CONSISTENCY IN COOPERATIVE EXECUTIONS

VASANTHA LAKSHMI ADLURI

Consistency In Cooperative Executions

by

Vasantha Lakshmi Adluri

A thesis submitted to the School of Graduate Studies
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Memorial University of Newfoundland

July 2002

St. John's Canada

Abstract

Computer-Supported Cooperative Work (CSCW) is the study of how computers can be used to help people work together. Cooperative work on shared, persistent data requires computing system support to coordinate the work of multiple users and to ensure data consistency. Attempts to extend the traditional concepts of transactions and serializability to specify consistency of cooperative executions have largely been unnatural and unsatisfactory.

In this thesis, a new approach is presented to specify consistency of cooperative executions. It is based on an intuitive notion of legality of the read operations. Five legalities, each capturing a different notion of 'recentness' of the values, with respect to a defining relation are explored. They are stated formally in terms of system executions in shared read/write variables. A cooperative execution is consistent in a strong sense when all reads obey all legalities. By relaxing the legality requirements, and also by choosing different defining relations, a large variety of (weaker) consistencies can be specified in a hierarchical manner.

We also give detailed algorithms for ensuring the various legalities. The algorithms correspond to three different environments - centralized, distributed, and mobile agent setups. We illustrate some examples where the legalities can be employed in various aspects of cooperative work.

Keywords: Consistency, Legality, Cooperative Executions, Mobile Agents.

Acknowledgements

I wish to express my thanks to my advisor, Dr. Krishnamurthy Vidyasankar, for his continuous guidance and suggestions. He always been a constant source of inspiration.

I would like to thank the systems support staff and the administrative staff for providing help and assistance during my program. I would like to thank my fellow graduates for their cherished comradeship, suggestions, friendly ears, timely distractions for the past few years.

I would like to thank my husband for his inspiration and spending hours together talking to me. Finally I would like to thank my lovely daughter for her patience over the last three years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context for the Thesis

This thesis deals with Computer-Supported Cooperative Work and consistency in cooperative executions. In particular, internal consistency of a cooperative execution is dealt with. The different levels of system support that can be provided in a cooperative application system are investigated.

1.2 Computer-Supported Cooperative Work (CSCW)

Computer-Supported Cooperative Work (CSCW) is the study of how computers can be used to help people work together. The recent progress of Computer-Supported

Cooperative Work has been fostered by constant improvements in base technologies (e.g., computer hardware, software and network infrastructures) and changing requirements. This resulted from an environment, growing in complexity and dynamics, that surpasses the capabilities of a single individual and demands a group work. The evolution of computing systems went hand in hand with the evolution of organizational work styles. Computing systems evolved from mainframe systems (which offered primitive collaborative applications like shared calendaring systems), to networked personal computers (PC) (which brought dissemination of computing power). Organizational work styles changed from a hierarchical, monolithic and rigid form of cooperation to flatter organizations and increased division of labour within and between companies. This trend has been reflected in a changing focus of many computer scientists from single-user applications to office automation systems and, later on, to CSCW and groupware systems.

Commercial CSCW products are often referred to as examples of *Groupware*. This term is frequently used almost synonymously with CSCW technology. The popularity of the CSCW technology is also evidenced by an increasing number of commercial products, such as Lotus Notes, Microsoft NetMeeting, and CoolTalk in Netscape Communicator. Many of these systems have been successfully used, usually in small groups, to facilitate data sharing among distributed participants. To this date, two main classes of groupware have been identified: *asynchronous* and *synchronous*[45]. The former class, consisting, for example, of e-mail and organizational memory systems, is clearly the most successful. Synchronous groupware is often called *desktop conferencing* applications [45]; examples include collaborative

writing/drawing/design tools, group decision support systems, and games.

An important research area in information systems is computerized support for cooperative users, where those users may be either humans or computers. The act of cooperation implies a means of communication. Further, the individual users need to *cooperate* and also *collaborate* to reach a common goal. Moreover, the users need to have access to information, both actual and historical, as well as have support for searching, sorting and selecting information from large repositories.

The design of Computer-Supported Cooperative Work (CSCW) systems involves a variety of disciplinary approaches, drawing as much on sociological and psychological perspectives on group and individual activity as on technical approaches to designing distributed systems.

Cooperative work on shared, persistent data requires computing system support to coordinate the work of multiple users and to ensure data consistency. The system support can range from loosely coupled collaboration such as electronic mail, to tightly coupled, real time collaboration support such as shared drawing or writing systems. Systems must deal with multiple workers, working in groups with possibly dynamically changing membership, different degrees of coordination and interaction, and diverse perspectives and conceptions of the shared work. This implies that different levels of support are required depending on the tasks and groups involved, and that systems must adapt to changes in tasks and groups.

Systems in cooperative work require the construction of applications which support interaction by multiple users. These applications exploit multi-user interfaces to promote cooperative work by a group of users. Users may be distributed across a number of locations and the associated interfaces run across a number of workstations (Networked or PC's). The need to support user interface execution in a distributed environment has resulted in combining the interests of user interface software and distributed systems.

The objective of a collaborative environment is to facilitate team working and, in particular, to enable a group of persons to manipulate shared objects, and modify them in a coherent manner. Maintaining consistency of objects produced during cooperative activities is an important issue in this environment. Different applications require different levels of consistency. There is a need to ensure consistency for both the work of a single user as well as the cooperative effort. Cooperating users may require information to be presented in a variety of formats corresponding to different levels of sharing. Since information usage is context dependent, cooperative systems must provide multiple views to the users in the group.

In support of this requirement, the present study concentrates on the provision of different levels of system support for cooperative executions. A new approach to specify consistency of cooperative executions is presented. Some types of consistencies explored in cooperative applications are Operation Transformation Scheme, Consistency Guarantees, Application defined consistency criteria, History Merging, some relaxed forms of Serializability, COO-Serializability (COO-SR), etc. Our ap-

proach is based on the assumption that if the values read by a *read* operation are consistent, then the values written in the *write* operation are also consistent. Ensuring the consistency of the *read* operations is the main idea of our approach. This is done by defining five different *Legalities*, each based on the values the *read* reads and each captures a different notion of 'recentness' of the value. These different notions of recentness can be related to different levels of system support that can be provided by our approach.

1.3 Structure of the Thesis

Chapter 1 gives introduction to Computer-Supported Cooperative Work (CSCW), and describes the context of the research reported in this thesis. Chapter 2 gives a brief description about the properties, characteristics, and classification of CSCW applications. A brief description of some of the existing systems developed for cooperative applications is also given. Chapter 3 gives a brief description about consistency and some of the consistency issues dealt in CSCW applications. In Chapter 4, a general definition of the legalities is given. Based on this general definition of legalities, two defining relations are explored: 1) *real-time order* ($\rho = \rightarrow_t$), ii) *causal order* ($\rho = (\rightarrow_i \cup \rightarrow_{rf})^*$). Detailed algorithms ensuring the five legalities with respect to the two defining relations and correctness proofs are also given in Chapter 4. Chapter 5 gives the algorithm for a different environment where mobile agents are involved. Chapter 6 gives the conclusion and examples describing possible scenarios where legalities described in Chapter 4 can be effectively employed in cooperative work.

Chapter 2

Survey of Cooperative Systems

Since the early 1980s, more and more *personal computers* or desktops have become available at work place. As a desktop tool for individuals, the personal computer initially provided services for helping a single user with his/her work. The most important application classes were databases, word processors, graphic tools and spreadsheets. In the 1990s these individual workstations became more and more wired together in local and wide area networks (LAN and WAN). The interconnection of computers was first used for distributed computation and data exchange. The next logical step is not only to connect programs that are running in different computers but also to connect the users themselves [50]. The efforts are intensified by the emerging need for people to work in teams that are locally dispersed.

The research area that is concerned with computer support for collaborating teams is called *Computer-Supported Cooperative Work (CSCW)*. CSCW is not a self-contained research area with its own technology. It is an interdisciplinary approach where the main issue is to integrate different technologies in order to sup-

port collaborative work. The following disciplines are part of CSCW, among others: communication technology, distributed systems, user interfaces, man-machine interaction, artificial intelligence and several other associated aspects such as sociology and organizational theory [8]. While CSCW is the name to the entire subject, the term *groupware* specifically stands for software solutions implementing CSCW. The term usually refers to a huge class of computer software systems which cannot be strictly distinguished from other classes of software systems.

2.1 The Characteristics of CSCW Applications

All applications, interfaces and tools rely, to a great extent, on the underlying services provided. Some of the characteristics of CSCW applications and services provided are [6]:

- Awareness,
- Information sharing,
- Dynamic change and flexibility,
- Multiuser conversion,
- Open infrastructure,
- Models of the real-world,
- Alternative models of control,

- General group mechanisms, and
- Explicit mechanism and policy separation.

These characteristics represent a set of issues and concerns which prevail in existing CSCW applications. Each of the above addresses a different aspect of the support of the cooperative applications. However, the effect of inadequately supporting one particular aspect may influence the successful provision of another. Some of the properties of a CSCW-System are:

- Cooperation
- Communication
- Coordination
- Distribution
- Groupwork
- Group Awareness
- Shared Resources
- Lightly Structured Work

2.1.1 Awareness

The desire for awareness can be found in a number of different aspects of group work and across several layers of a system architecture. Indeed, four out of the five

primitives identified by Dewan and Choudhary [6] involve awareness. For CSCW applications and services the basic awareness support should include:

- *Awareness of the actions of individuals*

  Consider two people, A and B who are remotely editing a document at the same time but in different portions of the document. A deletes an entire section of the document. Without any collaborative awareness B would notice the change only when he attempts to view that part of the document. Clearly, such a change may be important and B should have the option of being aware of A's actions. This type of awareness can also be called *group awareness.*

- *Awareness of the current status of the cooperating individuals* [47]

  In a co-located team, members typically learn from a wide range of cues about the activities of the other members, about the progress in the common task and about subtle changes in group structures and the organization of the shared task environment. Most of this group awareness is achieved without overhead effort. A distributed (virtual) team - even if its cooperation is based on a state-of-the-art groupware system - today is far from a similar level of awareness and opportunity for spontaneous, informal communication. This reduces the effectiveness of the joint effort, and makes cooperation a less satisfying experience for the team members.

  This concept is researched in the projects TOWER [47] and NESSIE [48]. TOWER provides awareness of collaborative activities of team members and their shared working context through symbolic presentations in a Theatre of Work. The aim is to enhance distributed teams with group awareness and

spontaneous communication capabilities close to those of co-located teams. NESSIE provides an awareness environment for cooperative settings which enables new ways to foster task-oriented and social awareness.

- *Awareness of the current state of the cooperation*

  In order to function successfully within a collaboration, an entity (user, application, process, etc.) must actually know what it should be doing. This knowledge is dependent on the current state of the cooperation itself. Therefore any changes in the cooperation will affect what the members of the cooperation should do.

- *Awareness of the state of the underlying system*

  If a shared object being used by a number of system entities fails, then all the collaborators should be automatically notified of the failure, as it may directly affect their subsequent actions. If notification does not occur, then the entities involved in it will be inevitably affected. Therefore, it is important that some feedback from the supporting services and system is available to CSCW applications.

- *Awareness of information ownership*

  Information may be unreliable, influenced by external factors and hence each participant must be aware of the identity of the originator of the given unit of information. Presentation of information ownership is an important factor in any cooperative work.

Closely connected to shared resources is group awareness which differs in certain significant traits from other approaches of computer supported work. Since lightly

structured work (which cannot be easily formalized) has to be supported, there must be a concept of coordinating this work. Group awareness lets the group members see in which tasks or data the other users are involved and take appropriate actions to coordinate their work (and thus collaborate). So the awareness of the other people's existence enables the work to be carried out with low conflict complexity and in a highly cooperative way.

2.1.2 Information sharing

Aspects of information use and representation are fundamental to CSCW. By its very nature, most cooperation between people takes place through an exchange of information. Cooperation relies on people sharing information (ideas, files, processes, etc.) Mechanisms for information sharing should work to actively facilitate the seamless cooperation between individuals and not restrict them with applications where sharing information is cumbersome.

The control of concurrent access to shared information is particularly important in supporting group working where a number of users may need to simultaneously access information. The form of concurrency control mirrors the style of cooperative interaction being supported. Concurrency control in groupware is needed to resolve conflicts and to perform tightly coupled group activities. Concurrency control is the activity of coordinating the potentially interfering actions of processes that operate in parallel. To share information, it must be presented to the user so as to be readily available. Access control is required to make sure that the information is not shared in unexpected and detrimental ways. Access control makes sure that the policies and rights on data objects attributed to the different processes are not

violated. Support for cooperative work needs to focus on sharing and to consider the dual issues of access and concurrency collectively.

2.1.3 Dynamic change and flexibility

Dynamic change and flexibility are closely related to the need for awareness. Group work and related activities can change rapidly in significant ways. An example of such change can be due to people leaving groups unexpectedly because of illness or because of their moving between different cooperative activities or work groups. The manner in which they interact with the applications and how these applications respond to that change are important.

2.1.4 Multiuser conversion

As more and more complex systems are built out of different components, the desirability of software reuse is very important. Reusing existing modules, objects and code can greatly speed up the development of any system. It is advantageous to view single user systems as building blocks for various parts of a CSCW application. Supporting CSCW applications and tools should provide some means of exploiting or integrating existing single-user applications and services.

2.1.5 Open infrastructure

Given the variety and diversity of possible CSCW applications, it is unlikely that a single service/environment will be sufficient to support every application requirement. Therefore, any part of the application's supporting infrastructure should be

sufficiently open to allow the information it contains to be used by other parts of the infrastructure and not to force an application to rely on any single piece of support.

2.1.6 Models of the real-world

Given that cooperative applications and services are intended to support the actual work of groups in real-life, it is important that some support is provided to aid applications and services in this necessary modeling. Failure to provide for this need forces all the CSCW applications to build their own versions of the real-world from scratch. Poor models of the real-world will have a tremendous impact on how useful an application is. In fact it is the core feature upon which real-life cooperation can be modeled within the system.

2.1.7 Alternative models of control

Many different types of control are imposed by applications on users and services. CSCW applications may provide control over the cooperation being supported to guide users towards a goal or conclusion. Each strand of control is imposed to some extent on the applications and services it supports. Because of the nature of CSCW (the large number and variety of possible applications), it would be very difficult for many of the applications and services to rely on a single form of control in order to successfully coordinate a group of people. Hence different type of roles and policies are assigned to different users at different levels in the system depending upon the requirements of the application. A wide variety of alternative models of control at

each level in a system (interface, application, service, etc.) is needed if cooperative services and application are to support, rather than constrain, any significant range of group activities.

2.1.8 General group mechanisms

CSCW applications contain general "group" mechanisms which are employed to support the semantics and the various rules of the application. Because of the range and sheer number of applications, very similar mechanisms are often developed side by side. What these applications require is access to a suite of very general and tailorable mechanisms which support commonly used concepts.

2.1.9 Explicit mechanism and policy separation

If any application or service is to be readily susceptible to change, the policy which the application embodies must be separated from the set of mechanisms which it drives. Separation of these policies concerns the developers to focus on the semantics of the cooperation being supported without having to unravel the mysteries of the mechanisms which perform them. This is one of the most desirable characteristics for any application and support services across the field of practical computer science. In the domain of CSCW, it is not only desirable but essential. Without this separation of concerns, many of the other needs (tailorability, alternative models of control, multiuser scaling, etc.) become unattainable.

2.2 Classification of CSCW systems

The essential precursor to the study of collaborative systems is the definition of a mechanism for classifying such systems. Two principal characteristics are common to all cooperative systems [46]:

1. *THE FORM OF INTERACTION (SYNCHRONOUS VS. ASYNCHRONOUS)*
   Creative problems, such as those tackled by brain-storming (for example, in teleconferencing, etc.,), require group members to cooperate in a *synchronous* manner since the creative input of each group member is required to generate strategy for tackling the task. In contrast, perspective tasks have a previously formulated solution strategy where group members take on particular roles and work in an *asynchronous* manner often without the presence of other group members.

2. *THE GEOGRAPHICAL NATURE OF THE USERS (REMOTE VS. CO-LOCATED)*
   Computer support for group interaction has traditionally considered the case of geographically distributed groups who work asynchronously with each other. More recent research is aimed at the support of face to face meetings. This division is as much logical as physical and is concerned with the accessibility of users to each other rather than their absolute physical proximity.

For reviewing the variety of systems that emerged under the heading groupware, the following taxonomy may be useful. It classifies groupware into several cate-

gories based on application-level functionality [8]:

Message Systems: These systems represent the largest class of cooperative systems. They have evolved from electronic mail programs. As wide area networks designed to support computer communication became more widespread [29], electronic mail systems increased in complexity and functionality. Since the proliferation of such systems has led to the information overload phenomenon [16], recent systems often provide help for users in structuring, filtering and pre-processing incoming messages (one example is INFORMATION LENS [26]).

Group Decision Support Systems and Electronic Meeting Rooms: Group Decision Support Systems (GDSSs) provide computer-based facilities for exploration of unstructured problems in a group setting. The aim of these tools is to improve the productivity of decision-making meetings, either by speeding up the decision-making process or by improving the quality of the resulting decisions [22]. Often, GDSSs are implemented as electronic meeting rooms. One example is the CATeam room and the GROUPSYSTEMS software of the University of Hohenheim in Germany [9]. A typical automated meeting room consists of a conference room furnished with a large screen video projector, a computer (or network of computers), video terminals, a number of individual input/voting terminals, and a control terminal. The computer system supporting the meeting often makes use of multi-user software based on some form of analytical decision technique.

Computer Conferencing: Computer conferencing systems are also related to electronic mail programs. However, the principles are different in that: (i) they impose a structure based on how messages are grouped, (ii) they store information in a central database. The computer serves as a communication medium in a variety of ways. It supports both asynchronous and real-time (synchronous) conferencing. A well known example for asynchronous systems is USENET NEWS. A typical computer conferencing system consists of a number of groups (called conferences), each of which has a set of members and a sequence of messages. Conferences are often arranged so that they individually address a single topic and users subscribe to conferences of interest.

The development of reliable high speed communications has lead to the emergence of new *real-time conferencing* systems. These allow conference members to communicate in real-time. *Multi-media conferencing* systems represent the introduction of a new technological development into conferencing systems. As computer systems become more powerful, their capability to handle wider classes of data increases. This has led to multi-media systems which integrate audio, text and video.

Multiuser Editors: Members of a group often work on data concurrently. Multiuser editors provide help for exchanging data, notifications and for avoiding or resolving conflicts emerging from concurrently accessing the same data [10, 14, 19, 23, 25]. Real-time group editors allow a group of people to edit the same object at the same time. The DistEdit system [21] tries to provide a toolkit for building and supporting multiple group editors.

Coordination Systems: Coordination Systems address the problem of "integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal" [40]. Coordination systems address this problem in a variety of ways. Typically these systems allow individuals to view their actions, as well as the relevant actions of others, within the context of the overall goal. Coordination systems can be categorized by one of the four types of models they embrace: form, procedure, conversation, or communication-structure oriented. Examples for coordination systems are electronic circulation folders [20], workflow management tools [35], Coordinator [5, 11], etc..

