Nested Group Communication for Wide-area Networks

By

Luping Liang

B.Eng., Tsinghua University, China, 1982
M.Math., University of Waterloo, Canada, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
March 3, 1992
© Luping Liang, 1992
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science
The University of British Columbia
Vancouver, Canada

Date March 6 1992

DE-6 (2/88)
To my motherland China,
To those ordinary people in Beijing,
To my parents and my wife.
Abstract

Group communication concerns sending messages to receiver groups in distributed systems. A process group comprises a set of processes and encapsulates their internal interactions, thereby simplifying the interactions between user programs and groups of receiving processes. Although the basic idea was proposed a few years ago, few systems or applications take advantage of it due to a lack of a comprehensive understanding of the requirements of group communication with respect to different classes of applications, and a lack of operating system support to meet those requirements. This dissertation consists of two parts that address these deficiencies.

The first part provides a comprehensive understanding of process groups by examining their potential applications, and the requirements to the system support expected by these applications. Groups are classified based on their structure and behavior. Also, a uniform treatment of grouping transparency is presented.

The second part of this dissertation focuses on a particularly important aspect of group communication — group naming in an internet. For the purposes of maintaining subnet autonomy and reducing traffic on internet links, a nested group model is proposed to allow internet groups to contain other groups as members. By formalizing this nested group model using a name graph, two problems in group name resolutions are identified: resolution loops and resolution duplications. After analyzing existing methods (centralized vs. distributed dynamic methods) and identifying their deficiencies, we propose a novel distributed static method. Instead of detecting loops at the time of name resolution, the static method transforms the system view of the name graph into a special structure which is updated whenever there is a change in group membership. To guarantee correctness, the name graph transformation preserves the property that name resolutions based on the system's view of the name graph are consistent with respect to the users' view. Based on the assumption that name graph updates occur much less frequently than name resolutions, static method trades a higher overhead of name graph updates for a better performance of name resolution to gain an improved over all group message transport performance. In this part of the dissertation, a static shadow tree algorithm is designed and analyzed. The correctness arguments for the algorithm are provided and aspects such as concurrency control and failure handling in name graph updates are investigated as well. A prototype implementation of the algorithm is conducted as an existence proof to demonstrate implementation feasibility and to test the behavior of the algorithm.
Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgement ix

1 Introduction 1

1.1 Thesis Motivation and Goals 3

1.2 Summary of Contributions 5

1.3 Thesis Organization 6

2 Classifications and Requirements 7

2.1 Process Group Model 8

2.2 Structural Classification 10

2.3 Behavior Classification and Requirements 12

2.3.1 Deterministic Groups 13

2.3.2 Nondeterministic Groups 23

2.4 Discussion 30

2.5 Related Work 32

2.6 Chapter Summary 33

3 Internet Group Name Resolutions 34

3.1 Nested Group Model 35

3.2 Group Naming Model 36

3.2.1 Group Name Graph 37

3.2.2 Group Name Resolution 38

3.2.3 Resolution Loops 39

3.2.4 Duplication Loops 40
3.3 A Taxonomy on RL and DL Control ........................................... 41
  3.3.1 Centralized Approach .............................................. 42
  3.3.2 Distributed Dynamic Approach ............................... 43
  3.3.3 Distributed Static Approach .................................. 45
3.4 Related Work ........................................................................ 47
3.5 Chapter Summary ................................................................. 47

4 Spanning Shadow Tree Algorithm ........................................... 49
  4.1 Name Resolution Consistency ........................................... 50
  4.2 Handling Resolution Duplications ................................... 51
  4.3 Handling Resolution Loops ............................................. 53
  4.4 Representing Topology of Ancestors ................................ 55
    4.4.1 Virtual Node and Derived Name Graph ...................... 56
    4.4.2 Pathnames ......................................................... 58
  4.5 Detecting RLs and DLs ................................................... 60
  4.6 Join Operation ............................................................. 61
    4.6.1 VN Topology Information ...................................... 61
    4.6.2 VN Shadow Tree Construction ................................ 62
    4.6.3 Operation Description .......................................... 63
    4.6.4 Handling Parallel Arcs ......................................... 65
  4.7 Leave Operation ............................................................. 66
    4.7.1 Determining Retained VN ....................................... 68
    4.7.2 Operation Description .......................................... 70
  4.8 Correctness ........................................................................ 71
  4.9 Related Work ..................................................................... 73
  4.10 Chapter Summary ............................................................. 73

5 Analysis and Experiments ....................................................... 75
  5.1 Communication Complexity .............................................. 75
    5.1.1 Worst Case Assumptions ........................................ 75
    5.1.2 Message Complexity ............................................. 77
    5.1.3 Discussion ............................................................ 80
  5.2 Prototype Implementation ................................................ 81
    5.2.1 Environment .......................................................... 81
    5.2.2 Prototype Structure .............................................. 82
    5.2.3 Prototype Limitations ............................................ 83
  5.3 Testing Experiments ........................................................ 83
    5.3.1 Random Testing ..................................................... 83
    5.3.2 Special-Case Testing ............................................. 84
  5.4 Chapter Summary ............................................................. 90
# Concurrency and Resiliency

