group membership are orthogonal to the issues of user-level co-ordination control that are the main focus of this thesis.

In our implementation, every session begins with a setup procedure where the system-wide session manager supplies a static list of group members to the underlying protocol modules. The member ids are basically UNIX process id's that implement the session-level protocol agents.

12.7 Play-Out of Media Messages

This module interprets the message contents based on the meta-information about device level play-outs carried in the messages. In an example of video clip that is encoded with JPEG compression, the play-out module invokes a JPEG decoder, creates a display window and plays it according to the specified rate. Only after the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

media messages of all committed actions are played-out on the user window, the user is allowed to generate its next action. This is achieved by having the window controller disable the user from sending out the messages in its private window until the current play-outs are completed. The disabling and a subsequent enabling of the user window by the agreement protocol module is based on the current epoch duration.

In summary, the scope of our implementation is limited to a 'proof of concept' of our model of context-aware distributed collaboration. In the future, we plan a more comprehensive implementation with performance engineering of the various protocols and a prototyping of more elaborate applications.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 13. Conclusion

Computer supported collaborative work (CSCW) has evolved over time and has been driven mainly by application-specific research. For example, distributed text editor, whiteboards, multi-player games, and interactive lectures have been extensively investigated. In such real-time distributed applications, multiple user entities interact with one another over a network to manipulate a (logically) shared data space that represents their collective goal. The user entities might be humans participating in a decision-making (e.g., tele-surgery), robots collectively planning a course of action (e.g., mapping out a navigation plan over a terrain), or humans and computer processes interacting with one another in a simulated a physical environment (e.g., tele-immersion), in this thesis, we focused on the system-level coordination control aspects to maintain a consistent and cohesive view of the shared data space among user entities during distributed collaboration activities.

Hitherto, not much effort has been devoted elsewhere to generalize the coordination control issues that arise in real-time collaborative systems. Particularly, the dynamics and interactive nature of collaboration activities have not been systematically considered in formulating a generalized model of coordination control. The lack of a generalized model has resulted in application-specific solutions rigidly built into a collaborative system, and has also hampered the evolution of newer types of applications by restricting the developers' view of the feasible solution spaces. In this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

light, most of the existing techniques for coordination control in distributed collaborations are borrowed from 'database synchronization' world with suitable modifications (namely, transactional serialization using, say, locking or rollbacks). Innovative use of these techniques have allowed for successful collaboration in some special settings, but they fail to address the problem in a generalized manner. Motivated by this observation, our thesis proposed a generalized model of coordination control in distributed collaborative systems.

We recognize that a collaboration is driven by a common goal partaken by the users participating in a joint task. The collective execution of the task is realized through a shared window that captures the task state, i.e., context, at various points in time. Due to the interactive nature of task execution, the user actions are inter-dependent, as driven by the current context perceived by the users through their local copies of window objects.

The distributed execution of a task by remote collaboration may cause the participants' perception of task state (i.e., context) to diverge. This is due to user-level asynchrony and system-level communication delays, compounded by the lack of physical shared member between users. As a result, users may possibly end up generating conflicting actions --- which may in turn lead to a lack of cohesiveness in the collaboration session.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The ability to detect conflicting actions becomes crucial in formulating a generalized model of coordination control for collaboration settings. Existing models of coordination control employ techniques that pre-suppose the serializability of user actions. A premise of our thesis, however, is that, user actions generated (spontaneously) in response to the currently perceived context can conflict with each other. Here, a user action that takes effect on the shared window (i.e., commits), thereby modifying the context, can possibly invalidate the other user actions. For example, a proposal for garden in a city area during city layout planning discussion among architects can invalidate the construction of a factory elsewhere in the city. To deal with potential conflicts among actions, we introduced the notion of action compatibility. It is based on meta-rules that prescribe what type of actions may conflict with one another over a given context space. Based on these rules, the coordination control mechanism can determine the set of actions that can commit on the shared window. Here, we emphasize that contextual integrity should be the basis for realizing distributed collaborations.