Co-Authoring and Argumentation Systems: Co-Authoring and argumentation systems are a general class of systems which aim to support and represent the negotiation and argumentation involved in group working. The cooperative authoring of documents is demonstrative of this class of cooperation where the final generation of a document represents the product of a process of negotiation between authors.

In most of the above categories, there are several overlapping characteristics. It is often not possible to say that one real-life system belongs exclusively to exactly one category. Hence one has to give a list of categories or just categorize by the primary emphasis and intent of an application.

In the remaining of the chapter, a brief description of some of the specific tools developed for Multiuser editors and Co-Authoring systems is given.

GROVE: GROVE [8] is a real-time outline editor allowing several people to edit at the same time. Users can enter and leave a GROVE 'session' at any time. Within the GROVE session, each user has his or her own workstation and bitmap display. Each user in the session knows via 'group windows' which other user is participating. The users in the session can see and manipulate one or more views of the text being worked on. GROVE separates the concept of a view from the concept of a viewer. A *view* is a subset of the items in an outline determined by the read access privileges. A *viewer* is a group window for seeing a contiguous subset of a view. GROVE views and viewers are categorized as private, public or shared. The differences are in the read access for the different users. A *private view* contains items which only a particular user can read, a *shared view* contains items readable by an enumerated set of users and *public view* contains items readable by all users. When users enter (or reenter) a session, they receive an up-to-date document unless they choose to retrieve a previously stored version. The default mode in GROVE is a mode where everybody can read and edit everything, i.e., there is no locking. The authors report that (after a learning period) this does not give a chaotic situation, but appears to be quite useful.

PREP: The focus in PREP [30] is on enhancing the effectiveness of loosely-coupled collaboration. The information in a document is defined as one or more columns and a column is composed of "chunks." PREP columns are used for the main text

of a document, for document plans, for a particular co-author's or commenter's annotations, for request for clarifications or responses to comments. The chunks can contain text, grids, trees or arbitrary images. They are shared by the users in a database. An important issue in PREP is the possibility of commenting. A user can define 'drafts'. A draft defines an area in the workspace that an author intends others to access and consists of a sparsely filled grid of chunks. Each column in a grid is used to store different workspace content such as document content, plan and comments. Collaboration occurs then by commenting on drafts. The authors of PREP find it also important to allow revisions of drafts to exist as distinct versions such that old information is not lost.

The PREP editor is basically asynchronous, although it supports simultaneous editing and commenting of copies of a document through merging. An underlying node-link architecture supports the merging of comments from multiple reviewers, allowing simultaneous display of annotations from several distributed commenter's to whom copies of the document were passed. Recipients can modify their copy of the original draft. A flexible difference finding algorithm allows one writer to see "at a glance" what has changed between the original and the copy (or any two arbitrary versions) and to decide whether to incorporate the change into the current version [33].

SEPIA: The SEPIA [41], [42] system is a hypertext authoring environment using a hypertext database (Hyperbase) constructed on top of a commercial relational database. SEPIA supports synchronous and asynchronous collaborative editing of hypertext documents. It supports the creation of hyperdocuments by providing

'activity spaces' which can be seen as task-specific browsers. Users create a hyperdocument by interacting with four activity space browsers dedicated to the tasks of content generation and structuring, planning, arguing, and writing the final hyperdocuments under a rhetorical perspective (Content Space, Planning Space, Argumentation Space, and Rhetorical Space respectively). Information can be shared or private. SEPIA's basic hypertext objects are atomic nodes, composite nodes, and labeled links. Furthermore, annotation nodes and links are provided as specializations of atomic nodes and labeled links. SEPIA provides annotation nodes and thus supports collaboration via draft passing. Composite nodes are heavily used as organizational means. Each author may see the actions of the other users but is free to navigate through the document. SEPIA forces one to select objects before being able to execute an operation on them. That way a user can lock an object. A coloring mechanism is used to give a user information about locked objects. SEPIA is a general hypertext authoring tool that can be used for the production of a variety of documents such as manuals, scientific articles, project proposals, etc.

Chapter 3

Consistency in Cooperative Work

Consistency maintenance is a fundamental issue in many areas of computing systems, including operating systems, database systems, distributed shared memory systems, and groupware systems. Traditional algorithms typically maintain consistency by restricting concurrency; however, this approach is unsatisfactory in general, as it often interferes with the flexible management of group activity. In traditional database systems, the concept of *Transaction* and *Serializability* are taken to provide for the necessary consistency. The traditional transaction read/write model uses a semantics-free approach to transactions, and considers each database access as either a *read* or a *write*. Traditional model has four desirable properties for transactions, often called the ACID (Atomicity, Consistency, Isolation and Durability) properties [1].

1. Atomicity: All operations of a transaction must be treated as a single unit; all of the operations are executed, or none are.

2. Consistency: A transaction when executed alone takes the database from one consistent state to another.

3. Isolation: Each transaction executes as if it were the only transaction in the database. Its Intermediate results are not seen by other transactions.

4. Durability: The results of a transaction are never lost if the transaction terminates normally.

An execution is *serializable* if its effect (generally for both transactions and database) is equivalent to that of some serial execution of the same set of transactions. Both these concepts reflect the expectations of the users that their executions are not interfered by those of the others. In contrast, mutual interference is inherent and welcome in collaborative applications. Therefore, only (some form of) Consistency and Durability are expected to be satisfied among transactions collaborating with each other. However, Atomicity and Isolation are also required of them with respect to transactions not collaborating with them. Concurrency control is required to guard against inconsistencies, and to handle conflicting actions. However, concurrency control in groupware must be handled differently than traditional concurrency methods, simply because the user is an active part of the process. For example, people doing highly interactive activities will not tolerate delays introduced by conservative locking and serialization schemes.

The consequences of inconsistency are quite different for different domains. For example in a Document Authoring domain the writers may be willing to trade "high availability" for "accuracy." Many applications do not require absolute consistency

in user data. For instance, in a "shared whiteboard", absolute consistency is rarely a concern. Conflicts over a single-plane bitmap are minimal and non-intrusive. In these circumstances, the system would unlikely attempt to maintain data integrity rigorously. Indeed, the overhead of many consistency management strategies would interfere with the responsive performance and free-form interactive style that a shared whiteboard requires. In some applications, however, consistency can be vital.

Different types of consistencies such as Consistency Guarantees [7], Application defined consistency criteria, Operation Transformation scheme [43], History Merging [49], some relaxed forms of Serializability[8, 33], COO-Serializability (COO-SR) [27, 28], etc., suitable for cooperative applications have been explored.

3.1 Types of Consistencies

1. *Consistency Guarantees* mechanism uses knowledge of application semantics and the semantics of particular operations to increase concurrency and parallel activity. This mechanism is used in Prospero [7] where control over the consistency management mechanisms in the toolkit is given to the application. Prospero is a prototype toolkit for collaborative applications which uses metalevel techniques to allow application developers to reach in and tailor toolkit structures and behaviors to the particular needs of applications. Prospero exploits the semantics of specific applications and operations involved to

allow multiple users to act over data simultaneously. This involves finding operations which can be performed in parallel without leading to inconsistency. Prospero adopts an optimistic strategy by presuming that simultaneous actions will probably not result in conflict, but that if conflict does occur, things can be sorted out later. These simultaneous actions cause different users to have different views of the data; this is called *divergence.* To compensate for this effect, *synchronization* is done to re-establish a common view of the data. So, data management takes the form of continual divergence and synchronization of views of the data. The problem is that the divergence model makes no commitment to the nature or extent of the divergence. The longer two streams of activity remain active but unsynchronized, the greater their potential divergence, and so the more complex it becomes to resolve conflicts at synchronization time. Indeed there is no guarantee that the system will ever be able to resolve two arbitrary streams into a single, coherent view of the data store.

A generalized locking mechanism is used to achieve constrained divergence. Locking is used to guarantee the client of future consistency (promise that "no other user can make changes, so consistency is maintained") in exchange for a prediction of the clients future activity. This flexible interpretation allows applications to balance freedom of action against eventual consistency as appropriate to the particular circumstances of use. This is the first principle: *locks are guarantees of achievable consistency.* However, support for opportunistic work without completely abandoning the synchronization of parallel

activities is achieved by allowing clients to break their promises of future activity (and hence not holding the server to its guarantee of later consistency), and looking at syntactic consistency when necessary. This is the second principle: *a client can break a promise, in which case the server is no longer held to its guarantee.*

2. Consistency is obtained in [39, 49] by semantically correct exchange of information among cooperating users by means of merging histories of user activities. CoAct transactional model is designed for supporting cooperative work in multi-user environments. Individual user "transactions" are called *user activities* which are executed within the scope of a group-based "transaction" called the *cooperative activity* to enable cooperation between users involved in a joint effort. The CoAct model assigns a *user activity* and a *private workspace* to every user who takes part in a *cooperative activity.* By default, the *private workspaces* of the co-workers are isolated from each other. Each *cooperative activity* is associated with its own workspace, called the *common workspace* which is isolated from the *user activities.*

Execution constraints are assigned for cooperative activities that govern the execution of the work of the individual users as well as the overall cooperative activity. The constraints describe which operations are allowed to be executed within the context of the cooperative activity. Cooperation among user activities (belonging to the same cooperative activity) is achieved by the explicit, semantically correct exchange of information (operations) between

co-workers. They can exchange operations through the common workspace by means of *save* and *import*. Direct exchange of operations among user activities is achieved by means of *import* and *delegate*. All the users involved in the cooperative activity would integrate their individual work into the *common workspace* such that there is a single result of the cooperative activity.

Compatibility between user actions is exploited to extend the concept of user activities to *activity histories*. An *activity history* is referred to as histories of single user activities as well as the cooperative activity reflected in the common workspace. Correct subhistories (which are independent and consistent atomic units of work) are extracted from the activity histories and these subhistories can be exchanged between different activity histories by applying the merge algorithm described in [39], [49]. This process of merging of activity histories is guided by merging rules (described in the algorithm) which ensure that the resulting activity history is a correct one again in the sense that the observable behavior of actions in the merged history is the same as in the original histories. If merging of some operations leads to an inconsistent history, then those operations are compensated. The correctness criteria of the merge approach guarantees that no inconsistencies are introduced due to the exchange of information between concurrently executed work.

3. The model in [43] deals with consistency in Real-time Cooperative Editing systems. Cooperative editing system allows multiple users to view and edit a shared document simultaneously from different sites, which are connected

by a communication network. The model adopts a replicated architecture regarding the storage of shared documents to achieve good response and unconstrained collaboration. The shared documents are replicated at the local storage of each participating site (user), and so, the editing operations are first performed at local sites and then propagated to remote sites. Three inconsistency problems in such situations have been identified.

First, operations may arrive and be executed at different sites in different orders, resulting in *divergent final results* or *divergence*. Secondly, due to the nondeterministic communication latency, operations may be executed out of their natural *cause-effect* resulting in *causality violation*. Thirdly, due to concurrent generation of operations, the *actual effect* of an operation at the time of its execution may be different from the *intended effect* of this operation at the time of its generation. The three inconsistency problems are independent in the sense that the occurrence of one or two of them does not always result in the others. An execution at each participating site is consistent if it always maintains the following properties:

(a) Convergence: The same set of operations have been executed at all sites, and all copies of the shared document are identical.

(b) Causality-preservation: For any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed at all sites before O_b.

(c) Intention-preservation: For any operation O, the effects of executing O at all sites are the same as the intention of O, and the effect of executing

O does not change the effects of independent operations.

Various concurrency control protocols have been proposed for cooperative editing systems and have been successfully used in non-real time or real-time constrained cooperative editing systems. However, none of the existing approaches has addressed all the three inconsistency problems in cooperative editing systems under the constraints of a short response time, a short notification time, and support for unconstrained cooperative editing in such environments. Many researchers have dealt with the first two problems and have devised different protocols to ensure them but failed to correctly solve the third issue '*intention violation.*' A novel and integrated approach to correctly solve the intention violation (in combination with the problems of divergent final results and causality violation) is proposed in [43].

Convergence is achieved by an optimistic serialization strategy using undo/do/redo and causality-preservation by a state-vector-based timestamping scheme. Achieving intention-preservation is much harder because it is not related to the execution order of operations and cannot be resolved by just re-scheduling of operations as in the other two cases. Intention-preserving scheme is achieved by applying the *Generic Operation Transformation* (GOT) algorithm (given in [43]) which uses *inclusion* and *exclusion* transformation strategies.

4. The COO-SR model of [28] assumes a central repository containing all the objects developed during the cooperative execution. The execution consists

of users reading the versions of the objects in the repository, and writing new versions of objects and adding them to the repository. Users may execute the operations concurrently. The complete execution, entered in a log, is supposed to satisfy some 'external' and 'internal' consistency requirements. For external consistency, the transactions are partitioned into several groups based on interactions between them, an useful sub-log is extracted from the log by eliminating some operations which primarily aid cooperation but are irrelevant for serializability, and it is required that the sub-log be serializable with respect to the groups of transactions. For internal consistency, properties like "the last version of an object produced by a transaction in a group must be read by all the other transactions in that group" are demanded in the cooperative execution.

3.2 Examples

1. Consider a cooperative execution of document authoring. A group of authors A, B, C, D, and E are involved in writing a large document. The document contains three sections x, y, and z. They distribute the three sections among the five authors and outline some constraints, as to when and how the document should be finished. All the three sections are jointly written by several authors. All the authors are aware of the progress and changes on all sections. Some sections are edited by more than one of the authors concurrently. Several intermediate versions of the sections are written. Suppose authors A and B are in charge of section x, authors B, C and D are in charge of section

y and authors D, E and A are incharge of section z. $W_{Ai}(x)$ represents the execution of a *write* operation by processor A of the object x. Similarly a read operation $R_{Bi}(x)$ represents the execution of a *read* operation by processor B of the object x. In the figure the *read* operation $R_{B1}(x)$ reads from $W_{A1}(x)$ (the reads from relation shown by the solid arrow), that is the version written by W_{A1}. For example, consider the partial execution shown below.

Authors

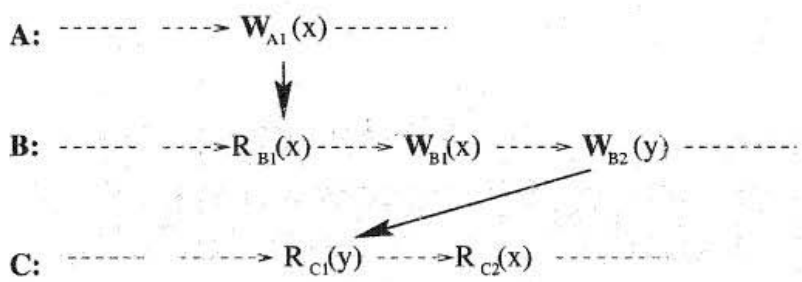

Figure 3.1: Example

In Figure 3.1, *process order* is shown by broken arrow and *reads from relation* by solid arrow. Author B reads section x written by author A ($R_{B1}(x)$ reads from $W_{A1}(x)$), and then writes section x, and after that, writes section y. Author C reads section y written by author B ($R_{C1}(y)$ reads from $W_{B2}(y)$), and then decides to read section x. At this stage there are two versions of section x ($W_{A1}(x)$ and $W_{B1}(x)$) available to author C. The problem is to decide which version is appropriate or suitable for author C. Depending on the requirements of the author and the application, author C might request for a latest write on a particular data item. So the system should support the user in deciding to select an appropriate version. For instance, on one extreme, the system can respond to author C by returning all the values on section x

from the database. And, on the other extreme, the user can specify a particular value to be returned. It may be assumed that, since author B writes section x after reading x written by author A, the version $W_{B1}(x)$ contains the modifications done in $W_{A1}(x)$. Hence, for author C, $W_{B1}(x)$ would be an appropriate choice if he wishes to read the 'recent' version on section x.

2. Consider another application of a weather data bank for a particular city. The system is serviced and used by different sources. The sources (or processors) are: a meteorological station at the city airport , a set of polar orbit satellites passing over the city, a geostationary satellite and a semi-permanent weather balloon operated by a local university. The sources collect physical data from their respective equipment and write into the data bank. These sources can also access the data reported (or written) by other sources. Each source reports the data at periodic intervals. These intervals vary for different sources. The actual data composition collected from various equipment might also vary depending upon the source. For example, the data from the polar satellite is significantly different from that of the weather balloon. However, all the processors work in a cooperative manner to update 'weather condition data'. Due to physical limitations not all sources might be able to collect the complete data at all times or may even collect partially incorrect data. The central data bank may be located at another location which serves several such cities.

Different weather related applications may be using the data for different re-

quirements. One such application may be about 'current weather conditions.' Each processor periodically writes data to the data bank. Before writing, it reads some recent data from the data bank (data may be from the same source or from different sources), checks it against its own data collected and writes into the data bank a filtered data. This reading is done in order to check if the current data collected is in line with the previous data (to preclude incorrect data from being written). It also helps in making sure that any missing data is filled in. For example, if the airport weather station is updating the 'current conditions', it may not be able to write information on road conditions (icy, snow covered, clear, etc). That information may be provided by other sources (perhaps satellite data). So the airport weather station requires this data and hence reads it from other sources. If the requirement of the application is such that, each set of physical data reported should be read at most once (by any of the cooperating processors) so that relevant information is passed on for further updates.

Using the system described above, we can look at a particular type of recentness. If a data item is read by a computer (or processor), it is assumed that it will process the data, filter it and write it back to the data bank. Sometimes, the data read may be partial or insufficient and so the processor might need to read data from other processors to complete its write. In that case, if a data is read first to be followed by another read or write operation, then the first data item that was read becomes obsolete and need not be read again. A processor keeps accessing different unread data only till its requirements

are satisfied.

Hence users (at different layers of the system) in a cooperative systems require different types of recentness of the values. In the next chapter we define five types of legalities, each capturing a different notion of 'recentness' of the values.

Chapter 4

Legalities

As discussed earlier, data given to a *read* request (by users) have to be consistent so that the writes (performed by the users) are also consistent. In this chapter, we describe different ways of satisfying a *read* request using generic legalities.

Depending on the requirements of the application and of the users involved (in the application), different levels of recentness of data may be required. In the next section, we give five legalities, each capturing a different notion of 'recentness' of the values [15]. We believe that the five legalities are very meaningful for cooperative environments where traditional concurrency methods [36] are too rigid to apply. This work is based on partial orders and illegality concepts. The illegality concepts are stated relative to certain defining relation (represented as ρ) on the operation executions. For example, for operation executions O and O', we defined $O \rightarrow_t O'$ if the execution of O is completed before that of O' starts in global *real time order*; the defining relation ρ ($\rightarrow_t$) here is *real time order*. In the thesis we have explored

the legalties for different defining relations. The defining relation ρ, stated also as $\rightarrow$, can be *real time order*, *causal order*, etc.

4.1 Definition

In this section we state the five illegalities based on a general defining relation ρ ($\rightarrow$), all in terms of the values the *reads* read. The five illegalities (or conversely, legalities) are *RR-illegal*, *RW-illegal*, *WR-illegal*, *WW-illegal*, and *NO-illegal*. For operations executions O and O', $O \rightarrow O'$ if the execution of O is completed before that of O' starts with respect to the defining relation.