6.1 Concurrency Control Issues .......................... 91
6.2 Update Ordering Protocol ............................ 97
   6.2.1 Description ...................................... 98
   6.2.2 Discussion ....................................... 100
   6.2.3 Correctness ..................................... 105
   6.2.4 Message Complexity ............................. 107
6.3 Resiliency Support Issues .......................... 108
6.4 Name Graph Update Protocol Resiliency .............. 111
   6.4.1 Making Ordering Determination Resilient ........... 111
   6.4.2 Making Execution Resilient ....................... 111
   6.4.3 Dealing with Failures during Update Execution ... 113
   6.4.4 Message Complexity ............................. 114
6.5 Name Graph Resiliency .............................. 115
   6.5.1 Handling Normal Arc Failures .................... 115
   6.5.2 Handling VN Partial Failures .................... 116
6.6 Name Resolution Resiliency ........................ 118
   6.6.1 Name Resolution Consistency ..................... 119
   6.6.2 Name Resolution Atomicity ....................... 120
6.7 Related Work ........................................ 123
6.8 Chapter Summary .................................... 124

# Conclusions and Future Research

7.1 Summary of Results .................................. 126
7.2 Limitations ......................................... 129
7.3 Future Research .................................... 130

A Group Name Grammar Model .......................... 145

B Prototype Experiments ............................... 147

C Glossary .............................................. 162
List of Tables

2.1 Deterministic vs. nondeterministic ........................................ 29
2.2 Sample applications based on the classifications ............................ 30
6.1 Originator, working arc and working set .................................... 92
List of Figures

1.1 One-to-many communication. ........................................ 1
2.1 Inter and intragroup communication. .............................. 9
2.2 Many-to-many (group-to-group) communication. .................. 9
4.1 Examples of duplication loop control. .............................. 52
4.2 The new name resolution procedure. .............................. 54
4.3 An example of the shadow tree scheme. ............................ 56
4.4 Examples of pathnames. ............................................. 59
4.5 Spanning tree construction. ......................................... 62
4.6 An example of VN breaking by a leave operation. ............... 69
5.1 The worst case name graph topology. .............................. 76
5.2 Worst case overhead analysis of the join. ......................... 78
5.3 Worst case overhead analysis of the leave. ....................... 79
5.4 Test cases for the join operation. ................................ 86
5.5 Test cases for the leave operation. ................................ 88
6.1 Examples of interference to join operation. ...................... 93
6.2 Generation of irrelevant messages after deleting < u, v >. .... 103
B.1 Join experiment 1. .................................................... 148
B.2 Join experiment 2. .................................................... 149
B.3 Join experiment 3. .................................................... 150
B.4 Join experiment 4. .................................................... 151
B.5 Join experiment 5. .................................................... 152
B.6 Join experiment 6. .................................................... 153
B.7 Join experiment 7. .................................................... 154
B.8 Join experiment 8. .................................................... 154
B.9 Leave experiment 1. ................................................... 155
B.10 Leave experiment 2. .................................................. 156
B.11 Leave experiment 3. .................................................. 157
B.12 Leave experiment 4. .................................................. 158
Acknowledgements

I am deeply indebted to my co-supervisors Dr. Samuel T. Chanson and Dr. Gerald W. Neufeld, for their personal friendship and professional support throughout my graduate studies at UBC, their encouragement which helped me in gaining confidence in myself, and their guidance which taught me how to conduct scientific research. Without Sam and Gerald, this thesis would never have existed.

I am also very grateful to other members in my thesis committee, Dr. Mabo Ito and Dr. Alan Wagner, for their insightful comments and suggestions. I appreciate Dr. Larry Peterson at the University of Arizona for his serving as the external examiner.

Many other people at UBC have been very supportive. Many thanks to Ian Cavers, Wei Chen, Francois Jalbert, Hilde Larsen, Hongbing Li, Geng Lin, Gang Liu, Ying Lu, Runping Qi, Yun Xie, Cheng Yan, Jiansheng Zhao and Ying Zhang and many others for their sharing the good time and smile with me. Their good wishes and positive encouragements always cheer me up. Thanks to Teresa Przytycka, Roger Chin, and Felix Jaggi for lending me their ears and their constructive comments, and to Wenjing Zhu for trying to crash the prototype. Thanks to Dr. Barry Brachman for carefully reading this thesis and making corrections, and to Donald Acton, Murray Goldberg, Norman Goldstein, Art Pope and Mike Sample for helping my English. Thanks to Peter Phillips for keeping SunDraw working and George Phillips for helping me to get my \LaTeX running at BNR.

ChinaNet, which carried the technical discussions on building network connections to China [Quarterman 90], and China News Digest (CND), which brings the daily news about China to thousands of oversea Chinese students and scholars all over the world, are electronic mail distribution lists built and maintained by tens of volunteers in North American universities. Much experience was gained from my technical involvement in setting up and managing both distribution lists. I wish to extend my thanks to those volunteers who worked together with me in these projects.

I would never become what I am without the unconditional love, understanding and support from my parents. Their letters from home have always been a constant source of strength and courage, and their determined faith in me has inspired me every step in my life. Above all, it is beyond the power of word to express my debt to the person closest to my heart, my wife Yueli, for her patience, her encouragement, and her endless love. I cannot imagine what it would have been like without her.

The research of this dissertation was supported by grants from the Natural Sciences and Engineering Research Council of Canada, the Graduate Fellowship from the University of British Columbia and the Fellowship from BC Tel.
Chapter 1

Introduction

In computer networks, *multicast* (one-to-many communication) is a message transmission mechanism that delivers a message from a single source to a set of destinations. Special cases of multicast are *unicast* (one-to-one communication) and *broadcast* (one-to-all communication). As Figure 1.1 shows, a single multicast transmits a message to a set of destinations in parallel.

![Diagram of One-to-Many Communication](image)

*Figure 1.1: One-to-many communication.*
allowing the receivers to process the message concurrently.