Though the notion of compatibility among actions is not entirely new, in existing works, it is defined in terms of the intended effects of actions on the shared window: say, whether two update operations occur on two distinct objects in the window. This view, in turn, allows employing the traditional models of concurrency control on shared data. In contrast, our notion of action compatibility is defined in terms of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

shared context that drives the generation of user actions and their subsequent committals on the shared window. We argued that anything short of an explicit compatibility check on the user actions will restrict the coordination control to a degenerate form that allows only one action to be committed in any round of user interactions. In light of the paradigm shift, existing works may be viewed as enforcing contextual integrity in a degenerate manner by simply committing exactly one user action in each round of interactions and aborting the remaining user actions. Our thesis argued that the ability to commit multiple non-conflicting actions is very useful in enhancing the efficacy of distributed collaborations.

Our coordination control mechanisms are made context-aware by prescribing a set of rules governing the potential conflicts between user actions in various contexts. The context awareness imparts the coordination control protocols with the ability to evaluate the compatibility relations among user actions and determine the set of actions that can commit. An object-based realization of our window system allowed us to incorporate the notion of action compatibility and realize the underlying context-aware coordination protocols.

Using our notion of action compatibility, we proposed a weaker form of WYSIWIS for achieving perceptive consistency. We argued that it is possible for two actions to be mutually compatible in such a way that ordering them differently does not lead to a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

divergence of perception among users. Thus, our model provides perceptive consistency in the presence of asynchrony in user actions and system-level communication delays.

To realize our model of context-aware coordination control, we first identified suitable programming constructs for use by collaborative applications, and then designed the underlying coordination protocols to realize these constructs. Our formalization of action compatibility naturally allowed for the grouping of actions that share a common context. Grouping of actions based on their context led us to the concept of temporal epochs. Basically, an epoch is the time granularity over which the user actions are perceived to have taken place in transforming the window from one consistent state to another[20]. An epoch-based coordination control allows for the committal of more than one non-conflicting actions --- which is a major improvement over existing works.

The thesis tied up the notion of epochs with object states and passage of time that occur during distributed collaborations. We proposed a software architecture that allows implementing epochs as part of a distributed collaboration system. Here, a collaborative application may be specified in terms of objects and actions, along with the rules governing the ordering relationship between actions. In a sense, we enforced

[20] Modeling user actions as a function of the shared context frees us from the burden of inferring a causal relationship among the actions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a relaxed consistency on the shared window objects within the application-specific context, while providing user-level responsiveness at the granularity of epochs. It may be emphasized here that consistency and responsiveness are not orthogonal properties in a real-time distributed collaboration.

For a system-level implementation, we incorporated temporal constraints on user actions. These constraints are prescribed in two forms: i) a relevance time for actions and real-time play-out duration for such actions; ii) an ordering of the actions that is determined by the compatibility relations between them. Given that user actions can be multimedia-based, we modeled an action as a hierarchy of sub-actions where each sub-action is associated with a component media. The ordering of sub-actions within an action however does not violate the contextual integrity that is enforced at the top action level. For the component sub-actions, we implemented a media-level causal ordering.

We have tested our concepts by implementing a multi-media presentation software, that allows users to distribute multimedia content, e.g., audio, video and draw clips in timely manner. We maintained object level consistency at all sites by enforcing a weaker form of WYSIWIS. The latter was realized by a family of distributed agreement protocols parameterized by the timing constraints associated with the actions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The most important contribution of our research is our success in bringing context awareness in the underlying collaboration protocols. We provided a system-level structure that can be readily implemented: epochs, timed actions, agreement protocols, as part of our programming framework to realize coordination control. We also contributed by extending the existing notion of intention violation and introducing the concept of 'contextual integrity'.