The first four illegalities refer to the occurrences of the following situation where a legal serialization extending ρ cannot have the following situation for any *read* $R(x_1)$:

$$O(x_1) \rightarrow O'(x_2) \rightarrow R(x_1) \tag{4.1}$$

where, $x_1 \neq x_2$. x_1 and x_2 refer to the same data item x but represent different versions. Operations O and O' may each be a *read* or a *write*. Both O and O' may be from the same processor or from different processors. However, our emphasis is on the relation (ρ) between two operations. A *write* into an object defines a new value for the object; a *read* allows to obtain a value of the object. If O is a *write*, then $O(x_1)$ denotes a write operation $W(x)$ writing value x_1 in x. Similarly, if O is a *read*, then $O(x_1)$ denotes a read operation $R(x)$ returning data item x_1. In the later chapters we use $O(W_1)$ (where O can be a *read* or a *write* operation) instead of $O(x_1)$ or $R(x_1)$

The situation defined in eq. 4.1 above implies that $R(x_1)$ is *illegal* with respect to $\rightarrow$. This illegality notion can be expanded further as follows:

- *RR-illegal* if O is a *read* and O' is a *read*,
- *RW-illegal* if O is a *read* and O' is a *write*,
- *WR-illegal* if O is a *write* and O' is a *read*, and
- *WW-illegal* if O is a *write* and O' is a *write*.

In addition to the above four, a fifth illegality is defined as the *New-Old-inversion* or *New-Old-illegality* (*NO-illegality*):

- *NO-illegal:* for two different *writes* W_1 and W_2, writing values x_1 and x_2 in data itemxX respectively, and two different *reads* R and R', if $W_1(x_1) \rightarrow W_2(x_2)$ and $R'(x_2) \rightarrow R(x_1)$.

WW denotes that *WW-illegality* is allowed, and $\overline{WW}$ denotes that *WW-illegality* is not allowed. We use similar notation for other illegalities. An execution is said to be *ρ-causally consistent* if all the *read* operations in that execution are legal in all respects (that is the execution satisfies $\overline{WW}$, $\overline{WR}$, $\overline{RW}$, $\overline{RR}$, $\overline{NO}$). This is the strongest possible consistency. Weaker consistencies can be defined by allowing the *read* operations to satisfy only some, but not all, legalities. An execution belongs to a certain consistency class based on the presence or absence of each of these illegalities with respect to $\rightarrow$. This framework facilitates specification of a large variety of consistencies. The present work is based on the reading of intermediate versions in a cooperative application subject to the *read* operations satisfying different

legalities.' A common notion adopted in the literature for correctness of concurrent execution of operations on shared variables is causal consistency. Though not explicitly mentioned in [28], it can be shown that under certain assumptions, the internal consistency requirements mentioned in that paper include a form of causal consistency. This is explained in detail in the next section.

4.2 A form of Causal Consistency - COO-SR model

Causal Consistency: Let $H = (H, \rightarrow_H)$ be a history. H is *causally consistent* if all its read operations are legal w.r.t. causal order. Causal order is the transitive closure of the union of *process order* and *reads from relation* represented as $((\rightarrow_i \cup \rightarrow_{rf})^*)$. A read operation is said to be legal if it returns the most recently written value to the location being read.

Under ceratin assumptions and a defining realtion, the internal consistency requirement of the COO-SR model [28] (given in Chapter 3) corresponds to a form of causal consistency. This is shown below with the following assumptions.

1. ρ is the causal order represented as $(\rightarrow_i \cup \rightarrow_{rf})^*$.
2. A processor does not read its own write.
3. A processor does not read a version more than once.
4. The latest version of a processor and the versions read by the processor after writing that latest version contribute to the next version.
5. Every version that is created is useful for the final result.

- Since each version of a processor is (implicitly) useful for the next version of the same processor, each version need not be useful *i.e.,* need not be read by any body else.

6. However the final version of each processor (except possibly one) must be read.

7. The reads need to be consistent so that the writes the processors perform are consistent.

8. Eventually all we need is a partial order of the writes (versions), ending in one version.

9. Concurrent writes commute.

Properties:

1. No processor reads its own write.

   Case1: Consider *WR-illegality*, $W_1 \rightarrow R(W_2) \rightarrow R'(W_1)$

   W_1 and R' are of different processors. Considering the defining relation, if there exist a $W \rightarrow_{rf} R$ path in between, *i.e.,*

$$W_1 \ldots \rightarrow W_3 \rightarrow_{rf} R(W_3) \ldots \rightarrow R'(W_1) \tag{4.2}$$

   We have *WW-illegality* and therefore $WR \Rightarrow WW$, *i.e.,* $\overline{WW} \Rightarrow \overline{WR}$. It has been shown in [15], when ρ contains $\rightarrow_{rf}$ in transitive closure, $(\overline{WW}, \overline{WR}) \Rightarrow$ ρ-causality. Therefore all we need is $\overline{WW}$.

Case2: Consider *WW-illegality*, $W_1 \rightarrow W_2 \rightarrow R(W_1)$

An intervening $W_3 \rightarrow_{rf} R(W_3)$ path gives rise to the situation

$$W_1 \ldots \rightarrow W_3 \rightarrow_{rf} R(W_3) \ldots \rightarrow R(W_1) \tag{4.3}$$

We have *WR-illegality* and therefore $WW \Rightarrow WR$, *i.e.*, $\overline{WR} \Rightarrow \overline{WW}$. Thus $\overline{WR}$ is also good enough. Therefore all we need is either $\overline{WW}$ or $\overline{WR}$, both are equivalent.

Assumptions 2 and 3 imply WW- and WR-legal which guarantees ρ-causality. Assumption 2 may be changed to 2a: a processor can only read its write immediately after the write or before reading any other write. Even then ρ-causality is satisfied.

In this thesis, we subscribe to the above idea for the internal consistency of cooperative executions: *consistency of an execution is determined by the legalities of its reads.* The five legalities are explored for two different defining relations and for two different environments. The five illegalities are dealt with two defining relations: i)*real time order* and ii)*causal order.* We give detailed algorithms for ensuring the various legalities. We believe that the five legalities are very meaningful for cooperative environments, and CSCW system support must include helping *read* operations (users) satisfy these legalities. In the thesis we try to relax the ACID properties so that executions that are not serializable are acceptable in certain CSCW systems. Some types of inconsistencies may be acceptable to people in certain groupware applications. Later on people might mediate and repair their actions and conflicts following some social protocols.

As seen in the examples of Chapter 3, the first example indicates that a data item satisfying *WW-legality* would be an appropriate choice by author C. The second example requirement relates to a data item satisfying *RR-legality* and *RW-leaglity.* It has been shown in [15], for some special values of ρ and/or special system executions, the absence of some illegalities imply the absence of some other illegalities. Raynal and Schiper [37] define an execution to be causally consistent if no *read* in that execution is *WW-illegal* and *WR-illegal* (in our terminology).

4.3 Motivation

Due to the popularization of the Internet, cooperative applications are expected to become common place on the WEB. Cooperative applications are the result of the cooperation between several users (humans or software), playing different roles,who build a relational system which is structured by a common objective, or project, and for the duration of this project. Users interact (cooperate) when they share objects not only at the start and at the end of the execution, as in traditional database applications, but during their execution. Users communicate through peer to peer communication channels or through cooperation spaces, especially common repositories.

Example : Consider a scenario in which the medical staff of a hospital wants additional diagnostics before concluding on a patient disease. For that, two other specialists are contacted for their advice or opinion. The two specialists are ge-

ographically distributed. Patient records and diagnostic images are sent to these specialists who must do studies and report together, producing one report, despite geographical separation. Each specialist comments on the report of the other and write their own report. This process is done in an iterative fashion. Before each specialist arrives at his/her own final report, some intermediate versions are written. The isolation between the two specialists is broken by explicitly allowing them to share some intermediate versions of their reports when executing their respective process. Our concern is not with the end result but how and which of the intermediate versions are accessed by each of the users involved. A partial execution of how the specialists cooperate in their execution is shown below:

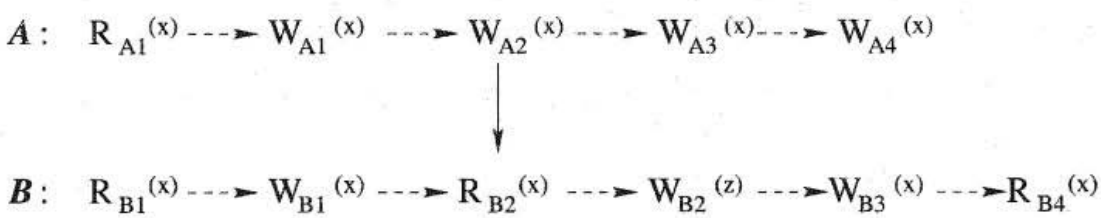

Figure 4.1: Example

The two specialists are represented as A and B and the data of the patient is represented as x. A and B read the patient data x (represented as $R_{A-}(x)$ and $R_{B-}(x)$ respectively) and write (represented as $W_{A-}(x)$ and $W_{B-}(x)$ respectively) their respective reports. In addition to this, each specialist reads the report of the other and writes comments which are denoted by data item z. They write intermediate versions before they arrive at the final report. For example, $W_{A1}(x)$, $W_{A2}(x)$ are the intermediate versions of the user A and $W_{B1}(x)$, $W_{B3}(x)$, and $W_{B4}(x)$ are the intermediate versions of the user B. $W_{B2}(z)$ represents comments written by

autor B.

As shown in the figure, user B reads a write of A and comments on that, *i.e.*, $R_{B2}(x)$ reads from $W_{A2}(x)$. For a further read by user B say, $R_{B3}(x)$, user B has more than one write to choose from (because user A has written $W_{A3}(x)$ and $W_{A4}(x)$ by the time user B issued the read $R_{B3}(x)$). In such situations giving all the write versions to user B may be waste of resources and not useful. Only key information should be provided to the user in order to reduce cognitive overload on the user and on the system. Suppose that user B is given all the versions and he/she chooses to read $W_{A1}(x)$. User B writes comments based on the write $W_{A1}(x)$ read. Thus the comments of B will be inconsistent because he/she has read an older version than the version read previously by B (*i.e.*, $W_{A1}(x)$ is an older version than $W_{A2}(x)$).

For example, suppose report $W_{A1}(x)$ contains the dosage say 15mg to be administered to the patient and later on user A decides to change it to 10mg which he modifies in the next version of the report $W_{A2}(x)$. Next, B writes a comment saying that 'Reduce the dosage by 5mg because of the age of the patient' after reading $W_{A2}(x)$. Again when B reads $W_{A1}(x)$ and writes the comments, he/she thinks that user A misunderstood and so he comments saying 'reduce the dosage by 10mg which confuses user A and inconsistency arises. In order to avoid such type of inconsistencies, the system should some how keep track of the versions read by the users and prompt or give the users only versions that are eligible, recent values or 'Legal'.

In the next chapter the five Illegalities are explained in detail with repect to *real time order.*

Chapter 5

Legalities in *Real Time Order*

We consider a cooperative system composed of a set of processors ($Proc_1$, $Proc_2$, $Proc_3$, etc.) which interact with each other by reading and modifying (writing) shared objects (x, y, z, etc.). The system supports two primitive operations: *read* and *write*. A *write* into an object defines a new value for the object. A *read* obtains a value of the object. Every (*read* or *write*) operation is assigned a unique *id* (*Rid* or *Wid*, respectively). The execution of a write operation by processor i of the object x is denoted by $W_{iu}(x)$: u^{th} *write* operation by processor i. Similarly a *read* operation is denoted by $R_{iu}(x)$. The system responds to the read request with references to a set of *writes* satisfying the desired "legality" criteria. The *read* in the model can read any value written by earlier or simultaneous writes.

The above representation and description holds good for all the three environments given in this thesis (centralized, distributed and mobile agent). In the following sections we explain in detail how the five legalities can be implemented

in a centralized environment considering the *Real Time Order*.

5.1 Definition

For operations O and O', we define $O \rightarrow O'$, that is O *precedes* O' if the execution of O is completed before that of O' starts in *global real time order*.

Scenarios for various illegalities for a *read* operation $R_{ij}(x)$ are shown in Figure 5.1. This figure is taken from [15].

Consider RR-illegality from Figure 5.1. *Writes* $W_{mn}(x)$ and $W_{pq}(x)$ are concurrent with each other. As shown in figure, *read* $R_{ij}(x)$ starts after the completion of the *reads* $R_{de}(x)$ and $R_{ab}(x)$, and *read* $R_{de}(x)$ starts after the completion of *reads* $R_{ab}(x)$. The three *reads* are concurrent with the *writes*. In order to avoid RR-illegality, *read* $R_{ij}(x)$ should not read *write* $W_{mn}(x)$.

Consider another case say, WW-illegality from Figure 5.1. *Write* $W_{pq}(x)$ starts after the completion of the *write* $W_{mn}(x)$ and *read* $R_{ij}(x)$ starts after the completion of the *write* $W_{pq}(x)$. Hence *write* $W_{pq}(x)$ is considered more recent *write* than $W_{mn}(x)$ and so if *read* $R_{ij}(x)$ reads $W_{mn}(x)$, then according to our terminology it is WW-illegal. Consider an example of a stock market news which is updated (displayed) by two news agencies. Another user is regularly reading or following the stock news. The user is interested in reading only the latest updated news and so such users should be given only WW-legal (according to our terminology) versions.

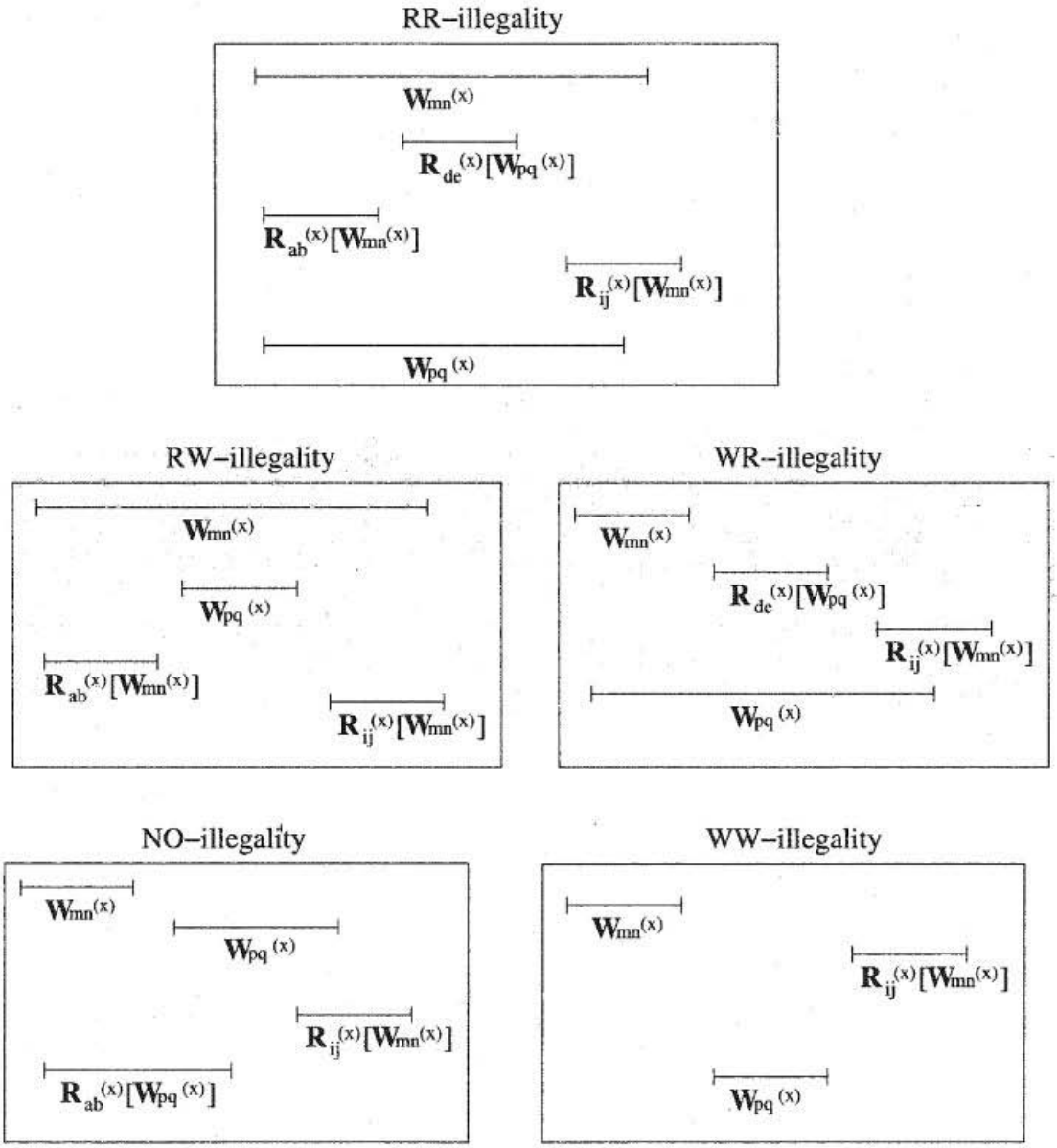

Figure 5.1: Example Showing Various Illegality Scenarios (in global real time order)

In the similar way, the other illegalities can be explained. In the next section we explain the data structures and give a detail algorithm (general case) satisfying all the five illegalities. The algorithm gives a step by step description of the actions to be taken when a *write* or a *read* operation is executed.

5.2 Data Structures

The algorithm requires the following data structures. All the sets given below are assumed to be maintained globally in a central repository without any replication and there exists a global clock. Individual processors do not have local copies of the data. The processors access the shared data from the central repository. Each *write* creates a version. As the operations are issued and completed, their respective start time and finish time are recorded in the appropriate sets.

In addition to the above information, *Wids* of *write* operations may be tagged as illegal with respect to one or more of the five legalities discussed. Based on the execution of an operation (*read/write*), appropriate sets are updated. We assume that no two operations start at the same time. The central repository contains the following data structures, one for each data item.

1. Ongoing Read Set: This set contains the *Rids* of all the ongoing *read* operations (a *read* operation that has started but not yet finished) reading that object. Tagged with each *read* operation is a list of *Wids* (of the *writes* on that object) that are concurrent with that *read* operation.

2. Ongoing Write Set: This set contains the *Wids* of all the ongoing *write* operations.

3. Finished Read Set: This set contains the *Rids* of the finished *read* operations. Tagged with each *read* operation is the *Wid* of the *write* read by that *read* operation.

4. Finished Write Set: This set contains the *Wids* of the finished *write* operations.

5.3 Algorithm

1. When a *write* operation $W_{ik}(x)$ is issued by processor i,

   (a) Enter *Wid* of $W_{ik}(x)$ in the Ongoing Write Set.

   (b) Tag the *Wid* of $W_{ik}(x)$ to each of the ongoing *read* operations in the Ongoing Read Set.

2. When a *write* operation $W_{ik}(x)$ is finished,

   (a) Remove *Wid* of $W_{ik}(x)$ from Ongoing Write Set and enter it in the Finished Write Set.