The fact that multicast reduces communication overhead and provides better concurrency than unicast can be seen from the following. The communication overhead of an operation is generally measured in terms of cost of bandwidth — the number of messages the operation triggers, and cost of host loading — the number of packet events that occur at the host(s) involved during execution of the operation. Packet events are defined as message transmissions and receptions [Mockapetris 83, Zwaenepoel 84]. To multicast to \( N > 1 \) receivers and receive reply from each, the sender does one multicast send\(^1\) and receives \( N \) replies. and each receiver does one receive and one reply after the message has been processed. Assuming no message is lost, the host loading cost is \( 1 + 3N \) packet events and the bandwidth cost is \( 1 + N \) messages. Sending a message to the same set of receivers using unicast results in \( 4N \) packet events in host loading and \( 2N \) messages in bandwidth cost. Furthermore, all these packet events often occur sequentially, which is not acceptable in many applications [Satyanarayanan and Siegel 90].

Multicast is a network abstraction; group communication is an operating system abstraction. Generally speaking, a group is a set of objects sharing common application semantics, as well as the same group identifier or multicast address.\(^2\) Therefore, each group can be viewed as a single logical entity, without exposing its internal structure and interactions to users. Generally, objects are grouped for (i) abstracting the common characteristics of group members and the services they provide; (ii) encapsulating internal state and hiding interactions among group members from the clients so as to provide a uniform interface to the external world; and (iii) using groups as building blocks to construct larger system objects. The exchange of messages among groups is called intergroup communication (IGC).

Group communication offers improved efficiency and convenience.

- It provides a high-level communication abstraction to simplify the interaction between user programs and a group of receivers (for example, instead of using multiple one-to-one

\(^1\) Assume broadcast medium such as Ethernet [Metcalf and Boggs 76].

\(^2\) Multicast exists at the medium access sublayer. The mapping between group identifiers and multicast addresses is implementation dependent and not necessarily one-to-one. See Section 2.3.2 for examples.
send operations to deliver a message to individual members in a group, only one group-send operation needs to be performed in the user program and only the group id has to be known).

- It hides the organization of a group (e.g., membership of the group) and internal coordination among group members (e.g., synchronization among members) from applications.

- It provides a high-level abstraction to take advantage of the multicast capability of a network, thereby, reducing the costs of network bandwidth and host loading, and increasing message processing concurrency.

Group communication is best supported by network multicast. It can also be emulated by unicast, or simulated by network broadcast. However, when emulated by unicast, not only is there higher overhead in delivering and processing messages sequentially, but the sender must keep track of group membership; thus, a group is no longer self-contained. With network broadcast, extra host loading overhead is required because all machines must examine every message regardless of its destination; furthermore, the communication is less secure since group messages can be seen outside a group.

1.1 Thesis Motivation and Goals

Although multicast was introduced some years ago, the advantages of group communication have not been fully realized. The reasons for this are:

1. a lack of understanding of the requirements of group communication with respect to different classes of applications;

2. the fact that few systems provide sufficient group communication support at the operating system level to meet these requirements; and

3. until recently, a lack of adequate multicast hardware support.
CHAPTER 1. INTRODUCTION

This thesis addresses only the first two problems. The first part of the thesis provides a comprehensive discussion of process groups. Potential applications of process groups and the required group communication support are examined. Groups are classified according to their internal structure and external behavior. Furthermore, a uniform treatment of grouping transparency is presented.

The second part of the thesis investigates internet group naming support. In an internet environment, an internet group $G$ may contain members in more than one subnet. The subnets are interconnected via relatively expensive internet links. In order to save internet link bandwidth, reduce message transmission delay, and support subnet/subdomain autonomy, we propose to organize internet groups using a nested naming structure so that members in each subnet/subdomain of a group $G$ constitutes a subgroup. A message sent to $G$ results in a single message being sent to each subgroup, where the message is delivered to the local members. Naturally, a group may contain other groups. The membership in a subgroup normally need not be known outside its subnet/subdomain for reasons of autonomy. Distribution lists in messaging systems are examples of nested grouping.

Unfortunately, there are two undesirable problems with the nested group structure:

1. A nested group may include itself directly or indirectly, forming a recursive group. This often occurs in distribution lists in message handling systems. For example, a mailing list $G_1$ at the University of British Columbia may contain a mailing list $G_2$ at the University of Waterloo (as its member), which may in turn contain a mailing list $G_3$ at the University of Toronto. Without knowing that $G_2$ is a subgroup of $G_1$, the list manager at the University of Toronto may include $G_1$ in $G_3$, creating a recursive group — a group that contains itself as a subgroup. When recursive groups are allowed, delivering a message to such a group may become an infinite process; that is, a name resolution request for a recursive group may never complete.

2. Another problem is duplicate message propagations because of duplicate name resolutions. This problem occurs when a group $G_2$ is included in at least two other groups that are
themselves included either directly or indirectly in the same supergroup $G_1$, so that a message to $G_1$ will be sent to $G_2$ more than once through the different subgroups of $G_1$, and every member of $G_2$ will receive multiple copies of the same message. Although this problem does not prevent the network from functioning correctly (because duplicate messages may be detected by receivers using, for example, a message sequence number), it does cause unnecessary message propagation that wastes internet bandwidth, especially when $G_2$ contains many subgroups.

An examination shows that existing approaches do not handle these problems correctly. The second part of this thesis proposes a new approach to solving these problems. Issues associated with the new approach, such as concurrency control and failure handling, are also discussed.

1.2 Summary of Contributions

The major contributions of this thesis are summarized below:

1. Analysis and classification of process groups on the basis of the group internal structure and the external behavior. This provides a comprehensive understanding of process groups with respect to their potential applications and the requirements to group communication mechanism. This classification also provides a uniform treatment of grouping transparency.