Developing a generic middleware architecture that allows the application-level specifications to be plugged into the coordination protocols and implementing a shared window software system using this architecture is part of our future research.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

Abad'94 Martin Abadi, Leslie Lamport. An Old Fashioned Recipe for Real Time. *A CM Transactions o n P rogramming L anguages a nd Systems, 1994, Vol. 16, No. 15. Pages 1543-1571.*

Afek'94 Y. Afek, H. Attiya, A. Fekete, M. Fischer, N. Lynch, Y. Mansour, D. Wang, L. Zuck. Reliable Communication Over Unreliable Channel. *Journal of the ACM, Vol. 41, No. 6. Pages 1267-1297.*

Bald'02 Roberto Baldoni, Michel Raynal. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online, Volume 3, Sepetember 2002.*

Bent'94 R. Bentley, T. Rodden, P.Sawyer, I. Sommerville. Architectural Support for Cooperative Multiuser Interfaces. *IEEE Computer May 1994 ,Vol. 27, No. 5. Pages 37-46.*

Birm'87 Kenneth P. Birman, Thomas A. Jo seph. R eliable C ommunication i n the presence of failures. *ACM transactions on computer systems, Vol. 5, No.1, February'87 pages 47-76.*

Birma'87 Kenneth P. Birman, Thomas A. Joseph. Exploiting Virtual Synchrony In Distributed Systems. *Proceedings of the eleventh ACM Symposium on Operating systems principles. Austin, Texas, United States Pages: 123 – 138.*

Bore'93 Nathaniel S . B orenstein. M IME: A Portable and Robust Multimedia Format for Internet Mail. *Multimedia Systems Journal 1993, Vol.1, No.1, Pages: 29-36.*

Boui'04 Nicolas Bouillot, Eric Gressier-Soudan. Consistency Models for Distributed Interactive Multimedia Applications. *Operating Systems Review, October 2004.*

Casn'92 S. Casner, S. Deering. First IETF Internet Audiocast. *ACM Comp. Communication Review, Vol. 22, No.3.*

Deer'88 S. Deering. Multicast Routing in Internetworks and Extended LANs. *Proceedings of the ACM SIGCOMM, pages 55--64, August 1988.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Diot'99 C.Diot, L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Networks magazine, vol. 13, no. 4, July/August 1999.*

Domm'97 H.P. Dommel, J.J. Garcia-Luna-Aceves. Floor Control for Multimedia Conferencing and Collaboration. *Multimedia Systems (ACM/Springer), Vol. 5, No. 1, January 1997.*

Dwor'88 C. Dwork, N. Lynch, L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the Association of Computing Machinery, Vol.32, No. 2, April 1988, Pages 288-323.*

Elli'89 C.A. Ellis, S.J. Gibbs. Concurrency Control in Groupware Systems. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data. Portland, Oregon, United States. Pages: 399 - 407.*

Elli'91 C.A. Ellis, S.J. Gibbs, G.L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM, January 1991, Vol. 34, No.1.*

Elno'96 E.N. Elnozahy. D.B. Johnson. Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *October 3, 1996. CMU-CS-96-181. School of Computer Science. Carnegie Mellon University. Pittsburgh, PA 15213.*

Fisc'85 M.J. Fischer, N. Lynch, M.S. Paterson. Impossibility of Distributed Consensus With One Faulty Process. *Journal of the Association of Computing Machinery, Vol.32, No.2, April 1985. Pages 374-382.*

Felb'01 P. Felber. F. Pedone. Probabilistic Atomic Broadcst. *Technical Report, Bell Labs, Lucent, Dec. 2001.*

Gaje'95 H.Gajewsaka, J.Kistler, M.S. Manasse and D. R. Redell. Argo: A system for distributed collaboration. *Tech. Report,DEC Systems Research Center, Palo Alto (CA),1995.*

Hags'90 Olof Hagsand. A Multi User Draw Editor. *Proceedings of 1st MultiG workshop, Stockholm, Nov 1, 1990.*

Lai'88 Kum-Yew Lai, Thomas W. Malone, Keh-Chiang Yu. Object Lens: A "Spreadsheet" for Cooperative Work. *ACM Transactions on*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Information Systems (TOIS), Volume 6, Issue 4 (October 1988). Pages: 332-353.