   (b) Check the Finished Read Set and collect all the *read* operations say, $R_{ab}(x)$ whose finish time is less than the start time of $W_{ik}(x)$. For each such *read* $R_{ab}(x)$, tag its respective *Wid* (*i.e.*, the *write* read by the *read* $R_{ab}(x)$) as RW-illegal in the Finished Write Set.

   (c) Check the Finished Write Set and tag all the *writes* whose finish time is less than the start time of $W_{ik}(x)$ as WW-illegal in the Finished Write Set.

(d) If any particular *Wid* in the Finished Write Set is tagged as illegal w.r.t all the five legalities, remove it from the Set.

(e) if the *Wid* removed from the Finished Write Set (in step 2d) also exists in the Finished Read Set (*i.e.,* this *Wid* is read by one or more *reads* in this set and accordingly this *Wid* will be tagged to those *reads*), then remove it from the Finished Read Set along with its respective *read.*

3. When a processor j issues a *read* $R_{ju}(x)$, an eligible list is formed from which $R_{ju}(x)$ can pick a *write* and read. In addition to the eligible list the *read* operation can choose from any of the *Wids* tagged to it. This tagged list of *Wids* are the *write* operations that are concurrent to $R_{ju}(x)$ and which have started after the *read* was issued. However *writes* which are concurrent to the *read* $R_{ju}(x)$ and which were started before the read was issued exists in the Ongoing Write Set. If the eligible list does not contain any value to be read, then a message is sent to the processor saying that there are no values that can be read satisfying the requested criteria.

   (a) RR-legality: The eligible list will contain all the *Wids* from the Finished Write Set excluding the *Wids* that are tagged as RR-illegal, plus all the *Wids* in the Ongoing Write Set.

   (b) RW-legality: The eligible list will contain all the *Wids* from the Finished Write Set excluding the *Wids* that are tagged as RW-illegal, plus all the *Wids* in the Ongoing Write Set.

   (c) WR-legality: The eligible list will contain all the *Wids* from the Finished Write Set excluding the *Wids* that are tagged as WR-illegal, plus all the

Wids in the Ongoing Write Set.

(d) NO-legality: The eligible list will contain all the *Wids* from the Finished Write Set excluding the *Wids* that are tagged as NO-illegal, plus all the *Wids* in the Ongoing Write Set.

(e) WW-legality: The eligible list will contain all the *Wids* from the Finished Write Set excluding the *Wids* that are tagged as WW-illegal, plus all the *Wids* in the Ongoing Write Set.

(f) Enter the *Rid* of $R_{ju}(x)$ in the Ongoing Read Set.

4. When a *read* operation $R_{ju}(x)$ is finished and it read $W_{ik}(x)$,

(a) Remove *Rid* of $R_{ju}(x)$ from Ongoing Read Set and enter *Rid* of $R_{ju}(x)$ along with the *Wid* of $W_{ik}(x)$ in the Finished Read Set.

(b) Check in the Finished Read Set for any *read* operation say, $R_{ab}(x)$ whose finish time is less than the start time of $R_{ju}(x)$. For each such *read* $R_{ab}(x)$ collected, tag its respective *Wid* (*i.e.*, the *write* read by the *read* $R_{ab}(x)$) as RR-illegal in the Finished Write Set.

(c) Check in the Finished Write Set and tag all the *writes* say, $W_{mn}(x)$, whose finish time is less than the start time of $R_{ju}(x)$ as WR-illegal in the Finished Write Set.

(d) Check in the Finished Write Set and tag all the *writes* say, $W_{mn}(x)$, whose finish time is less than the start time of $W_{ik}(x)$ ($W_{ik}(x)$ is read by the *read* operation $R_{ju}(x)$) as NO-illegal in the Finished Write Set.

(e) If any particular *Wid* in the Finished Write Set is tagged as illegal w.r.t all the five legalities, remove from the Set.

i. if the *Wid* removed from the Finished Write Set also exists in the Finished Read Set (*i.e.,* this *Wid* is read by one or more *reads* in Finished Read Set and accordingly this *Wid* will be tagged to those *reads*), then remove it from the Finished Read Set along with its respective *read.*

It is clear from the data structures and the algorithm described that some of the data items may have duplication even after taking care of some garbage collection. However based on the application and the implementation technique adopted, the sets can be optimized. One way is by removing *Wids* which are illegal with respect to all the five legalities from the Finished Write Set and Finished Read Set (In the Finished Read Set, *Wid* along with the *Rid* is removed). Different types of implementation techniques can be employed to arrive at an optimized algorithm. Due to the simplicity of understanding we have dealt the algorithm in this manner. The algorithm can be tailored based on the requirements of an application.

In the next section we give the correctness proof of our algorithm. The proof for each legality is given separately.

5.4 Correctness Proof

Proof of RR-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t RR-legality (step 3a in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* (*i.e., writes* that are not tagged as RR-illegal) from the Finished Write Set plus all the *Wids* in the Ongoing Write

Set. We will need to show that every *write* that is RR-illegal for $R_{ij}(x)$ exists in Finished Write Set tagged as RR-illegal. For a *write* $W_{m,n}(x)$ to be RR-illegal for $R_{ij}(x)$, we should have

$$R_{ab}(x)[W_{mn}(x)] \rightarrow R_{de}(x)[W_{pq}(x)] \rightarrow R_{ij}(x) \tag{5.1}$$

According to step 4b in the algorithm, when the *read* operation $R_{de}(x)$ is finished (*i.e.*, $R_{de}(x)$ read *write* $W_{pq}(x)$), *Wid* of $W_{mn}(x)$ is tagged as RR-illegal in the Finished Write Set. Hence, a *write* which is RR-illegal for *read* exists in the Finished Write Set tagged as RR-illegal or removed in garbage collection.

Proof of RW-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t RW-legality (step 3b in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* (*i.e.*, *writes* that are not tagged as RW-illegal) from the Finished Write Set plus all the *Wids* in the Ongoing Write Set. We will need to show that every *write* that is RW-illegal for $R_{ij}(x)$ exists in Finished Write Set tagged as RW-illegal. For a *write* $W_{mn}(x)$ to be RW-illegal for $R_{ij}(x)$, we should have

$$R_{ab}(x)[W_{mn}(x)] \rightarrow W_{pq}(x) \rightarrow R_{ij}(x) \tag{5.2}$$

According to step 2b in the algorithm, when the *write* operation $W_{pq}(x)$ is finished, *Wid* of $W_{mn}(x)$ is tagged as RW-illegal in the Finished Write Set. Hence, a *write* which is RW-illegal for *read* exists in the Finished Write Set tagged as RW-

illegal or removed in garbage collection.

Proof of WR-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t WR-legality (step 3c in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* (*i.e., writes* that are not tagged as WR-illegal) from the Finished Write Set plus all the *Wids* in the Ongoing Write Set. We will need to show that every *write* that is WR-illegal for $R_{ij}(x)$ exists in Finished Write Set tagged as WR-illegal. For a *write* $W_{mn}(x)$ to be WR-illegal for $R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow R_{ab}(x)[W_{pq}(x)] \rightarrow R_{ij}(x) \tag{5.3}$$

According to step 4c in the algorithm, when the *read* operation $R_{ab}(x)$ is finished (*i.e.*, $R_{ab}(x)$ reads *write* $W_{pq}(x)$) *Wid* of $W_{mn}(x)$ is is tagged as WR-illegal Finished Write Set. Hence, a *write* which is WR-illegal for *read* exists in Finished Write Set tagged as WR-illegal or removed in garbage collection.

Proof of NO-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t NO-legality (step 3d in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* (*i.e., writes* that are not tagged as NO-illegal) from the Finished Write Set plus all the *Wids* in the Ongoing Write Set. We will need to show that every *write* that is NO-illegal for $R_{ij}(x)$ exists in Finished Write Set tagged as NO-illegal. For a *write* $W_{mn}(x)$ to be NO-illegal for

$R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow W_{pq}(x) \tag{5.4}$$

$$R_{ab}(x)[W_{pq}(x)] \rightarrow R_{ij}(x) \tag{5.5}$$

According to step 4d in the algorithm, when the *read* operation $R_{ab}(x)$ is finished (*i.e.*, $R_{ab}(x)$ reads *write* $W_{pq}(x)$), *Wid* of $W_{mn}(x)$ is is tagged as NO-illegal in Finished Write Set. Hence, a *write* which is NO-illegal for *read* exists in Finished Write Set tagged as NO-illegal or removed in garbage collection.

Proof of WW-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t WW-legality (step 3e in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* (*i.e.*, *writes* that are not tagged as WW-illegal) from the Finished Write Set plus all the *Wids* from the Ongoing Write Set. We will need to show that every *write* that is WW-illegal for $R_{ij}(x)$ exists in Finished Write Set tagged as WW-illegal. For a *write* $W_{mn}(x)$ to be WW-illegal for $R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow W_{pq}(x) \rightarrow R_{ij}(x) \tag{5.6}$$

According to step 2c in the algorithm, when the *write* operation $W_{pq}(x)$ is finished, *Wid* of $W_{mn}(x)$ is tagged as WW-illegal in Finished Write Set. Hence, a *write* which is WW-illegal for *read* exist in Finished Write Set tagged as WW-illegal or removed in garbage collection.

Chapter 6

Legalities in *Causal Order*

Consider a Cooperative application where the number of processors and data objects involved are small in number and the processors are geographically distributed. Cooperation among processors (belonging to the same cooperative activity) is achieved by exchange or sharing of information (data) between co-workers. This exchange or sharing of information is not only at the start and at the end of their execution but during their execution also. Each processors involved in the cooperative activity is aware of their co-workers. They communicate through peer to peer communication channels or through cooperation spaces (common repositories, etc.) and the communication system is reliable. There exists a local clock with each processor. The granularity of the data objects is assumed to be small and simple. As the processors are geographically distributed, it is efficient to keep the data distributed. We give in detail how the legalities (with respect to causal order) can be implemented in a system with the above charecteristics.

In this chapter, the Legalities are defined with respect to *causal order*, as defined below. Section 4.2 gives the data structures required for the algorithm given in section 4.3 Section 4.4 gives the correctness proof of the algorithm and discussion is given in section 4.5.

6.1 Definition

For operations O and O', we define (i) $O \rightarrow_i O'$ (*process order*) if both are executed by the same processor, and in the order O and then O', and (ii) $O \rightarrow_{rf} O'$ (*reads from relation*) if O is a *write*, say W, and O' is a *read* R reading the value written by O, denoted as $R(W)$. We define $\rightarrow$ as $(\rightarrow_i \cup \rightarrow_{rf})^*$. where the asterisk $(*)$ indicates transitive closure. We say that O *precedes* O' if $O \rightarrow O'$.

Broken arrow and solid arrow in Figure 6.1 represents *process order relation* and *reads from relation* respectively. Examples of illegalities w.r.t causal order for different *reads* R_{mn} are shown below, referring to Figure 6.1:

(i) RR-illegality: Consider $R_{21}(x)[W_{11}(x)] \rightarrow R_{52}(x)[W_{21}(x)] \rightarrow R_{mn}(x)[W_{11}(x)]$. *Read* $R_{mn}(x)$ should not read the *write* $W_{11}(x)$ because it is RR-illegal for $R_{mn}(x)$ due to the relation $R_{21}(x)[W_{11}(x)] \rightarrow_i W_{21}(x) \rightarrow_{rf} R_{52}(x)[W_{21}(x)]$.

(ii) RW-illegality: Consider $R_{12}(z)[W_{42}(z)] \rightarrow W_{12}(z) \rightarrow R_{mn}(z)[W_{42}(z)]$. *Read* $R_{mn}(z)$ should not read the *write* $W_{42}(z)$ because it is RW-illegal for $R_{mn}(z)$ due to the relation $W_{42}(z) \rightarrow_{rf} R_{12}(z)[W_{42}(z)] \rightarrow_i W_{12}(z)$.

(iii) WR-illegality: Consider $W_{11}(x) \rightarrow R_{11}(x)[W_{51}(x)] \rightarrow R_{mn}(x)[W_{11}(x)]$.
Read $R_{mn}(x)$ should not read the *write* $W_{11}(x)$ because it is WR-illegal for $R_{mn}(x)$ due to the relation $W_{11}(x) \rightarrow_i R_{11}(x)[W_{51}(x)]$.

(iv) WW-illegality: Consider $W_{11}(x) \rightarrow W_{21}(x) \rightarrow R_{mn}(x)[W_{11}(x)]$.
Read $R_{mn}(x)$ should not read the *write* $W_{11}(x)$ because it is WW-illegal for $R_{mn}(x)$ due to the relation $W_{11}(x) \rightarrow_{rf} R_{21}(x)[W_{11}(x)] \rightarrow_i W_{21}(x)$.

(v) NO-illegality: Consider $W_{31}(z) \rightarrow_i W_{32}(z) \rightarrow_{rf} R_{41}(z)[W_{32}(z)]$ from figure 6.1. For a *read* $R_{mn}(z)$ where $R_{41}(z) \rightarrow R_{mn}(z)$ ($R_{41}(z)$ precedes $R_{mn}(z)$), reading $W_{31}(z)$ gives rise to a new-old inversion.

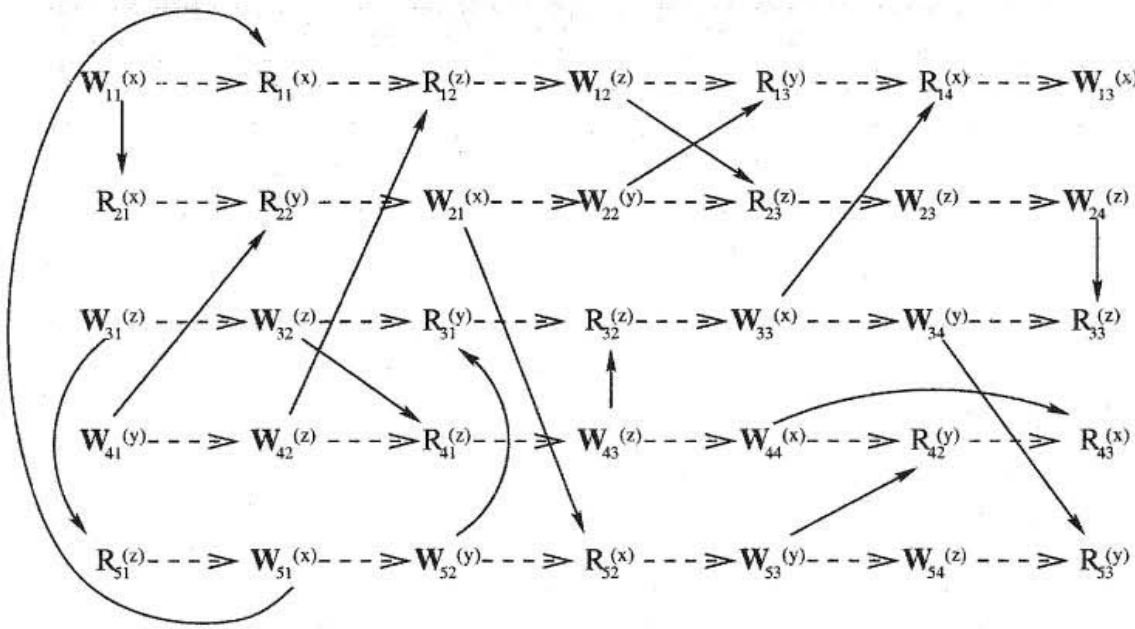

Figure 6.1: Example Showing Partial Interaction Between Different Processors

An execution is *causally consistent* if all the *reads* in that execution satisfy all the five legalities. We note that, for certain defining relations, the absence of

some illegalities may imply the absence of some other illegalities. Several interesting interdependencies of illegalities for different combinations of ρ are given in [15]. One such interesting result igiven in [15] is that, for $\rightarrow$ as defined above $(\rightarrow_i \cup \rightarrow_{rf})^*$, if all reads are WW- and WR-legal, then the execution is causally consistent. Another interesting case is, if $\rightarrow$ contains $\rightarrow_{rf}$, then

1. RR-illegality implies WR-illegality, and therefore $\overline{WR} \Rightarrow \overline{RR}$.
2. RW-illegality implies WW-illegality, and therefore $\overline{WW} \Rightarrow \overline{RW}$.
3. NO-illegality implies WW- and WR-illegality, and therefore $\overline{WR} \vee \overline{WW} \Rightarrow \overline{NO}$.

Proof: Consider the case of RR-illegality, that is

$$R_{ab}(x)[W_{mn}(x)] \rightarrow R_{cd}(x)[W_{pq}(x)] \rightarrow R_{ef}(x)[W_{mn}(x)] \tag{6.1}$$

If $\rightarrow$ contains $\rightarrow_{rf}$, then adding $W_{mn}(x) \rightarrow_{rf} R_{ab}(x)[W_{mn}(x)]$ gives

$$W_{mn}(x) \rightarrow_{rf} R_{ab}(x)[W_{mn}(x)] \rightarrow R_{cd}(x)[W_{pq}(x)] \rightarrow R_{ef}(x)[W_{mn}(x)] \tag{6.2}$$

which is WR-illegal, that is

$$W_{mn}(x) \rightarrow R_{cd}(x)[W_{pq}(x)] \rightarrow R_{ef}(x)[W_{mn}(x)] \tag{6.3}$$

Hence RR-illegality implies WR-illegality. Others also can be proved in a similar way. Several such implications have been given in [15].

6.2 Data Structures

The algorithm requires the following data structures. All the sets described below are maintained locally by each processor. The contents of the Illegality Set and the History Set of individual processors differ. Each *write* creates a version. Whenever a *write* operation is started, it is notified to all the processors involved in the execution. As the *write* operation on a particular object is completed, it is notified to all the processors involved in the execution. *Writes* executed by a particular processor are kept locally except that the *id* of that *write* is notified to other processors. So the actual data resides with the owner of the *write*.

The *read* request is assumed to be of two parts. The first part of the *read* is for getting a list of *writes* (*Wids*) satisfying a particular legality and the second *read* is to actually read a *write* that has been selected from the list of *Wids* given. For example, consider a *read* $R_{ij}(x)$ by processor i which has picked a *write* $W_{mn}(x)$ from the eligible list. The system supports or responds to this *read* request by collecting the data (*write*) from processor m. Along with the data, the History Tree and the Illegal Write List attached to that *write* ($W_{mn}(x)$) is also given. The History Tree (HT) and the Illegal Write List (IWL) are explained later. This information is provided to processor i because all the illegal *writes* (w.r.t all the five legalities up to the *write* $W_{mn}(x)$) for processor m also holds good for processor i due to the transitivity nature of the causal order $(\rho = \rightarrow_i \cup \rightarrow_{rf})^*$. Based on the History Tree and the Illegal Write List provided by processor m, appropriate sets of processor i are updated as explained in the algorithm below.

1. Ongoing Write Set: This set contains the *Wids* of all the ongoing *write* operations (a *write* operation that has started but not yet finished). A Ongoing Write Set is maintained by each of the processors involved in the execution. The structure of the Ongoing Write Set is similar to that of the Master Set. For example, when a *write* $W_{mn}(x)$ (started) is notified to processor i by processor m, then processor i enters the *Wid* of $W_{mn}(x)$ in the *m-th* row of its (processor i) Ongoing Write Set.