2. Extension of groups to a nested structure for supporting efficient internet group communication as well as subnet/subdomain autonomy. Based on a formal name graph model, problems related to group name resolution in an internet environment are identified and analyzed. Existing approaches are investigated and their shortcomings analyzed. A new static approach to handling these problems is proposed.

3. A thorough study of the design, implementation and analysis of static algorithms, which avoids resolution loops and controls resolution duplications for internet group communication. This also includes the design of concurrency control and failure handling protocols.
in group membership updates, as well as the protocol for atomic and ordered group communication.

1.3 Thesis Organization

The remaining six chapters are organized as follows. In Chapter 2, process groups are analyzed and classified according to their internal structure and external behavior to provide a better understanding of group applications and their system support requirements. In Chapter 3, a nested structure for internet groups is proposed to reduce communication overhead on internet links and support autonomy in subnets/subdomains. Furthermore, problems related to the name resolution of nested groups are investigated, and formal models as well as correctness criteria for internet group naming are proposed. A taxonomy of existing approaches to dealing with resolution loops and message duplications is also provided. In Chapter 4, a shadow tree algorithm is designed. It is a distributed static algorithm which controls resolution loop and resolution duplication at the time when the name graph is updated. In Chapter 5, the communication complexity of the shadow tree algorithm is analyzed, and the implementation and testing of a prototype are reported. The aspects of concurrency control and failure handling are investigated in Chapter 6. Chapter 7 contains the conclusions of the thesis and an outline of future work. An up-to-date bibliography of group communication is included at the end of the thesis.
Chapter 2

Classifications and Requirements

In Chapter 1, we saw that group communication using multicast provides many advantages over broadcast and unicast communication methods. To build a group communication mechanism in distributed systems, we need a good understanding of the kind of applications for which groups will be useful and the kind of operating system support expected by these applications. This chapter is intended to serve this purpose.

In this chapter, process groups are classified based on an analysis of group internal structure and applications. The organization of this chapter is as follows. The process group model is defined in Section 2.1. In Section 2.2, we classify groups on the basis of the internal homogeneity among group members. A group is classified into one of four categories: data and operation homogeneous, operation homogeneous only, data homogeneous only and heterogeneous. In Section 2.3, different distributed applications of groups and their group communication requirements are examined. Here, groups are classified into deterministic and nondeterministic categories according to their external behavior. Grouping transparency is identified as a desirable property commonly expected by group applications. The relationship between the two classifications is discussed in Section 2.4. Section 2.5 contains a brief overview of the related work in the area.

---

1A modified version of this chapter has been published [Liang et al 90a].
2.1 Process Group Model

A process group is defined as a set of processes that cooperatively provide a service. Each process, serving as an object manager, maintains a set of objects. Generally, an object can be characterized by a set of variables and a set of operations defined on those variables. Object managers control the way that objects are accessed; that is, users have to send their requests to the object managers to execute the operations associated with the objects managed by the managers. Object managers in the same group share the same group id. They may have to coordinate among themselves to maintain consistency among the objects that they maintain. It is sometimes useful to have an object be accessible to more than one group. Therefore, a process (the manager of the shared object) can be a member of several groups and process groups can overlap.

In distributed systems, machine boundaries prevent processes on different hosts from physically sharing memory. In the following discussions, unless stated otherwise, we assume that interactions with process groups are through message passing. The underlying network can be either a local area network (LAN) or an internet. We make no assumption regarding the implementation of group communication.

A group is closed when only its members are allowed to send requests to it; otherwise, it is open. In this chapter, we assume the open model, as it is more general and corresponds to the client-server model commonly used in distributed operating systems.

In the client-server context, the process invoking an operation is called a client and the process receiving and processing the invocation is called a server. A process can play both client and server roles, depending on its communication context. This client-server model can be extended to group communication; that is, client group and server group can be defined similarly. As Figure 2.1 indicates, communication between external clients and a server group is called intergroup communication and internal communication among group members is known as intragroup communication. Intergroup communication could also occur between groups in

\[2\] Other communication paradigms such as remote operation invocation can be implemented on top of message passing.
CHAPTER 2. CLASSIFICATIONS AND REQUIREMENTS

Figure 2.1: Inter and intragroup communication.

Figure 2.2: Many-to-many (group-to-group) communication.
the form of many-to-many communication as shown in Figure 2.2. Sending messages from a
single process to a group is called one-to-many communication, and from a group to a single
process is called many-to-one communication. Usually many-to-many communication can be
decomposed into one-to-many and many-to-one communication [Cooper 85].

2.2 Structural Classification

Viewed as a collection of object managers, a group can be classified on the basis of the
homogeneity of the internal state of the objects maintained and operations supported by each
group member. An object manager can be characterized by:

- **Application level objects** — the set of objects maintained by the process. Their value
determines the internal state of this process.

- **Application level operations** — the set of operations that can be executed on the applica-
tion level objects. Other processes can modify the value of these objects only by invoking
these operations through this manager.

Operations on the objects are what define the services a process provides. Because services are
accessed through interprocess communication in message passing systems, operation executions
and process state transitions are stimulated by the events of message arrival.

A **selection rule** of a group \( G \) is a set of criteria for selecting accessible objects in \( G \) and is
determined by \( G \)'s application. For simplicity, we assume that an object is accessible in group
\( G \) if and only if the object satisfies \( G \)'s selection rule, and that a process is a member of \( G \) if
and only if the process maintains at least one object satisfying that rule. The services provided
by a group are implemented by the group members cooperatively and are accessed by clients
through intergroup communication. A group \( G \) can be characterized by:

- **Group objects** — the subset of objects maintained by each member process in \( G \), which
  satisfies \( G \)'s selection rule.