Lamp'78 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *ACM Communication Vol.21, No.7 (July'78). Pages 558-565.*

Lauw'90 J.C. Lauwers, T.A. Joseph, K.A. Lantz, A.L. Romanow. Replicated Architectures for Shared Window Systems: A Critique. *ACM SIGOIS Bulletin Volume 11 , Issue 2-3 (April 1990) Pages: 249-260.*

Lela'88 M.D.P. Leland, R.S. Fish, R.E. Kraut. Collaborative document production using quilt. *Proceedings of the 1988 ACM conference on Computer-supported cooperative work, Portland, Oregon, United States. Pages: 206- 215.*

Li'04 F.W.B. Li, L.W.F. Li, R.W.H. Lau. Supporting Continuous Consistency in Multiplayer Online Games. *Proceedings of IEEE MM'04, October 10-16, 2004,New York. Pages 388-391.*

Lisk'88 Barbara Liskov. Distributed Programming in ARGUS. *Communications of the ACM, March '88, Vol. 31, No. 3. Pages 300-312.*

Minz'91 Minzer S. A Signalling Protocol for Complex Multimedia Services. *IEEE Journal on Selected Areas in Communications, Vol., No. 9, pages. 1383-1394, December 1991*

Muns'96 J. Munson and P. Dewan. A concurrency control framework for collaborative systems. *ACM, CSCW,'96. Pages 278-287.*

Paza'94 Paul Pazandak, J. Srivastava. A Multimedia Temporal Specification Model and Specification. *Technical Report 94-33, University of Minnesota.*

Ravi'93 K. Ravindran, A. Thenmozhi. Extraction of Logical Concurrency in Distributed Application. *International Conference on Distributed Computing Systems, IEEE-CS, Pittsburg, May, 1993.*

Ravi'04 K. Ravindran, A Sabbir, K. A. Kwiat. Extended Atomic Write: Primitives to Incorporate Timeliness of Data in Distributed Embedded Systems Programming. To apper in *International Journal of Embedded Systems, Inderscience Publishers, 2005.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ravin'04 K Ravindran, Ali Sabbir. Event-based Programming Structures for Multimedia Information Flows. *In proc. MMNS 2004 (Management of Multimedia Networks and Services), San Diego, CA, October 3-6, 2004.*

Sabb'04 Ali Sabbir, K Ravindran. User assisted Tools for Concurrency Control in Distributed Multimedia Collaborations. *In proc. ACM MM'2004 (ACM Multimedia), New York, October 2004.*

Schm'93 C. Schmandt. Phoneshell: The Telephone as Computer Terminal. *Proceedings of 1st ACM conference on Multimedia. Pages 373-382.*

Scho'94 Eve M. Schooler. Conferencing and Collaborative Computing. *Proceedings of the Dagstuhl International Workshop on Fundamentals and Perspectives on Multimedia Systems, pp.175-208, Dagstuhl, Germany (July 1994).*

Stei'90 R. Steinmetz. Synchronization Properties in Multimedia Systems. *IEEE Journal on Selected Areas in Communications, Vol. SAC-8, No.3, pages 401-412.April 1990.*

Sun'96 C. Sun, Y. Yang, Y. Zhang, D. Chen. A consistency model and supporting schemes for real-time cooperative editing system. *Proc. 19th Australian computer science conference, Melbourne, Australia, jan31-feb2'96. Pages: 582-591.*

Tachi'97 T. Tachikawa, M.Takizawa. Delta-Causality in Wide-Area Group Communication. *International Conference on Parallel and Distributed Systems (ICPDS'97), December 11-13, 1997, Seoul, Korea.*

Tana'02 S. Tanaraksiritavorn, S. Mishra. Evaluation of Gossip to Build Scalable and Reliable Multicast Protocol. *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02), October 11-16, Fort Worth, Texas.*

Vin'91 H. M. Vin, P. T. Zellweger, D. C. Swinehart, P. V. Rangan. Multimedia Conferencing in the Etherphone Environment. *IEEE Computer. Vol. 24, No.10. Pages 69-79.*

Voge'01 J. Vogel, M. Mauve. Consistency control for distributed interactive media. *Tech Report, Praktische Informatik IV, University of Mannheim, Germany'01.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Yava'92 Raj Yavatkar. MCP: A Protocol for Coordination and Synchronization in Multimedia Collaborative Applications. *International Conference on Distributed Computing Systems. 1992. Pages 606-613.*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.