2. Master Set: This set contains the *Wids* of all the writes executed by all the processors or in other words it can be assumed to be the database of all the writes. A Master Set is maintained by each of the processors involved in the execution. Figure 6.2 shows the structure of Master Set. For example, when a *write* $W_{mn}(x)$ (finished) is notified to processor i by processor m, then processor i moves the *Wid* of $W_{mn}(x)$ from its (processor i) Ongoing Write Set to the *m-th* row of its Master Set.

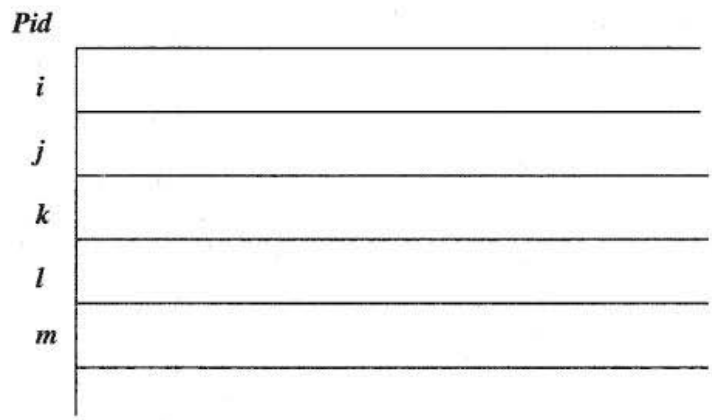

Figure 6.2: Master Set

3. History Set (HS): A History Set is maintained by each of the processors

involved in the execution. The History Set of a particular processor k contains the *ids* of all the *read* and *write* operations (*Rid* and *Wid*, respectively) issued by the processor k on all the objects. The structure of the History Set is shown in Figure 6.3. The content of the History Set of each processor differs. For example, if a *write* or a *read* operation is executed by processor i, then the *id* of the operation is entered in the i^{th} row of the History Set of processor i. Each slot contains two parts. If part 1 of a slot contains a *write* operation, part 2 of that slot is empty. If part 1 contains a *read* operation, then part 2 contains the *id* of the *write* read by the *read* operation. HS[i] represents the History Set of processor i. In addition to the above information, every *write* (part 1 or part 2 of the History Set) operation in the History Set of any processor has a History Tree and a Illegal Write List.

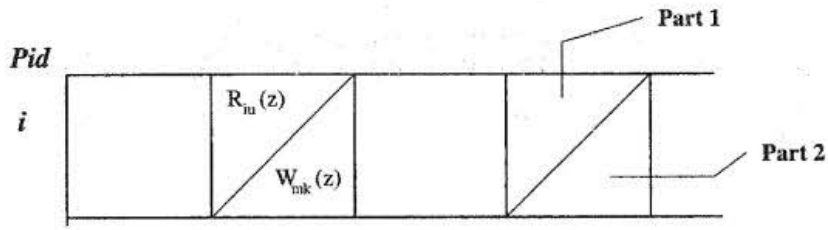

Figure 6.3: History Set

4. History Tree and Illegal Write List: For every *write* (*i.e.*, either in part 1 or part 2) in the History Set of any processor, some information has to be stored. This information is stored in History Tree and Illegal Write List. Each such History Tree exists for each of the *writes* mentioned above. The History Tree of a particular *write* is nothing but all the causally preceding operations of that *write*. History Tree is represented as HT. HT[$W_{ik}(x)$] represents History Tree of the *write* $W_{ik}(x)$. The Illegal Write List of a particular *write* say $W_{ik}(x)$

contains all the *Wids* present in the Illegality Set of processor i (except those *ids* w.r.t object x) when the *write* $W_{ik}(x)$ is executed by processor i. Illegal Write List is represented as IWL. IWL$[W_{ik}(x)]$ represents the Illegal Write List of the *write* $W_{ik}(x)$. For example, consider the History Set of processor 1 from Figure 6.1 after the *write* $W_{12}(z)$ is executed. The History Set looks like as shown in Figure 6.4.

As shown in Figure 6.4, all the causally preceding operations of a particular *write* form the History Tree of that *write*.

5. Illegality Set: Each such set is maintained locally by each of the processors involved in the execution. This set contains the writes which are tagged as illegal w.r.t any of the legalities discussed. Any *write* in the set may be tagged as illegal w.r.t to more than one legality. Figure 6.5 shows the structure of Illegality Set. X^{th} row of the Illegality Set of processor m contains all the illegal *writes* w.r.t object x.

In the next section we give a step by step description of the actions to be taken when a *write* or *read* operation is executed. The algorithm gives in detail how a processor updates its sets when a particular operation is executed.

6.3 Algorithm

1. When a *write* operation $W_{ik}(x)$ (*i.e.*, *write* operation on object x) is issued by processor i, it is notified to all the processors. All the processors enter this *write* into their respective Ongoing Write Set as soon as they are notified of

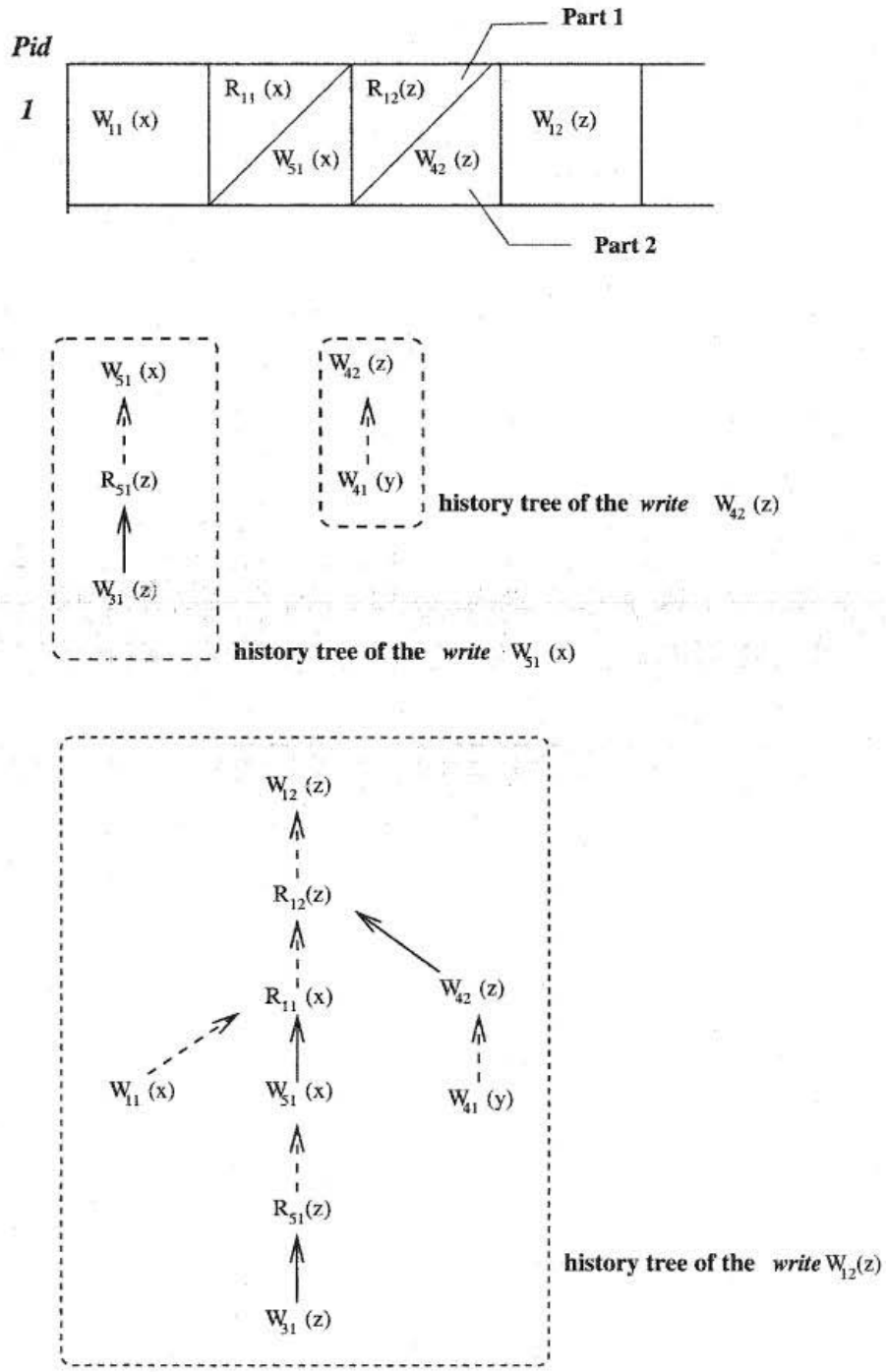

Figure 6.4: Example History Trees

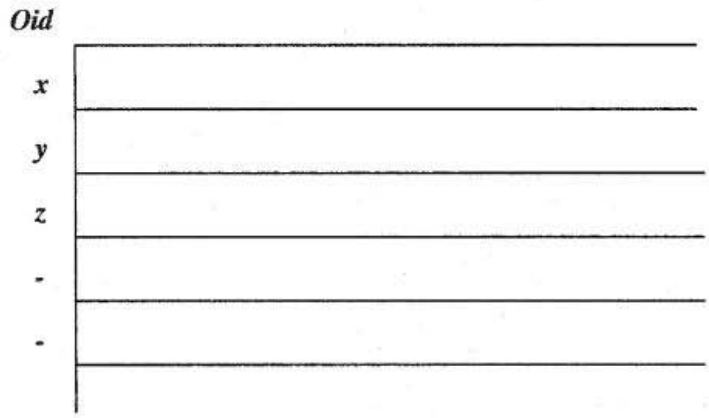

Figure 6.5: Illegality Set

this *write*. Processor i enters the *Wid* of $W_{ik}(x)$ into its Ongoing Write Set.

2. When a *write* operation $W_{ik}(x)$ (*i.e.*, *write* operation on object x issued by processor i) is finished, it is notified to all the processors.

   (a) All the processors move the *Wid* of this *write* from their respective Ongoing Write Set to their Master Sets as soon as they are notified of this *write*.

   (b) Processor i enters the *Wid* of $W_{ik}(x)$ into its Master Set and History Set and removes this *Wid* from its Ongoing Write Set. After entering into the History Set, processor i forms the History Tree (HT) and the Illegal Write List (IWL) of the *write* $W_{ik}(x)$.

      i. The History Tree is nothing but all the *ids* present in the History Set along with any other History Trees of the *writes*. Looking back at Figure 6.4, the HT$[W_{12}(z)]$ contains all the contents of the History Set (in the same order) along with the History Trees of the *writes* $W_{51}(x)$ and $W_{42}(z)$. Once the History Tree of $W_{ik}(x)$ is formed and

stored, all the other History Trees of the previous *writes* can be removed.

ii. Now the Illegal Write List of the *write* $W_{ik}(x)$ must be formed. The Illegal Write List of the *write* $W_{ik}(x)$ (ILW$[W_{ik}(x)]$) contains all the *Wids* (except those *Wids* w.r.t object x) in the Illegality Set of processor i. Both the History Tree and Illegal Write List of the *write* $W_{ik}(x)$ is formed because, this information should be provided to any processor who reads this *write*.

(c) Check the History Tree of the *write* $W_{ik}(x)$ and

i. collect all the *read* operations on x, say, $R_{pq}(x)$ that transitively precede $W_{ik}(x)$ and tag the *read* operations as RW-illegal in the Illegality Set of processor i.

ii. collect all the *write* operations, say, $W_{mn}(x)$ that transitively precede $W_{ik}(x)$ (including $W_{ik-1}(x)$, if it exists) and tag them as WW-illegal in the Illegality Set of processor i.

(The above two checkings are done for all the operations that lie between the previous *write* on x by processor i (if it exists) and $W_{ik}(x)$.)

3. If a processor j request to read a *write* $W_{ik}(x)$, go to step 5, else

4. When a processor j issues a *read* request $R_{ju}(x)$, a eligible list is formed from which $R_{ju}(x)$ picks a *write* and reads. As all the information is available locally with each processor (*i.e.*, the Master Set, Ongoing Write Set and the Illegality Set), the eligible list can be formed locally by a processor. If the

eligible list does not contain any value to be read, then a message is sent to the processor saying that there are no values to be read satisfying the requested criterion. Requesting for versions satisfying more than one legality can be satisfied by providing different eligible lists to the *read* operation (one for each legality criterion),

(a) RR-legality: The eligible list will contain all the *write* operations on object x in the Master Set and Ongoing Write Set of processor j minus the *writes* on object x that are tagged as RR-illegal in the Illegality Set of processor j.

(b) RW-legality: The eligible list will contain all the *write* operations on object x in the Master Set and Ongoing Write Set of processor j minus the *writes* on object x that are tagged as RW-illegal in the Illegality Set of processor j.

(c) WR-legality: The eligible list will contain all the *write* operations on object x in the Master Set and Ongoing Write Set of processor j minus the *writes* on object x that are tagged as WR-illegal in the Illegality Set of processor j. For example, consider the *read* $R_{13}(y)$ (in the Figure 6.1) on object y requesting WR-legal values. The eligible list (which is formed locally by processor 1) given to the *read* will contain $W_{22}(y), W_{41}(y), W_{52}(y)$ and $W_{53}(y)$, assuming operation $W_{34}(y)$ has not been executed yet.

(d) NO-legality: The eligible list will contain all the *writes* operations on object x in the Master Set and Ongoing Write Set of processor j minus

the *writes* on object x that are tagged as NO-illegal in the Illegality Set of processor j.

(e) WW-legality: The eligible list will contain all the *writes* operations on object x in the Master Set and Ongoing Write Set of processor j minus the *writes* on object x that are tagged as WW-illegal in the Illegality Set of processor j. For example, consider the *read* $R_{14}(x)$ (in the Figure 6.1) requesting WW-legal values. Accordingly the list given to the *read* will contain $W_{21}(x), W_{33}(x)$ and $W_{44}(x)$.

Note: The members of the eligible list continue to be eligible until the *read* operation is over. However one or more *writes* may be added to the Ongoing Write Set during the interval of the *read* getting the eligible list and selecting a *write* to read. This can be taken care of depending on the requirements of the application and the users involved. As mentioned in the algorithm, several sets (*e.g.,* Master Set, Ongoing Write Set and Illegality Set) need to be accessed to get the eligible list. The order in which these sets are accessed is first the Illegality Set, then the Ongoing Write Set and then the Master Set.

5. When $R_{ju}(x)$ picks a *write* $W_{ik}(x)$ from the eligible list given,

   (a) Get the *write* ($W_{ik}(x)$) from processor i and give it to processor j along with the History Tree and Illegal write List.

   (b) Enter the *Rid* of $R_{ju}(x)$ and *Wid* of $W_{ik}(x)$ in $HS[j]$. Store the History Tree of $W_{ik}(x)$ in the HT[$W_{ik}(x)$].

(c) Scan the *process order tree* (which can be constructed from the History Set of processor j) of $R_{ju}(x)$ and collect all the *read* and *write* operations on object x that precede $R_{ju}(x)$. This scanning of the *process order tree* is done for all the operations between the previous *read* operation on x (if it exists) by processor j and $R_{ju}(x)$.

i. *Affects RR-legality:*

For each *read* operation collected, tag its respective *Wid* as RR-illegal in the Illegality Set of processor j,

ii. *Affects WR-legality:*

For each *write* operation collected, tag its *Wid* as WR-illegal in the Illegality Set of processor j.

(d) Scan the *reads from relation tree* of $R_{ju}(x)$ (nothing but the HT$[W_{ik}(x)]$) and collect all the *read* and *write* operations on object x that precede $W_{ik}(x)$. The *reads from relation tree* is nothing but the history tree provided by the processor i along with *write* $W_{ik}(x)$.

i. *Affects RR- and RW-legality:*

For each *read* operation collected, tag its respective *Wid* as RR- and RW-illegal in the Illegality Set of processor j,

ii. *Affects WR-, NO-, WW-legality :*

For each *write* operation collected, tag its *Wid* as WR-, NO-, and WW-illegal in the Illegality Set of processor j,

6. Check the Illegal Write List of the *write* $W_{ik}(x)$ and update accordingly in the Illegality Set of processor j. Any *write* in the tagged list tagged as illegal

w.r.t a particular legality should also be tagged as illegal in the Illegality Set of processor j.

7. Empty the Illegal Write List of the *write* $W_{ik}(x)$.

Note: Execution of a *read* operation on a certain object x may cause some other versions on any object to become illegal w.r.t any of the legalities discussed, to the processor that executed the *read* operation. This is due to the *reads from relation* where *read* $R_{ju}(x)$ reading from the *write* $W_{ik}(x)$.

Tree Scanning: In the algorithm discussed above, we use scanning of a tree in order to identify some illegal *writes* and update the appropriate sets. The main idea behind this scanning is to identify some of the transitively preceding operations. For example consider the Figure 6.1 and the *read* operation $R_{14}(x)$. When the *read* operation $R_{14}(x)$ is executed (*i.e.*, $R_{14}(x)$ has read the *write* $W_{33}(x)$), then the tree of $R_{14}(x)$ has to be scanned for any preceding operations that might affect the legalities for processor *1*. The read operation $R_{14}(x)$ has two branches in the tree, one is the *process order tree* and the other is the *reads from relation tree* or the history tree provided by processor *3* to processor *1* along with the write $W_{33}(x)$. The tree (including both *process order tree* and *reads from relation tree*) of an operation of a particular processor i can be extracted from the History Set of processor i. Assume that the tree is scanned from left to right. The Figure 6.6 shows the tree of the *read* operation $R_{14}(x)$. For instance the tree of $R_{14}(x)$ is scanned to collect all the WW-illegal *writes* w.r.t object x. So once the *write* operation $W_{11}(x)$ is reached there is no need for scanning the tree further beyond the *write* $W_{11}(x)$ because

$W_{11}(x)$ is already tagged as WW-illegal for processor *1*.