- **Group operations** — the set of operations on the above objects.
CHAPTER 2. CLASSIFICATIONS AND REQUIREMENTS

Depending on how each group member implements and maintains objects and operations, a group can be placed in one of four categories:

**Data and operation homogeneous** (DOH): Every member in a DOH group maintains a complete replica of the set of group objects and implements an identical set of operations on these objects. To guarantee consistent external behavior, a DOH group maintains consistency among replicas of the objects and requires every member to execute exactly the same sequence of operations. DOH groups are used mainly to increase service reliability and availability. Examples are groups in ISIS [Birman and Joseph 87a], troupes in Circus [Cooper 85] and fault-tolerant state machine [Schneider 90].

**Operation homogeneous only** (OHO): In an OHO group, the object space is partitioned among group members — each member maintains only a part of the global group state. Object space partitions may overlap. Also, every member supports an identical set of operations on its portion of the objects. However, when an operation is invoked on an object, only members with the relevant object need to perform the operation. OHO groups are mainly used to distribute the work load among group members. Each member may maintain the integrity of its own objects independently of the other members. The distributed name service discussed in Section 2.3.2 is an example of the OHO group [Ahamad et al 88].

**Data homogeneous only** (DHO): Members in a DHO group share a set of objects by sharing the same address space on a single machine, or in some other distributed manner (for example, data replication). Each member supports a set of operations on the same objects. These operations may or may not be identical to those of other members. Upon invocation of a group operation, members may act differently. For example, a coordinating member accepts an operation invocation from a client and accomplishes the task through internal cooperation with other group members. The role of coordinator need not always be played by the same member. Also, members in a DHO group must synchronize themselves to serialize concurrent updates to the objects. This requires an underlying mechanism similar to that for DOH group support. The usual purposes of a DHO group are to provide
group services cooperatively via a set of worker processes; and to simplify the design, implementation and interface of the service by masking member cooperation from external observation. Examples include teams in V [Cheriton 88b] and the primary-secondary replication scheme [Schroeder et al 84].

Heterogeneous (het): As far as the group application is concerned, the objects and operations each member implements and maintains could both be heterogeneous. There may or may not be cooperation among group members, and their internal states may be completely independent of one another. Rather than encapsulating interactions among members to provide a cooperative group service, heterogeneous groups facilitate system control and simplify the interactions between the client and server groups. Electronic mail distribution lists, computer conferencing, news groups and distributed process-control are applications of heterogeneous groups.

2.3 Behavior Classification and Requirements

One cannot appreciate the design requirements for operating system group support without understanding the quality of service requirements of the intended applications. This section reviews various group applications, highlighting the service requirements expected from the group system, and classifies process groups according to the external behavior expected by the applications.

According to their external behavior, distributed process groups can be classified into two major categories: deterministic and nondeterministic. The former groups are mainly used in replicating data and services to enhance reliability; the latter are mainly used in distributing data and workload among multiple servers to improve information availability and resource sharing. More complete definitions of the two categories appear later.

Basically, a deterministic group requires high reliability in group communication to maintain strong consistency among members. Such groups are "heavy weight" in the sense that they require the communication layer to maintain complete group membership information and
atomic, consistently ordered group interactions. In contrast, nondeterministic groups are “light weight” since they need only basic datagram multi-delivery transport support. Inconsistencies and unreliable group interactions are handled in an application-specific manner, resulting in more flexible and efficient — but more complex — application programming. Whether a group is deterministic or nondeterministic depends solely on its application, not on its structural characteristics.

2.3.1 Deterministic Groups

A group is deterministic if it requires that every member in the group receive and act on a request.\(^3\) In addition, the order of the requests being processed is the same at each of the recipients. This requires coordination and synchronization among the group members. In most deterministic groups, member processes are equivalent [Cooper 85]\(^4\) — upon receiving the same request in the same state, the same procedure is invoked, and every member transfers to the same new state and produces the same response and external effects.

We now look at some deterministic group applications in terms of their basic characteristics and communication requirements.

Replicated File Systems

In a fully replicated file system, all file servers constitute a group. Files are replicated at every file server to enhance file availability and reliability. Two common methods of supporting replicated file systems are peer-member and primary-secondary [Chang and Maxemchuk 84b]. In the peer-member scheme, all members in a file server group are identical and group communication and coordination occur between the client and all members. In the primary-secondary scheme, a primary member handles the communication interface between clients and the group, and group communication and coordination occur between the primary and all secondaries in-

\(^3\)The term deterministic is used to characterize the relationship among group members, and is less restrictive than that used by some authors to mean only one possible execution of a single procedure.

\(^4\)There are cases where members of a deterministic group are not equivalent. See Sections 2.3.1 and 2.3.1.
side the group. A fully replicated file server group is a deterministic DOH group if the peer-
member scheme is used, or a deterministic DHO group if the primary-secondary scheme is used.

Replication transparency is an important characteristic of these systems. Clients of a file
system usually prefer a single file image regardless of whether a file is implemented by a single
server or many servers. The file abstraction as seen by a client is called a logical file image
and operations on it are called logical operations. The physical file copies maintained by the
file servers are known as file replicas and operations on them are called physical operations.
Reliability requires that file replicas be kept consistent at all servers, so that files are always
available to clients as long as at least one file server is functioning. Availability requires that
users should be able to read the files with minimum latency, reading a consistent copy of a file
from the closest functioning server, for example.

Both reliability and availability imply that file replicas must be consistent. Thus, every
logical file update must be atomic; that is, either executed by all servers or by none, and a
client must be informed whether its update has completed. Furthermore, because different
sequences of the same set of update operations can result in different file states, all servers
must execute exactly the same sequence of operations with respect to each logical file. Two
logical operations \( \text{op}_1 \) and \( \text{op}_2 \) are in conflict if they are data dependent; since they manipulate
overlapping logical data, the execution order affects the results. Two conflicting operations
collide if they are executed concurrently.

Consider two clients \( C_1 \) and \( C_2 \) that issue a LockFile request independently to acquire a lock
on the logical file \( F \). If the two requests are not consistently ordered at all server group members,
some servers may allocate the physical locks on their physical replicas of \( F \) to \( C_1 \), others to \( C_2 \),
depending on which request is executed first: thus, a deadlock may occur. Clearly, colliding
operations must be ordered: the order can be arbitrary as long as it is consistent at all member
sites. This ordering of colliding operations is called absolute ordering [Birman and Joseph 87a].

File servers can be added or deleted dynamically. Clients expect the file server group
to coordinate internally to hide membership changes from external observation. This type of
internal coordination includes keeping consistent name binding between a group identifier and a
set of server process identifiers, and bringing the state of a new server up-to-date. Satisfying this requirement not only simplifies the group interface to clients, but also increases the flexibility of file system configuration. Since a host cannot always distinguish between network partitioning and host failure, file system level consistency requires that both clients and servers be notified when either failure occurs, and that some higher level consistency control algorithms be used to handle the failure.

Replicated Program Execution

Replicated program execution is another example of the deterministic DOH group. In some distributed applications, a major concern is the resiliency of computations. Programs can be decomposed into abstract data type modules, each having a set of internal states and a set of procedures manipulating the states. Modules are replicated at multiple sites [Cooper 85] and replicas of a module form a group. Group members may be different implementations of the same abstract data type written by different programmers (subject to the constraint of equivalent deterministic behavior). Thus, module reliability can be enhanced by both replication and multiversion programming. Multiversion programming tends to reduce program design faults (assuming fail-stop behavior); replicated module execution tends to make the module run time execution robust. A procedure call to a module can proceed as long as at least one group member is functioning. An application programmer would expect that the syntax and semantics of a replicated procedure call is the same as for a non-replicated one, making the replication transparent.

Consistency in replicated procedure calls requires that the group members be deterministic and equivalent and that every member execute the same sequence of calls. It is the responsibility of the application programmer to ensure that modules are deterministic and equivalent. The group system, on the other hand, must guarantee atomicity and ordering (defined in the previous section) to each procedure call, as well as replication transparency. For each replicated procedure call from a client group to a server group, each client group member makes

\footnote{Also denoted as a troupe in the original reference [Cooper 85].}
a one-to-many call to the server group, each server group member accepts many-to-one iden-
tical procedure invocations (resulting from one call per client group member) and makes a
one-to-many reply to the client group after execution. Each client group member then handles
many-to-one replies from the server group. These multiple calls and receptions are best han-
dled in the group communication layer so that client/server programmers need not be aware of
multiple entities in the caller/callee group.

An advantage of making replicated procedure calls at the module level is that the degree
of replication can be adjusted dynamically according to the functions of different modules,
thus optimizing system performance and reliability. In other words, more copies of critical
modules could be made to enhance their reliability and fewer replicas would have to be made
for less important modules. Also, dynamically changing group membership allows system auto-
reconfiguration to be transparent to applications as group members fail and recover at run time.
However, this dynamic group membership makes it difficult to bind a group name to a set of
modules.

A replicated program execution often happens within a single local area network; therefore,
network partition failure is unlikely. When any group member fails, it is expected to reconfigure
itself autonomously, making partial failures transparent to client groups.

**Distributed Industrial Process Control**

Distributed process control is an example of the deterministic HET group. Imagine a simple
distributed process control environment in which the temperature in a reaction container is to
be controlled. A sensor measures the current value of the temperature and a console displays
that value to human operators, records the history of the sensor signals and allows an operator
to set the control parameters. A set of controllers manages the flow of cooling fluid into the
container by opening or closing valves. The sensor can view the console and the controllers
as a group. When a control parameter diverges from a preset value, the sensor multicasts the
measured result to the group so that the console informs the operators and the controllers open
or close the valves to adjust the flow.
This group is obviously heterogeneous because each member, although driven by the same sequence of signal stimuli, maintains completely different objects and performs different operations. These components are grouped together simply for convenience of communication; they are identified by a single receiver ID in each group message. If the underlying network supports multicast, only one copy is transmitted for each sensor signal, saving network bandwidth and speeding the signal processing.

Although there is no application level consistency requirement, this group communication has reliability requirements. Unless the sensor fails, its signals must be reliably delivered to all active members in the same order as generated. Also, each signal from the sensor must be delivered by a predefined deadline or the signal will become obsolete. The sensor expects no reply from the group. Any individual member failure must be detected quickly and operators must be notified to repair the failed component. Before recovery of the failed component, however, the remaining group members are expected to continue fulfilling their duties even though the complete group service may not be available.

Email Distribution List and Computer Conferencing

Electronic mail distribution lists and computer conferencing are two other examples of the deterministic HET group. People registered in the same distribution list or conference constitute a group. Grouping is for convenience of communication; for each message, a sender prepares a single copy and performs one send operation. Generally, a sender does not know who participates in the distribution list, because registration is handled by an independent authority. The message is sent to the address of the list, which has the same syntactic format as other single-user addresses and which logically includes all participants. The same is true for computer conferencing except that, normally, conferences are closed groups in which only the participants can send messages to the conference. In contrast, a distribution list is open to anyone with access to the its address.