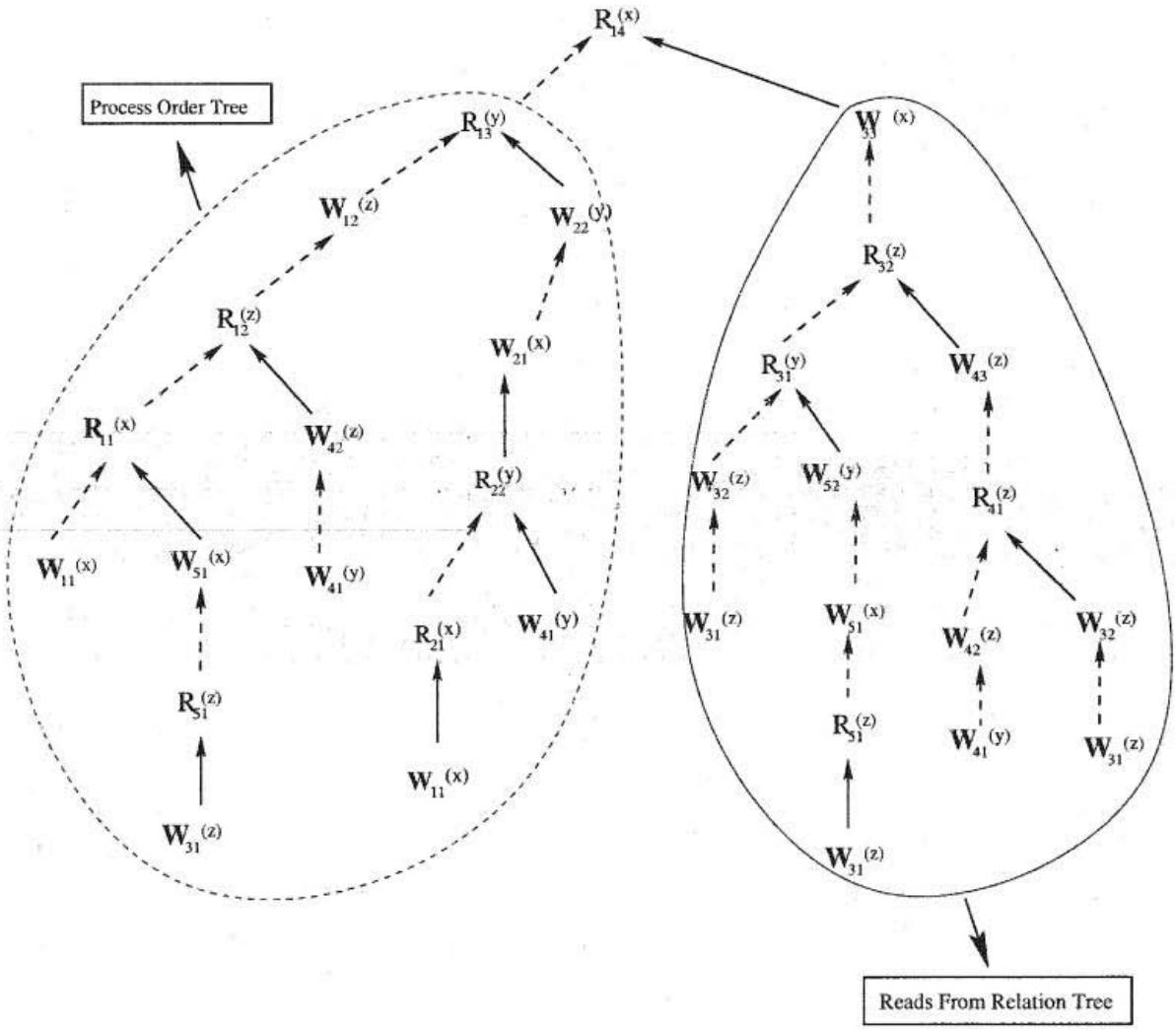

Figure 6.6: Scanning a Tree

In the next section we give the correctness proof of our algorithm. The proof for each legality is given separately.

6.4 Correctness Proof

Proof of RR-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t RR-legality (step 4a in the algorithm) given to $R_{ij}(x)$ consists of all the *writes* on object x in the Master Set and Ongoing Write Set minus the *writes* that are tagged as RR-illegal in the Illegality Set of processor i. We will need to show that every *write* that is RR-illegal for $R_{ij}(x)$ is tagged as RR-illegal in the Illegality Set of processor i. For a *write* $W_{mn}(x)$ to be RR-illegal for $R_{ij}(x)$, we should have

$$R_{ab}(x)[W_{mn}(x)] \rightarrow R_{de}(x)[W_{pq}(x)] \rightarrow R_{ij}(x) \tag{6.4}$$

Three cases arise:

<u>case 1</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow_i W_{ab}(z) \rightarrow R_{ih}(x)[W_{pq}(x)]$

Then, according to step 5c(i), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RR-illegal in the Illegality Set of processor i.

<u>case 2</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow_i W_{ao}(x) \rightarrow_{rf} R_{ih}(x)[W_{ao}(x)]$

Then, according to step 5d(i), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RR-illegal in the Illegality Set of processor i.

<u>case 3</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow R_{de}(x)[W_{pq}(x)] \rightarrow W_{ko}(z) \rightarrow_{rf} R_{it}(z)[W_{ko}(z)] \rightarrow_i R_{ij}(x)$

Then, according to step 6, when the *read* operation $R_{it}(z)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RR-illegal in the Illegality Set of processor i.

Proof of RW-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t RW-legality (step 4b in the algorithm) given to $R_{ij}(x)$ consists of all the *writes* on object x in the Master Set and Ongoing Write Set minus the *writes* that are tagged as RW-illegal in the Illegality Set of processor i. We will need to show that every *write* that is RW-illegal for $R_{ij}(x)$ is tagged as RW-illegal in the Illegality Set of processor i. For a *write* $W_{mn}(x)$ to be RW-illegal for $R_{ij}(x)$, we should have

$$R_{ab}(x)[W_{mn}(x)] \rightarrow W_{pq}(x) \rightarrow R_{ij}(x) \tag{6.5}$$

Three cases arise:

<u>case 1</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow W_{ik}(x) \rightarrow_i R_{ij}(x)$

Then, according to step 2a, when the *write* operation $W_{ik}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RW-illegal in the Illegality Set of processor i.

<u>case 2</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow W_{pq}(x) \rightarrow_{rf} R_{ip}(x)[W_{pq}(x)] \rightarrow R_{ij}(x)$

Then, according to step 5d(i), when the *read* operation $R_{ip}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RW-illegal in the Illegality Set of processor i.

<u>case 3</u>: $R_{ab}(x)[W_{mn}(x)] \rightarrow W_{lq}(x) \rightarrow W_{pk}(z) \rightarrow_{rf} R_{ip}(z)[W_{pk}(z)]$

Then, according to step 6, when the *read* operation $R_{ip}(z)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as RW-illegal in the Illegality Set of processor i.

Proof of WR-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t WR-legality (step 4c in the algorithm) given to $R_{ij}(x)$ consists of all the *Wids* on object x in the Master

Set and Ongoing Write Set minus the *Wids* that are tagged as WR-illegal in the Illegality Set of processor i. We will need to show that every *write* that is WR-illegal for $R_{ij}(x)$ is tagged as WR-illegal in the Illegality Set of processor i. For a *write* $W_{mn}(x)$ to be WR-illegal for $R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow R_{ab}(x)[W_{pq}(x)] \rightarrow R_{ij}(x) \tag{6.6}$$

Three cases arise:

<u>case 1</u>: $W_{mn}(x) \rightarrow R_{ih}(x)[W_{pq}(x)] \rightarrow R_{ij}(x)$

Then, according to step 5c(ii), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WR-illegal in the Illegality Set of processor i.

<u>case 2</u>: $W_{mn}(x) \rightarrow W_{pq}(x) \rightarrow_{rf} R_{ip}(x)[W_{pq}(x)] \rightarrow R_{ij}(x)$

Then, according to step 5d(ii), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WR-illegal in the Illegality Set of processor i.

<u>case 3</u>: $W_{mn}(x) \rightarrow R_{ab}(x)[W_{pq}(x)] \rightarrow W_{lk}(z) \rightarrow_{rf} R_{ip}(z)[W_{lk}(z) \rightarrow R_{ij}(x)$

Then, according to steps 6, when the *read* operation $R_{ip}(z)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WR-illegal in the Illegality Set of processor i.

Proof for NO-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t NO-legality (step 4d in the algorithm) given to $R_{ij}(x)$ consists of all the *writes* on object x in the Master Set and Ongoing Write Set minus the *writes* that are tagged as NO-illegal in the Illegality Set of processor i. We will need to show that every *write* that is NO-illegal for $R_{ij}(x)$ is tagged as NO-illegal in the Illegality Set of processor i. For a *write*

$W_{mn}(x)$ to be NO-illegal for $R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow W_{pq}(x) \rightarrow_{rf} R_{st}(x) \rightarrow R_{ij}(x) \tag{6.7}$$

Two cases arise:

case 1: $W_{mn}(x) \rightarrow_i W_{mo}(x) \rightarrow_{rf} R_{ih}(x)[W_{mo}(x)] \rightarrow R_{ij}(x)$

Then, according to step 5d(ii), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as NO-illegal in the Illegality Set of processor i.

case 2: $W_{mn}(x) \rightarrow W_{cd}(x) \rightarrow_{rf} R_{ab}(x)[W_{cd}(x)] \rightarrow W_{pq}(z) \rightarrow_{rf} R_{ih}(z)[W_{pq}(z)] \rightarrow R_{ij}(x)$

Then, according to step 6, when the *read* operation $R_{ih}(z)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as NO-illegal in the Illegality Set of processor i.

Proof for WW-legality:

Consider a *read* $R_{ij}(x)$. Recall that the eligible list w.r.t WW-legality (step 5d in the algorithm) given to $R_{ij}(x)$ consists of all the *writes* in the Master Set minus the *writes* that are tagged as WW-illegal in the Illegality Set of processor i. We will need to show that every *write* that is WW-illegal for $R_{ij}(x)$ is tagged as WW-illegal in the Illegality Set of processor i. For a *write* $W_{mn}(x)$ to be WW-illegal for $R_{ij}(x)$, we should have

$$W_{mn}(x) \rightarrow W_{pq}(x) \rightarrow R_{ij}(x) \tag{6.8}$$

Three cases arise:

case 1: $W_{mn}(x) \rightarrow W_{ik}(x) \rightarrow R_{ij}(x)$

Then, according to step 2b, when the *write* operation $W_{ik}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WW-illegal in the Illegality Set of processor i.

case 2: $W_{tn}(x) \rightarrow W_{mo}(x) \rightarrow_{rf} R_{ih}(x)[W_{mo}(x)] \rightarrow R_{ij}(x)$

Then, according to step 5d(ii), when the *read* operation $R_{ih}(x)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WW-illegal in the Illegality Set of processor i.

case 3: $W_{mn}(x) \rightarrow W_{ab}(x) \rightarrow W_{pq}(z) \rightarrow_{rf} R_{ih}(z)[W_{pq}(z)] \rightarrow R_{ij}(x)$

Then, according to step 6, when the *read* operation $R_{ih}(z)$ is executed, *Wid* of $W_{mn}(x)$ is tagged as WW-illegal in the Illegality Set of processor i.

6.5 Discussion

The algorithm given above is a general one with a push based scheme. Different applications may impose different requirements on the system. A push based scheme is one where any changes in a particular data is propagated to all the users involved in that execution. However different methods can be easily achieved without much modification to the algorithm. Consider a pull based scheme where in the notification of any changes in a particular data is propagated only to the users who are interested or who have subscribed for it. This can be achieved in the algorithm by having a list of subscribers for each object and notifying accordingly. Another requirement may be of finding out if a certain *write* has been read by anybody. This can be achieved in the algorithm by making each processor to keep track of who is reading its writes. Recalling from the algorithm, a *read* request is satisfied by providing suitable versions. However, what if those suitable versions are not

available at that moment and need to be provided when available. The algorithm can accommodate such requirement in different ways. Hence different requirements can be achieved with little modifications to the algorithm.

Chapter 7

Legalities in Mobile Agent Environment

Recent years have seen an explosion in the amount of information available in electronic form, forcing the developers of information acquisition systems to re-evaluate their model of the world. Vast amounts of electronic information are freely available at a multitude of sites to anyone with access to the Internet. The problems of searching for information on the WWW are familiar to every web surfer: slow search response time, irrelevant links, broken links, etc,.

The location of information on the WWW today requires the use of one of the many "search engines" available. The continuing rapid expansion of the WWW leads to three related problems with current search technology: currency, bottlenecks and coverage. Increasing use of the web is likely to exacerbate these problems

and hence different solutions to solve these problems are being proposed. One solution proposed in [12] is to distribute the indexing process and the use of mobile, collaborative agents to search the information. How the indexing process is dealt with is not the subject of this thesis but we address the aspects of interaction (or dialogue) between agents which is at the core of their system. The mobile agents are used to search and collect the information. This can be thought of as browsing in a collaborative fashion or cooperative agent based solution for information gathering. Hence we introduce the legality concept to restrict the interaction between the agents. These restrictions are based on the user requirement and the application in the form of legalities.

The algorithm explained in the previous chapter is based on a system where all the participating processors are aware of each other and the communication medium is message broadcasting. Any change with a particular processor is notified to all the others immediately and the assumption made is that no messages are lost. In this chapter, the legalities (defined in chapter 4) are used to address a different environment and in an implementation different from the one given in chapter 4. Consider a system where the participating sites are not aware of each other due to the dynamic nature of the sites or due to the large number of the sites involved, etc. One such system can be thought of as WWW. In this chapter, we tune our algorithm to accommodate such systems by relaxing some of the previous assumptions. The legalities defined with respect to *causal order* (refer to chapter4, section4.3) are considered.

Mobile agent is a piece of mobile code and associated data which can move from one machine to another. Each agent has a unique identifier which can be used to locate and communicate with the agent at any time. These mobile agents (hopefully) meet other agents of similar interest and exchange information. Agents represent real people or organizations in cyberspace. They are able to autonomously act on behalf of their human counterparts and can negotiate with each other in order to achieve their goals.

Consider a system composed of a set of processors ($Proc_1$, $Proc_2$, $Proc_3$, etc.) which interact with each other by reading and modifying (writing) shared objects (x, y, z, etc.). The system supports two primitive operations: *read* and *write*. Two types of mobile agents are introduced into the system: a publicist and a ferret [12]. Every processor, its operations (*read* and *write*), and the mobile agents are assigned unique identifiers (*Pid*, *Rid*, *Wid*, or *Aid* respectively). Whenever a *read* operation is to be executed, a ferret is fired to collect the required information. In a similar way, when a *write* is executed, a publicist is fired to advertise the write. Along with the *Rid* and the *Wid* of the *read* and the *write* operations, the respective agent *id* is also tagged to these operations. Ferrets and publicists keep track of all the relevant agents they meet in a list called Meta Data List (MDL). Ferrets may be timed to return home after collecting the required information. Unlike ferrets, publicists have no need to go home at the end of their mission. They can be allowed to expire after certain time. The actual *writes* reside at the owner sites and any exchange of information between the agents is w.r.t *ids* of the *writes* and not the actual values of the *writes*. By relying simply on chance encounters and allowing

sufficient time, ferrets and publicists with similar interests will meet and be able to exchange information.

The system responds to the read request by firing a ferret to collect the required information. The ferret meets other agents (be a publicist or a ferret) and collects the required information. The ferret returns to the processor and presents to the *read* with a list of eligible *Wids* satisfying the requested legality criterion. The reader can choose any *Wid* from the list and update its sets based on the information collected and the *write* that was chosen by the *read*.

In the next section we give the data structures and a detailed algorithm satisfying the five legalities with respect to causal order in such an environment. The legalities are w.r.t the information gathered by a particular processor and not w.r.t the whole system.

7.1 Data Structures

This algorithm requires the following data structures:

1. Master Set (MS)
2. History Set (HS)
3. Illegality Set
4. History Tree (HT) and The Illegal Write List (IWL)

5. Meta Data List (MDL)

All the data structures given above are similar to the ones explained in chapter 4 (refer section 4.3.1) except for one additional data structure called the Meta Data List (MDL). All the sets given above are maintained locally by each site except for the MDL. Meta Data list is used by an agent to keep track of all the relevant agents (ignored agents are not entered into the MDL) it meets. Each fired agent maintains such list and contains the *ids* (along with *Wids*) of the agents it meets. The MDL of a publicist is destroyed when the publicist dies (or its time expires) whereas the MDL of a ferret is destroyed after the ferret returns to the site and appropriate sets are updated. In the next section we give the algorithm description in detail which explains the actions of a ferret or a publicist and how they should interact with others under certain given conditions. P_iP_j represents a publicist P_j fired from processor P_i. P_iF_j represents a ferret F_j fired from processor P_i.

Before going to the step by step details of the algorithm, we give an informal description of the algorithm. The steps of the algorithm gives the actions that are to be performed by a processor or by the mobile agent. The algorithm gives the interactions between the agents and what type of information that has to be exchanged when an agent meets another agent. These interactions are between two publicists or two ferrets or a publicist and a ferret. The role of a ferret is to collect as much legal information (*Wids* along with the *Aids*) as possible and that of publicist is to advertise a *write* and meet other agents of similar interest and exchange the required information (*Wids* along with the *Aids*). Only the *Wids* (along with *Aids*) are collected by a ferret or a publicist. There is no exchange of partial history

between the agents at this time. The ferret does not tag any *Wid* as illegal while in the process of meeting other agents and collecting the required information. The ferret returns back to the processor/site and the *read* picks a *write* from the eligible list.

After the actual *write* is read by the *read*, the *reads from relation* is established and then the History Tree and the IWL of the *write* are collected. Based on the History Tree and the IWL, the processor/site updates its Illegality Set. An agent A can ignore another agent B (when met) if agent A has met agent B (directly or indirectly) before or agent A has met a more recent agent than agent B. This can be done by checking the information present with both the agents. A ferret carries with it two types of data: one is the $\alpha\beta$-legal and $\alpha\beta$-illegal lists (check the algorithm for a detailed explanation) and the other is the MDL. The $\alpha\beta$-legal and $\alpha\beta$-illegal lists are tagged to the ferret at the beginning and the data in the MDL gets accumulated as the ferret meets other agents. The $\alpha\beta$-legal and $\alpha\beta$-illegal lists are carried along with the ferret to make sure that the ferret does not meet with agents whose *writes* are already tagged as legal or illegal. The ferret keeps track of all the relevant agents it meets in the Meta Data List (MDL). Hence the ferret can make sure not to meet with an agent with whom it has already met. The *ids* in the MDL are provided to other agents (when met) who are interested. If a ferret likes to know more information or double check with a particular agent, it can do so by contacting that agent directly.

Two types of meta data are carried along with the publicist. One is the History Tree tagged at the beginning (when the publicist is fired) and the other is Meta Data List (MDL). The HT contains all (w.r.t to the processor which issued that write) the transitively preceding operations of the *write* that is advertised by the publicist. The HT can be used by other agents to determine whether they have met this agent (directly or indirectly) before and which *Wids* are relevant (in order to be collected) for them. The Meta Data List is empty at the beginning. Appropriate data (*i.e.*, the *Aids* and the corresponding *Wids*) is entered into the MDL only when the publicist meets other agents and exchange some information. The *Wid* advertised by a publicist and the *Wids* in the MDL of the publicist are w.r.t to the same object. Apart from advertising its own *write*, a publicist P_iP_j is also advertising other publicists (listed in the MDL of P_iP_j) involved in the same object as P_iP_j. Hence, if an agent meets publicist P_iP_j, it will also be made aware of other agents who have similar interest (w.r.t same objects) as P_iP_j. Therefore after meeting some relavent agents, the publicist P_iP_j has more information to give other agents (of similar interest) than at the beginning. In the next section we give a step by step explanation of our algorithm.

7.2 Algorithm

1. When a *write* operation $W_{ik}(x)$ (*i.e.*, *write* operation on object x issued by processor i) is executed,

   Actions to be taken by the processor

   (a) Processor i enters the *Wid* of $W_{ik}(x)$ into its Master Set (MS) and History

Set (HS).