Assuming registration and network connections are set up properly, messages to a conference are expected to be delivered atomically. If any participant receives a group message, then
other members in the same group should also receive the same message within a bounded period. For each message delivered, however, the sender may receive zero or many replies from recipients, either in the form of private one-to-one correspondence or as follow-up discussions to all conference participants. In a follow-up message \( m \), the speaker makes his point on the basis of all the messages related to \( m \) that the speaker has received: these messages are called the context associated with \( m \). For other participants to understand \( m \) properly, they must be able to reference its context [Peterson 88, Peterson et al 89]. A recipient should not see a message without having also received its context. This dependence relationship is sometimes called a causal relationship [Birman and Joseph 87a] and defines a partial ordering among messages submitted to a conference.

In both distribution lists and conferencing, absolute ordering is not required. Concurrent messages usually can be delivered in arbitrary order because they are not context related and because the participants' states are not message dependent. Generally in a conference, a single person is elected as the conference chair who determines when to start and close the conference and how to arbitrate the order of concurrent messages when necessary. Sending and receiving messages can be concurrent and asynchronous for each participant. Because incoming messages may affect the content of out-going ones, receiving may be given higher priority than transmission.

In conferencing, a follow-up message may become a competitive orphan (explained in Section 2.3.2) because of the asynchronous communication pattern. In this situation, several people may respond identically to the same message before seeing responses from each other. Fast delivery makes this problem less likely but does not eliminate it entirely.

Membership changes should not affect the group communication. A new member to a group normally becomes up-to-date by reading the conference bulletin board\(^5\) or the discussion archive. The failure of a member is normally treated as if it has left the group.

\(^5\) A bulletin board may be necessary as a conference proceeding so that speeches delivered to a conference can be read later. Even with distribution lists, a message archive for group discussions is very useful.
Distributed Databases

A deterministic HET group may be used within a distributed database. A distributed database model has a transaction manager (TM) and a data manager (DM) at each site. Each TM accepts user requests and translates them into commands for DMs. Each DM maintains part of the database stored at its site and may concurrently execute transactions from multiple TMs. A transaction group consists of all DMs participating in the same transaction. This group is heterogeneous because each DM maintains a different part of the database and each may respond differently (accept or reject the pre-commit from the TM) according to its local status. Messages from the coordinating TM are atomically delivered to the transaction group.

For the purposes of concurrency control and failure recovery, a two-phase commit (2PC) protocol is normally used at the end of each transaction. When the coordinating TM and all cohorts (DMs) that know the decision (commit or abort) fail, the conventional 2PC will be blocked until some processes recover.

In [Chang and Maxemchuk 84a], a simple nonblocking 2PC is designed on the basis of ordered atomic group communication. In their nonblocking 2PC, the protocol proceeds as the conventional 2PC in the absence of a coordinating TM failure. When the coordinating TM fails, a communication-layer-generated failure notification is broadcast to the group and each DM aborts the transaction upon receiving this notice. This protocol requires not only an atomic but also an ordered group communication mechanism to guarantee:

1. that every DM in a transaction group will receive exactly the same sequence of instructions from the coordinating TM; and

2. that if the coordinating TM fails after broadcasting commit, its failure notification is delivered either before or after its commit message consistently and atomically to all DMs, allowing them to either commit or abort the transaction consistently.

Although the membership of a transaction group is governed by the coordinating TM and a DM failure is detected by the TM during the execution of 2PC, the communication layer still needs
this membership information for ordered atomic communication to support the nonblocking algorithm. Inconsistent replies from DMs are handled by the coordinating TM itself rather than by the group mechanism.

**Summary of Requirements**

The above analysis of deterministic group applications makes clear that grouping transparency is a desirable property allowing a client to treat a service of a group as if it were provided by a single server. A single call is made and a single result, if any, is expected from the server group. Also, a server maintaining group object replicas need not know that another co-server exists, and thus application programmers need not be aware of group coordination.

Grouping transparency for deterministic groups must satisfy the following requirements:

**Communication transparency.** Communication transparency consists of two aspects — atomicity and ordering [Birman and Joseph 87a].

- **Atomic message delivery.** An atomic group message is either received and processed exactly once by all members in the recipient group or by none at all. Atomicity hides a partial group communication failure by converting it into a total failure.

- **Application level absolute ordering.** The delivery order of messages to group members needs to be synchronized if the order will affect the result. Every member of a group must see the same sequence of requests on dependent data and adjust its internal state accordingly. Also, in the case of colliding requests, the common members of two overlapping groups must see a consistent “combined” sequence of requests to both groups.

---

7It is worth noting that some systems introduce another type of group message ordering which respects the causal relationship among messages. It is our belief that maintaining causal relationship is a general requirement in process communication, even in the case of one-to-one communication. Once absolute ordering is supported, a group can be treated as a single entity and causally ordered group communication can be built on top of absolute ordering using any causal synchronization mechanism designed for one-to-one interprocess communication.
Reply handling transparency. Because client and server interactions normally follow the request-response pattern [Cheriton 86b], reliable multicast from client to server is not enough. Replies from a group also have to be collected and processed properly to achieve grouping transparency [Birman and Joseph 87a, Cooper 85]. For each group request there exists a potential for multiple responses from a server group. These responses may or may not be identical. Reply-handling transparency guarantees that a client need not be aware of multiple replies to its request. It sees a single reply result without having to be concerned with how this result is derived from multiple replies (for example, by weighted voting).