(b) Check the History Set of processor i, and collect all the *read* and *write* operations on object x, say $R_{pq}(x)$ and $W_{mn}(x)$ that transitively precede $W_{ik}(x)$, (check also in the History Trees of the *writes* in the HS of processor i)

Note: The above two checkings are done for all the operations that lie between the previous *write* on x by processor i (if it exists) and $W_{ik}(x)$.

i. For each *read* operation $R_{pq}(x)$ collected, tag its respective *Wid* as RW-illegal in the Illegality Set of processor i.

ii. For each *write* operation $W_{mn}(x)$ collected, tag its *Wid* as WW-illegal in the Illegality Set of processor i.

(c) Form the History Tree (HT) and the Illegal Write List (IWL) of the *write* $W_{ik}(x)$ (refer to chapter 4 for a detail explanation),

i. History Tree (HT) of the *write* $W_{ik}(x)$ contains all the *read* and *write* operations from the History set of processor i. After forming the HT of the *write* $W_{ik}(x)$ and stored, the History Trees of other *writes* are removed.

ii. Illegal Write List (IWL) contains all the Wids from the Illegality Set of processor i (except those Wids w.r.t object x). The HT and IWL are tagged to the actual data while entering the *write* $W_{ik}(x)$ into the Master Set of processor i.

(d) Fire a mobile agent P_iP_j (publicist) advertising the *write* $W_{ik}(x)$. The agent carries along with it the HT of $W_{ik}(x)$. **Actions to be taken by**

the mobile agent

(e) If P_iP_j meets an agent say P_mP_n (publicist) advertising a *write* w.r.t the same object as P_iP_j (*i.e.,* object x),

i. If the *Aid* of P_mP_n exists in the MDL of P_iP_j, then ignore the publicist P_mP_n.

ii. If the *Aid* of P_mP_n does not exist in the MDL of P_iP_j, but if the *Wid* advertised by P_mP_n exists in the HT of P_iP_j, then ignore the publicist P_mP_n.

iii. If the *Aid* of P_mP_n does not exist in the MDL of P_iP_j and *Wid* of the *write* advertised by P_mP_n does not exist in HT of P_iP_j, then

A. P_iP_j enters the *Aid* of P_mP_n in its MDL along with the *Wid* advertised by P_mP_n.

B. Check the HT of P_mP_n and collect all the *Wids* on object x and enter them in the MDL of P_iP_j along with the corresponding *Aid*.

C. Compare the MDLs of P_iP_j and P_mP_n and copy the *Aids* (into the MDL of P_iP_j, along with the *Wid* being advertised) that are present in P_mP_n but not present in the MDL of P_iP_j.

iv. If P_iP_j meets an agent say P_sP_t (publicist) advertising a *write* w.r.t a different object than the publicist P_iP_j, but there exists *Wid* w.r.t object x in the HT of the *write* advertised by P_sP_t, then

A. If the *Aid* of P_sP_t exist in the MDL of P_iP_j, then ignore the agent P_sP_t.

B. If the *Aid* of P_sP_t does not exist in the MDL of P_iP_j, but if the *Wids* w.r.t object x (found in the HT of P_sP_t) exist in the HT of P_iP_j, then ignore the agent P_sP_t.

C. If the *Aid* of P_sP_t does not exist in the MDL of P_iP_j and the *Wids* w.r.t object x (found in the HT of P_sP_t) does not exist in the HT of P_iP_j, then

- Copy the *Wids* (w.r.t object x found in the HT of P_sP_t) along with the corresponding *Aid* in the MDL of P_iP_j.

v. If P_iP_j meets an agent say P_iP_t (publicist) (both the publicist are fired from the same processor), then

A. If both the agents are advertising the same objects, then compare the MDLs of both the publicists and exchange the *Aids* (along with the *Wid* being advertised) that are present with one agent but not with other.

B. If both the agents are advertising different objects, then ignore the agent P_iP_t.

2. When a *read* operation $R_{jk}(x)$ is issued by processor j requesting for $\alpha\beta$-legal *Wids*,

Note: $\alpha\beta$ can take any of the five legalities discussed (*i.e.,* RR, RW, WR, NO, or WW).

(a) Fire a mobile agent P_jF_m (Ferret) to get the requested information.

Actions to be taken by the mobile agent

(b) The agent carries with it all the $\alpha\beta$-legal (taken from the MS of P_j

excluding those tagged as $\alpha\beta$-illegal in the Illegality Set of P_j) and $\alpha\beta$-illegal (taken from the Illegality Set of P_j) *Wids* w.r.t object x.

(c) If agent P_jF_m meets an agent P_jP_q (agent advertising a *write* w.r.t object x, agents P_jF_m and P_jP_q are from the same processor j), then

i. Enter the *Wids* (along with their respective *Aids*) present in the MDL (Meta Data List) of P_jP_q into the $\alpha\beta$-legal list of P_jF_m except those already tagged as $\alpha\beta$-illegal or already present in the $\alpha\beta$-legal list or MDL of P_jF_m.

ii. Enter the *Wids* (along with their respective *Aids*) present in the MDL and $\alpha\beta$-legal list of P_jF_m but not present in the MDL of P_jP_q into the MDL of P_jP_q.

(d) If agent P_jF_m meets an agent P_jP_r (agent advertising a *write* w.r.t a different object than the one required by P_jF_m), ignore the agent P_jP_r.

(e) If agent P_jF_m meets an agent P_aP_l (agent advertising a *write* $W_{ai}i(x)$ written by processor a), then

i. If the *Wid* of the *write* $W_{ai}(x)$ exists in the $\alpha\beta$-legal or $\alpha\beta$-illegal list, then ignore the agent P_aP_l.

ii. If *Aid* of P_aP_l does not exist in the MDL of P_jF_m and *Wid* of the *write* $W_{ai}(x)$ does not exist in $\alpha\beta$-legal and $\alpha\beta$-illegal list, then

- Agent P_jF_m collects all the *Wids* (except those already present in the $\alpha\beta$-legal and $\alpha\beta$-illegal list) w.r.t object x from the HT of $W_{ai}(x)$ and are entered into the $\alpha\beta$-legal list of P_jF_m.

(f) If agent P_jF_m meets an agent P_aF_l (agent looking for information on

object x and required to satisfy any legality criterion), then

i. Compare the MDL, $\alpha\beta$-legal lists of P_jF_m and P_aF_l and exchange the *Wids* (along with their respective *Aids*) that are present with P_jF_m but not present with P_aF_l (excluding those present in $\alpha\beta$-illegal list).

(g) If agent P_jF_m meets an agent P_aF_m (agent looking for a different data object), then ignore the agent P_aF_m.

3. When agent P_jF_m returns to the site after collecting the information,

Actions to be taken by the processor

(a) Agent P_jF_m presents to the *read* $R_{jk}(x)$ a list of eligible legal *Wids* (w.r.t requested legality criterion and object x), *i.e.*, the $\alpha\beta$-legal list is given to $R_{jk}(x)$, then

i. All the newly tagged *Wids* to the $\alpha\beta$-legal list are copied into the MS of processor j.

ii. If $R_{jk}(x)$ picks $W_{pq}(x)$, then the actual data is read along with its HT and IWL. The HT of $W_{pq}(x)$ is stored in the MS of processor i along with the *write* $W_{pq}(x)$.

(b) *Rid* of $R_{jk}(x)$ and *Wid* of $W_{pq}(x)$ are entered into the History set (HS) of processor j.

(c) Check the IWL of $W_{pq}(x)$ and update accordingly in the Illegality Set of processor j. Any *Wid* in the IWL of $W_{pq}(x)$ tagged as illegal w.r.t a particular legality, should also be tagged as illegal in the Illegality Set of processor j. Delete the IWL of $W_{pq}(x)$.

(d) Scan the HS of $R_{jk}(x)$ or process order tree and collect all the *write* and *read* operations on object x that transitively precede $R_{jk}(x)$.

Note: This scanning is done for all the operations between the previous read on x (if it exists) by processor j and $R_{jk}(x)$.

i. For each read operation collected, tag its respective *Wid* as RR-illegal in the Illegality Set of processor j.

ii. For each *write* operation collected, tag its *Wid* as WR-illegal in the Illegality Set of processor j.

(e) Scan the History Tree of $W_{pq}(x)$ or reads from relation tree of $R_{jk}(x)$ and collect all the *read* and *write* operations on object x that transitively precede $R_{jk}(x)$,

i. For each read operation collected, tag its respective *Wid* as RR- and RW-illegal in the Illegality Set of processor j.

ii. For each *write* operation collected, tag its *Wid* as WR-, NO-, and WW-illegal in the Illegality Set of processor j.

7.3 Correctness Proof

When an operation is executed, it is clear from the step by step description of the algorithm given above that certain actions are performed by the processor and certain other actions by the mobile agent. The steps performed by the processor are same as the previous algorithm (given in the algorithm of chapter 6). So the proof for these steps is similar to the one given in chapter 6. However as seen from the description of the algorithm, the role of the mobile agents is to go and meet

other agents and collect information (relevant information). The mobile agents do not tag any *Wid* as illegal w.r.t any of the five legalities during their (mobile agents) life time. Therefore this part of the algorithm need not be proved.

7.4 Discussion

In the previous sections, we have seen how the agents (ferrets and publicists) behave under certain conditions and in a particular type of system. We have addressed the legalities in a system with just ferrets and publicists. Depending on the requirements of the application, a more complex system with different agents for different roles can be considered. One such agent called a "guru" [12] can be introduced into the system to increase the probability of sufficient like minded agents to meet. The role of a guru is to remember which agent had which interest and direct like minded agents to meet and negotiate. Another possibility is to introduce the concept of cloning the agents to reduce the load on a single agent.

In the next chapter we discuss in general the applicability of the legalities in various scenarios.

Chapter 8

Discussion And Conclusions

In this thesis, consistency of cooperative executions in terms of legalities of the *read* operations is defined. Two defining relations (ρ) are taken into consideration: i) *real-time order* ($\rho = \rightarrow_t$), ii) *causal order* ($\rho = (\rightarrow_i \cup \rightarrow_{rf})^*$). Detailed algorithms and their respective correctness proofs for each defining relation for selecting appropriate versions (*writes*) for each legality are also given. The three algorithms correspond to three different environments - centralized, distributed and mobile agent setups. Algorithms are given for two different types of systems: i) One in which the cooperating users are aware of each other and the communication medium is message broadcasting, and ii) The other environment where the participating users are not aware of each other and collecting and notification of information is through the mobile agents. The approach and the algorithms developed in the thesis will help in providing different levels of system support for cooperative executions. Some illustrations are given in the following sections.

8.1 Document Authoring

The underlying theme in the notion of consistency presented in the thesis is that, if all the *reads* are consistent, that is, legal in all respects (*completely-legal*), then all the *writes* are 'consistent.' Therefore, each *read* operation may simply want a completely-legal set of *writes* instead of specifying one or more legalities explicitly; this can be obtained by the intersection of the eligible sets for each legality. Furthermore, depending on the semantics of the variables and/or cooperative execution, certain legalities may not be essential with respect to some variables. (For example, any version of section x may be sufficient for writing section y in a document authoring environment). The system can keep track of such properties and select the writes appropriately. In some instances the user may settle for a lesser legal version than the latest version due to slow response time.

There may be cases when no completely-legal version is available. Then a 'somewhat legal' version can be given for the time being, and a 'more legal' version can be given later on. For example, we may have $W(x) \rightarrow W'(x) \rightarrow R(x)$, and so W is WW-illegal for R. However, the version of W' may not be available temporarily. Then the version of W may be given, and a later *read* of the version of W' can be forced by the system (assuming that W' makes a few more 'changes' in x, keeping all the changes W has made). In this case, conceptually, causal consistency of a sub-execution (like the useful sublog [28]), obtained by eliminating, for instance, the 'tentative' *reads* as described above, may be required of the cooperative execution.

Other internal consistency constraints like "the last version of an object produced by a transaction in a group must be read by all the other transactions in that group" and "any processor reading an intermediate version of x must also read the final version of x" of [28] can also be facilitated as follows: the system may keep track of the (legal) versions that must be read by a processor and prompt that processor to read.

Again, as mentioned earlier (in chapter 3), if all legalities are satisfied for all the *reads*, then the execution is causally consistent. It is shown that, under certain assumptions, the internal consistency requirements mentioned in [28] include a form of causal consistency. Depending on the semantics of the variables and/or cooperative execution, some illegalities may be tolerable for some reads, either temporarily (that is, the execution can be 'corrected' later on) or permanently. The CSCW system can keep track of these features and, with the help of the algorithm, suggest suitable values for the reads.

8.2 World Wide Web

The World Wide Web is developing at a furious pace, with new innovations appearing with every release of Web browser and server software. The Web was originally intended to support a richer, more active form of information sharing than is currently the case. The explosive growth of the World Wide Web and its penetration into academic, commercial and domestic environments is well documented. The

World Wide Web is considered to be a medium for information browsing, publishing, etc. The combination of a global addressing system, network protocol, document mark-up language and client-server architecture provides for a simple method for users to search, browse and retrieve information as well as share information of their own with others. However, all the concepts of the World Wide Web do not fully and directly support more collaborative forms of information sharing, where widely-dispersed working groups work together to jointly author, comment and annotate documents, and engage in other forms of collaboration such as group discussion. There are a number of reasons to suggest that support for such collaborative working based on information sharing is becoming more necessary. Trends in the current business world towards decentralization, joint ventures, outsourcing of business functions and so on are highlighting a need for effective methods of sharing information and coordinating activities. Hence many researchers are focusing on how to utilize and extend the Web technology to provide richer forms of cooperation.

Underlying any activity in the web is the finding of new or 'recent' data. Recentness may be due to new sources, up-to-dateness with respect to a time scale (e.g., hurricane watch), changes in organization (change in group membership), etc. Recentness may help keeping personalized web pages up-to-date, finding best commodities at best price, or even reorganizing the web itself in terms of redesigning web page, replicating the contents in different sites, etc. Recentness can be measured in different ways. (The variable in the context of this application could be a page, a URL, or any piece of identifiable data.) In the Web context, the illegalities can be called as obsoleteness or recentness. The five illegalities can be expressed

in terms of five obsoleteness (*i.e.*, RR-obsolete, RW-obsolete, WR-obsolete, WW-obsolete, and NO-obsolete). We give several examples illustrating the occurrences and usefulness of these recentness notions later.

One research group at GMD [3, 4] is focusing on how to transform the Web from a primarily passive information repository to an active cooperation tool. The Basic Support for Cooperative Work (BSCW) project at GMD is attempting to realize this potential through development of Web-based tools which provide cross-platform collaboration services to groups using Web technologies. In particular, one of the tools developed in the project is the *BSCW Shared Workspace System* -a centralized cooperative application integrated with an unmodified Web server accessible from standard Web browsers. The BSCW system supports cooperation through 'shared workspaces;' small repositories in which users can upload (write) documents, hold threaded discussions, and obtain information on the previous activities of other users to coordinate their own work.

The BSCW Shared Workspace system is an extension of a standard Web server through the server CGI Application Programming Interface. A 'BSCW server' (Web server with the BSCW extension) manages a number of shared workspaces; repositories for shared information, accessible to members of a group using simple name and password scheme. In general, a BSCW server will manage workspaces for different groups , and users may be members of several workspaces. A shared workspace can contain different kinds of information such as documents, pictures, URL links to other Web pages or FTP sites, threaded discussions, member con-

tact information and more. Members can transfer (upload) information from their machines to the workspace and set access rights to control the visibility of this information or the operations which can be performed by others. In addition members can download, modify and request more details on the information objects by clicking on one of the 'event icons' provided in the interface.

The *event service* of the BSCW systems is an attempt to provide users with information on the activities of other users, with respect to the objects within a shared workspace. Events are triggered whenever a user performs an action in a workspace, such as uploading a new document, downloading (reading) an existing document, renaming a document and so on. The system records the events, and presents the recent events to each user as event icons in the workspace listing. Each event icon captures different meaning and information about the objects in the workspace. The first two events are related to the ideas discussed in this thesis. The first event called the *new* event (an object that has been created or modified since the user last caught up or last read) directly corresponds to the NO-legality in *real time order* discussed in Chapter four. The second event called the *read* event (which shows that an object has been read by someone) does not directly correspond to any of the legalities discussed in this thesis. However it can be achieved by the current algorithm without much modification just by keeping track of an object and which processor has read it. These events can be caught up at different levels, for example real time order, causal order, etc. But real time order is inherent in the system described in [3]. As each of the five legalities give a different notion of the *recentness* of the values, using them as event icons in the workspace listing

seems appropriate and conveys more meaning to some applications.

8.3 Shared Health Care System

Consider the shared care system of a diabetic patient by a number of clinics and doctors who are networked together.

- A diabetic patient may be seeing several clinicians concurrently over a period of time. That is, a diabetic individual is treated for his/her diabetes and for other medical problems by different specialists over the course of the disease.
- Clinicians, for example, share the treatment of patients and supply test results and other information to one another. The requirement by different people involved may be different at different stages or instances of the treatment.

By providing a shared or collaborative care, we can avoid some inconsistencies in the system such as.

- Duplication of tests or any other information
- Omission of certain important facts
- Delays in communication
- Avoiding unwanted data or overload, etc.

The following actions take place in such a system,

- The patient monitors himself daily (weight, glucose levels, etc) and enters this data for reference to the doctors and others.
- A nurse or a program sketches a graph of the readings daily and reports to the doctor (GP or General Practitionar) or nurse of any unusual readings.
- The nurse might get an appointment with the GP who may further refer the patient to a specialist and/or to a dietician or to have some other tests done.
- The specialist and/or the dietician may check the patient and enter further comments or treatment to be administered.
- If any blood test or x-rays are to be done, then the respective labs perform the tests and enter that data into the system accordingly.

Some specific instances are considered below how the legalities can be applied to avoid certain inconsistencies in the system is explained:

1. Consider the following trivial case of the shared care system where the patient enters his daily readings (weight, glucose level) into the system and another processor say N, reads these readings regularly and sketches a graph (Figure 8.1). The writes by processor N are considered to be of incremental updates. So writing of the graph ($W_{N1}(x)$, $W_{N2}(x)$, etc.,) depends on reading the data written by the patient. In other words writing $W_{N1}(x)$ depends on the read $R_{N1}(x)[W_{P1}(x)]$. Hence processor N (nurse or a program), needs only WW-legal values for such an instance. Looking carefully at the Figure 8.1 $W_{P1}(x)$ is illegal w.r.t all the five legalities to a *read* operation after $R_{N2}(x)$ and $W_{P2}(x)$ is WW- and RW-illegal to a *read* operation after $W_{N2}(x)$.

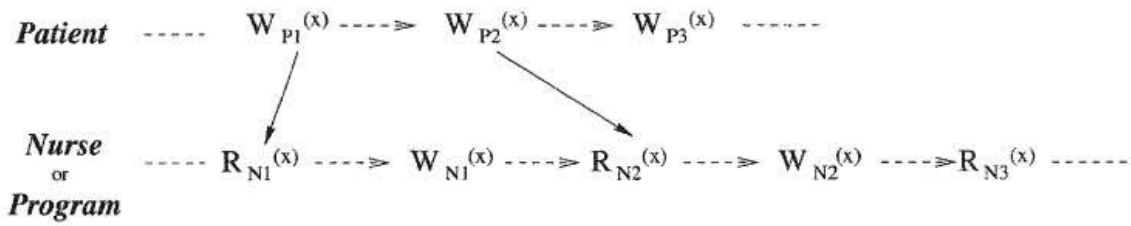

Figure 8.1: Shared Care System

2. Consider another instance where the patient is advised to take blood tests and x-ray (Figure 8.2). X-ray is done by the x-ray department and blood tests by the blood clinic. A radiologist reads the x-rays and writes his comments. Later the GP reads these reports and considers referring the patient to a specialist. So the GP inputs her comments along with the results from the x-ray and blood tests into the system. The specialist reads the comments ($W_{41}(z)$) from the GP and decides to take another x-ray and blood test. Later when the GP reads these latest test results and before commenting further (may be the GP bumped into the radiologist and after the conversation between them) decides to recheck with the radiologist the values read ($R_{44}(x)$ read $W_{22}(x)$) previously. So the next *read* operation ($R_{45}(x)$) by the GP has to request for a x-ray version that is legal w.r.t all the five legalities. Consider an extension to the partial executions given in the Figure 8.2. If $W_{12}(x)$ is the x-ray that is taken after the request from the specialist, then as usual the radiologist examines the x-ray and writes a report ($W_{22}(x)$ in Figure 8.2 which contains the x-ray plus the comments by the radiologists). The GP reads this report which says that there is no difference from the previous x-ray and hence writes the same into the patients record say $W_{42}(z)$. Later when the specialist reads this decides to have a look at the radiologists

report. Hence the radiologist request for a version that is legal in all respects *i.e.*, $W_{22}(x)$. All the others ($W_{12}(x)$, $W_{12}(x)$, $W_{21}(x)$) become illegal w.r.t all the five legalities due to the transitivity nature of the causal order.

Hence different instances require different legalities or combination of them to be satisfied.

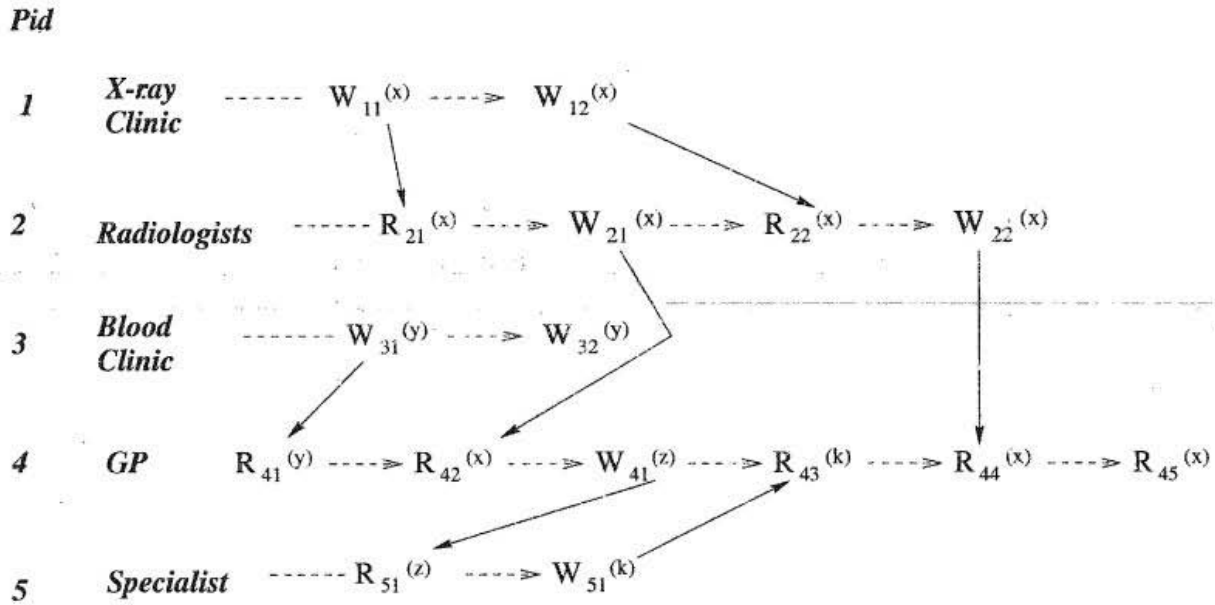

Figure 8.2: Shared Care System2

8.4 Other Examples

1. Consider an example of an online exam in which different sections have to be answered, and they will be given one at a time. The examiners have imposed a restriction that once a section is finished, and another section is taken up, you cannot go back to the former section. Then, all the previously solved

sections are no longer of any relevance to subsequent *reads*, or in other words RR-obsolete.

2. Consider an application where any *write* ($W2$) fully incorporates the knowledge gained through all the previous reads of W_1 ($R(W1)$). For such an application, any *read* that follows the *write* ($W2$) need not reread any *writes* that have already been read. In terms of the obsoleteness, all such previously read *writes* then become RW-obsolete.

   For example, take a project that compiles a comprehensive summary of a collection of data being made available in the form of different *writes* stored in a database. The compilation proceeds by creating partial compilations. Perhaps each partial compilation may address a particular sub-aspect. Each partial compilation (say $W2$) is also considered as new data for subsequent work and hence is written back into the database. However, if a partial compilation is completed based on collating the data contained in certain other *writes* collected by means of reading them (say $R(W1)$), any subsequent *read* need not reread any of the *writes* ($W1$) that have already been read for compiling the just completed *write* (W2). Hence, such *writes* ($W1$) that have been read and processed, become obsolete (*i.e.*, RW-obsolete) for further *reads*.

3. Consider a situation where the different *writes* of a given variable are nominally equivalent (that is, they contain the same information but may be presented in different formats) to each other. An example would be different

web sites on the net which contain information on the score of an ongoing Baseball game.

This illustrates a situation where different *writes* give information about the same item. A professional sports commentator group that regularly prepares statistical and other types of analysis of Baseball matches for a web site can access different sources on the web (newspapers, broadcast networks, privately owned web sites, etc) to get the scorecards. The different sources contain nominally equivalent basic information, although the actual packaging and presentation format from the source can differ widely. When a *read* is issued requesting the information about a particular match, the sports commentator is satisfied to get the information from any one of the available sources. After the choice is made and the information obtained, the remaining *writes* automatically are of no use in the sense that they cannot augment the information gathered by issuing the *read* request. In terms of the obsoleteness, all these *writes* are WR-obsolete for any subsequent *read*. So, any subsequent *read* should get a fresh set of *writes* (information).

Similar situation arises when a *read* operation is employed in order to fill a single specified slot by choosing from among a given choice of *writes* such that once the slot is filled, the remaining *writes* that exist at the time of the *read* have no more useful function and should not be read by a subsequent *read*.

4. Consider a web site which lists all the share prices of a number of stocks. Individual traders sitting at their desk (remotely) would like to view the latest changes in share prices of all or some of the stocks (of their interest). That is, once the share price of a particular stock changes, the old values are of no interest or use for the trader. Getting or knowing the latest value for a particular stock is important or required. In other words, all those previous values are WW-obsolete.

5. Consider a category of applications where a condition is based on the latest version of the data being read. That is, once a particular data is read say $R(W2)$, then all the previous intermediate versions which have contributed to this *write* $(W2)$ directly or indirectly should not be read by any subsequent *read*. For example, if a Baseball match is in progress and the scorecard is updated every few minutes. Each updation is called an intermediate result. If a sports columnist sitting at his desk remotely is accessing this site from time to time. Every time he accesses or reads the scorecard, he would like to see all the changes done since he last visited the site. That is in a way informing the system that he is familiar with all the previous changes and that he does not want to see them anymore. In terms of the obsoleteness, NO-obsoleteness provides such support to the user.

8.5 Conclusion

This thesis begins with an overview of Computer-Supported Cooperative Work and gives the survey of some of the charecteristics of the cooperative systems. We give a review of some of the consistencies proposed in the literature for cooperative systems and present a new approach to specify consistency of cooperative executions. It is based on the intuitive notion of legality of the read operations. Five different notions of 'recentness' or 'obsoleteness' of the values have been presented for two defining relations.

When all reads are legal in all five respects, the execution is said to be *ρ-causal*, and if this property holds with respect to "exclusion-closure" of ρ, then the execution is *ρ-atomic*. When ρ is the global real time order, ρ-atomicity is *linearizability* [17], and when ρ is the transitive closure of the union of the *process order* and *reads-from relation*, ρ-causality is *causal consistency* [38], and ρ-atomicity is *sequential consistency* [24]. Thus the five notions of recentness are not just intuitive and meaningful, but also 'complete'.

The recentness notions can be defined with respect to some special writes (instead of all the writes) and, similarly, with respect to only some reads. This would amount to different defining relations ρ. This will facilitate, in various ways, notifications in collaborative work [8], collaborative browsing [44], etc.

Bibliography

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *"Concurrency Control and Recovery in DataBase Systems,"* Addison-Wesley Publishing Company, 1987.

[2] N. S. Barghouti, and G. E. Kaiser, *"Concurrency control in advanced database applications"*, in Computing Surveys, 23(3), 1991, pp. 269-317.

[3] R. Bentley, W. Applet, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor, and G. Woetzel, *"Basic Support for Cooperative Work on the World Wide Web"*, in International Journal of Human Computer Studies: Special issue on the Novel Application of the WWW, Spring 1997, Academic Press, Cambridge.

[4] R. Bentley, T. Horstmann, and J. Trevor, *"The World Wide Web as enabling technology for CSCW: The case of BSCW"*, in CSCW: the Journal of Collaborative Computing, 2-3, 1997, Kluwer Academic Press, Amsterdam.

[5] P. M. Cashman, and D. Stroll, *"Developing the management system of the 1990s: The role of collaborative work,"* in Technological Support for Work Group Collaboration, M. H. Olson, Ed., Lawrence Erlbaum Associates, Publishers, Hillsdale, N.J., 1989, pp. 129-146.

[6] P. Dewan, and R. Choudhary, *"Primitives for programming multi-user interfaces"* in User Interface Software and Technology (UIST), Nov. 11-13, Hilton Head, South Carolina, 1991, ACM Press, pp. 69-78.

[7] P. Dourish, *"Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit,"* in Proc. ACM Conference on Computer-Supported Cooperative Work, CSCW'96, Boston, USA, 1996.

[8] C. A. Ellis, S. J. Gibbs, and G. L. Rein, *" Groupware - Some Issues and Experiences,"* in Communications of the ACM, 34(1), January 1991, pp. 38-58.

[9] T. Ferwagner, *"Experiences in designing the Hohenheim CATeam room,"* in J. M. Bowers, and S. D. Benford, editors, Studies in Computer Supported Cooperative Work, North-Holland Publishers, Amsterdam, 1991.

[10] R. Fish, R. Kraut, M. leland, and M. Cohen, *"Quilt: A collaborative tool for cooperative writing,"* in Proc. of the conference on Office Information Systems, (Palo Alto, California, March, 23-25), ACM, New York, 1988, pp. 30-37.

[11] F. Flores, M. Graves, B. Hartfield, and T. Winograd, *"Computer systems and the design of organizational interaction,"* in ACM Transactions of Information Systems, 6(2), April 1988, pp. 153-172.

[12] D. J. Grey, and R. I. Ferguson, *"AgentSpace: A toolkit for constructing mobile agent systems,"*, University of Sunderland, School of Computing, Engineering and Technology, Occasional Paper Series, No. SCET-1999-14, 1999.

[13] D. J. Grey, P. Dunne, and R. I. Ferguson, *"WebSeeker: a means of efficiently locating resources on the World Wide Web using mobile, collaborative agents,"*, in Proc. ASIM Agent-Based Simulation Workshop, May2-3, 2000.

[14] I. Grief, R. Seliger, and W. Weihl, *" Atomic data abstractions in a distributed collaborative editing system,"* in Proc. of the 13th Annual Symposium on Principles of Programming Languages, (St. Petersburg, Fla., January 13-15), ACM, New York, 1986, pp. 160-172.

[15] S. Haldar and K. Vidyasankar, *"Unified Consistency Specifications of Read/Write Shared Variables,"* Internal Report, Department of Computer Science, Memorial University of Newfoundland, Canada 1998.

[16] S. R. Hiltz, and M. Turoff, *"Structuring computer-mediated communication system to avoid information overload,"* in Communications of the ACM, 28(7), July 1985, pp. 680-689.

[17] M. P. Herlihy and J. M. Wing, *"Linearizability: A correctness condition for concurrent objects,"* in ACM TOPLAS, Vol. 12(3), 1990, pp. 463-492.

[18] J. Trevor, *"Infrastructure Support for CSCW,"* Ph.D thesis, 1994.

[19] G. E. Kaiser, S. M. Kaplan, and J. Micallef, *"Multiuser, distributed language-based environments,"* in IEEE Software, 4(6), November 1987, pp. 58-67.

[20] B. Karbe, N. Ramsperger, and P. Weiss, *"Support for Cooperative Work by Electronic Circulation Folders,"* in Proc. of Conference of Office Information Systems, (Cambridge, MA), ACM Press, New York, April 1990, pp. 109-117.

[21] M. J. Knister, and A. Prakash, *"DistEdit: A distributed toolkit for supporting multiple group editors,"* in Proc. of the third conference on Computer-Supported Cooperative Work, (Los Angeles, California, October 8-10) , ACM, New York, 1990.

[22] K. L. Kraemer, and J. L. King, *"Computer-based systems for cooperative work and group decision making,"* in ACM Computing Survey, 20(2), June 1988, pp. 115-146.

[23] M. Koch, *"Design issues for a distributed multi-user editor,"* in the international journal of Computer-Supported Cooperative Work, 5(1), 1996.

[24] L. Lamport, *"How to make a multiprocessor computer that correctly executes multiprocess programs,"* in IEEE TC, Vol. C-28(9), 1979, pp. 690-691.

[25] M. D. P. Leland, R. S. Fish, and R. E. Kraut, *"Collaborative document production using Quilt,"* in Proc. of the conference on Computer-Supported Cooperative Work, (portland, Oregan, September 26-28), ACM, New York, 1988, pp. 206-215.

[26] T. W. Malone, K. R. Grant, F. A. Turbak, S. A. Brobst, and M. D. Cohen, *"Intelligent information-sharing systems,"* in Communications of the ACM, 30(5), 1987, pp. 390-402.

[27] P. Molli, *"COO-Transactions: Supporting Cooperative Work,"* in 7th International Workshop on Software Configuration Management (SCM7), LNCS. Springer-Verlag, 1997.

[28] P. Molli, M. Munier, G. Canals, F. Charoy, and C. Godart, *"COO-serializability: A Correctness Criterion for Cooperative Executions,"* Technical Report, Centre de Recherche en Informatique de Nancy, 1997.

[29] E. Mortensen, *"Trends in electronic Mail and its role in office automation,"* in Electronic Publishing Review, Vol 5, No 4, December 1985, pp. 257-268.

[30] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, *"Issues in the design of computer-support for co-authoring and commenting,"* in Proc. of the third conference on Computer-Supported Cooperative Work (CSCW'90), ACM Press, 1990, pp. 183-195.

[31] C. M. Neuwirth, D. S. Kaufer, D. S. Erion, P. Morris, and D. Miller, *"Flexible diff-ing in a collaborative writing system,"* in Proc. of the fourth conference on Computer-Supported Cooperative Work (CSCW '92), ACM Press, 1992, pp. 147-154.

[32] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris, *"Computer Support for Distributed Collaborative Writing: Defining Parameters of Interaction,"* In Computer-Supported Cooperative Work (CSCW '94), Chaphill, North Carolina, USA.

[33] M. Nodine, S. Ramaswamy, and S. Zdonik, *"A Cooperative Transaction Model for Design Database,"* in Database Transaction Models for Advanced Applications, A. Elmagarmid, Ed. Morgan Kauffman, 1992.

[34] Tim Oates, M.V. Nagendra Prasad, and Victor R. Lesser, *"Cooperative Information Gathering: A Distributed Problem Solving Approach,"* in UMass Computer Science Techinical Report 94-66.

[35] L. Osterweil, *"Automated support for the enactment of rigorously described software process,"* in Proc. of the fourth International Software Process Workshop, (Devon, UK, May 11-13, 1988), ACM SIGSOFT, 14, 4, June 1989, pp. 122-125.

[36] C. Papadimitriou, *"The Theory of Database Concurrency Control,"* Computer Science Press Inc, 1986.

[37] M. Raynal, and A. Schiper, *"From Causal Consistency to Sequential Consistency in Shared Memory Systems,"* in Proc. of the 15th Int. Conf. FST-TCS (Foundations of Software Technology and Theoritical Computer Science), Bangalore, India, Springer-Verlag LNCS 1026, (P.S. Thiagarajan Ed.), December 1995, pp. 180-194.

[38] M. Raynal, G. Thia-kime, and M. Ahamad, *"From Serializable to Causal Transactions for Collaborative Applications,"* in Proc. of the 23rd EUROMICRO Conference, Budapest, 1997, pp. 314-321.

[39] Rolf de By, Wolfgang Klas, Jari Veijalainen(eds), *"Transaction Management Support for Cooperative Applications,"* Kluwer Academic Publishers, December 1997.

[40] B. Singh, *"Invited talk on coordination systems,"* in Organizational Computing conference, (Austin, Texas), November 1989, pp. 13-14.

[41] N. A. Streitz, J. M. Haake, J. Hannemann, A. Lemke, W. Schuler, H. Schutt, M. Thuring, *"SEPIA: A Cooperative Hypermedia Authoring Environment,"* in Proceedings of the ACM conference on Hypretext, November 30-December 04, Milan, Italy, 1992, pp. 11-22.

[42] J. M. Haake and B. Wilson, *"Supporting Collaborative Writing of Hyperdocuments in SEPIA,"* in Proceedings of the conference on Computer-supported cooperative work, November 1992, Toronto, Canada.

[43] C. Sun, X. Jia, Y. Zhang and Y. Yang, *"A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems,"* in Proc. of International ACM SIGROUP Conference on Supporting Group Work, Phoenix, Arizona, USA, 1997, pp.425-434.

[44] M. B. Twindale, D. M. Nichols, and C. D. Paice, *"Browsing is a Collaborative Process,"* in Information Processing and Management, 33(6), 1997, pp. 761-783.

[45] Tore Urnes and Roy Nejabi, *"Tools for Implementing Groupware: Survey and Evaluation,"* in Technical Report No. CS-94-03, York University, North York, Ontario, Canada.

[46] Tom Rodden, *"A Survey of CSCW Systems,"* in Interacting with Computers, 3(3), December 1991, pp. 319-353.

[47] W. Prinz, A. McGrath, A. Penn, P. Schickel and F. Wilhelmsen, *"TOWER - Theatre of Work Enabling Relationships,"* in Proceedings of eBusiness and eWork, Madrid 2000.

[48] W. Prinz, *"NESSIE: An Awareness Environment for Cooperative Settings,"* in Proceedings of The Sixth European Conference on Computer Supported Cooperative Work - ECSCW'99, S. Bdker, M. Kyng, and K. Schmidt (eds.), Kluwer Academic Publishers, 1999, pp 391-410.

[49] J. Wäsch and W. Klas, *"History Merging as a Mechanism for Concurrency Control in Cooperative Environments,"* RIDE-NDS, 1996.

[50] P.Wilson, *"Computer Supported Cooperative Work: An Introduction,"* Oxford, UK:Intellect Books, 1991.