Multiple timely replies may cause congestion (in a short period) at the original message sender if the speed of reply processing at the latter cannot catch up with that of the reply arrivals. Some reply messages may be lost due to message receiving buffer over run. For a system to be useable, a mechanism for coordinating and reliably delivering multiple timely replies will be required.

Naming transparency. This involves the problem of dynamically and transparently binding group members to a single name. Group naming consists of two parts: mapping a logical service name into a group of servers and allowing group membership to change dynamically. A group-view is defined as a snapshot of the group membership at a particular instant of time. It is maintained by each of the concerned parties, be they the group members, the system name servers, or any client needing to make decisions on the basis of the group-view. A group-view changes in parallel with other group message activities as members fail and recover, or are inserted and deleted.

Since atomic group interactions rely on a consistent group-view to verify that all active members have confirmed reception of each atomic message, group-view changes have to be detected in a consistent manner. It is convenient to serialize group-view changes consistently with respect to other group message activities. ISIS group members achieve this
by having a system generated announcement, issued on behalf of the failed member,\(^8\) follow all its pre-failure messages. This announcement arrives at every member in the same order with respect to the other group messages, so that the members will all see the same sequence of group-view transitions at virtually the same time. Therefore, it is guaranteed that no message will arrive from a failed member once its failure is announced [Birman and Joseph 87a]. If a member recovers, the state of the new member must be brought up-to-date with a consistent snapshot of the group internal state, and all concerned parties should perceive the existence of the new member before any message is received from it.

**Failure transparency.** Depending on the purpose of a group, either the clients and server group members are notified of the failure to take application level recovery actions, or a member failure is hidden from the clients. In the latter case, the failure may be presented as a complete group failure, or other group members may take over the role of the failed member. The technique chosen depends on the function of the group. When a deterministic group is used to enhance data reliability, as in replicated file systems or databases, strong consistency is required among the group object replicas. Therefore, the group consistency control strategy may exaggerate a partial failure as a total failure;\(^9\) if any member fails or the network is partitioned during a group operation execution, the operation cannot succeed until the failure is repaired or the group is properly reconfigured. When a group is used to enhance service reliability, however, as in replicated procedure call, maintaining service to the clients is important. During recovery of the failed member, the remaining active group members should continue to fulfill their duties to keep damage to a minimum.

**Real-time requirement.** We can measure multicast message delay in terms of *distribution time* — the time taken for all operational members in a group to receive a multicast, or

---

\(^{8}\)In this context, failure also includes deletion, and recovery also includes insertion.

\(^{9}\)A similar idea has been used in atomic message delivery.
in terms of *completion time* — the time taken for the sender to learn that all destinations have received its message reliably [Mockapetris 83]. In deterministic group applications, consistency is more important than efficiency. When trade-offs between the two have to be made, system support often gives priority to consistency. Usually in real-time systems, on the other hand, messages must be delivered within a client-specified deadline; otherwise, a message is deemed obsolete and a timing fault is triggered.

In one-to-one interprocess communication, a server can simply ignore a timing fault message or respond to the client with an operation failure. In group communication, it is possible for some servers to receive the message on time while others see a timing fault, even though atomicity and ordering are guaranteed. When this occurs, servers in the recipient group must coordinate to act consistently on each timing fault message to guarantee consistency. Multicast distribution time should be bounded so that group actions can be scheduled to occur atomically and simultaneously in virtual time at all group members [Cristian et al 85].

### 2.3.2 Nondeterministic Groups

Deterministic groups require strong data and behavior consistency, and synchronization among all members. Nondeterministic groups assume that their applications do not need such strong built-in consistency and they relax it in various application-specific ways. Nondeterministic group members generally need not be equivalent [Cooper 85]. Each member may respond differently to a group request, or not respond at all, depending on the state and function of each individual member. Normally, either member states of a nondeterministic group are unaffected by processing user requests or they are not necessarily consistent. Either requests to such a group do not require all members of the group to act or missing requests can be detected and recovery completed within the application. Because of the relaxed consistency, maintenance overhead for the group is generally lower than that for a deterministic group. Whether a group should be implemented as deterministic or not depends on the requirements of the application.

Next, we will describe some nondeterministic group applications in terms of their basic
characteristics and communication requirements.

**Distributed Clock Service**

The time-of-day service in the V system [Cheriton 88b] is an example of a nondeterministic \( \text{DOH} \) group, in which all group members implement the same set of functions and play the same role in the service. Although all members are supposed to maintain identical objects, the state of these objects may differ when a partial group communication failure results in some members not receiving a group request.

In the distributed time-of-day service, every station periodically receives a clock tick from a central clock. Between clock ticks, each clock replica simply caches the latest time stamp from the central clock and extrapolates forward using its local clock. Clock drifting can be corrected by the next clock tick. Time requests are handled using the locally extrapolated time values. Should any clock update message be missed, the next clock update corrects it. Although the time value stored in the local clock may not be absolutely correct, it is accurate enough for most non-time-critical applications.

Similarly, many applications that replicate data do not require absolute consistency. The required level of consistency is obtained by using application semantic knowledge and assuming that the client can detect, recover from, or tolerate inconsistencies. This reduces communication complexity and promotes efficiency. The nondeterminism in these applications stems from the fact that group members may maintain an inconsistent or inaccurate global group state. Applications of this type need only an efficient and "best effort" multicast mechanism.

**Distributed Name Service**

In a distributed name service, a set of name servers operate at several machines in the network. In some designs the global name space is partitioned and a different name server maintains each partition [Ahamad et al 88]. This type of distributed name server forms a nondeterministic \( \text{OHO} \) group because every member performs the same set of operations. As an example, each object in the operating system Clouds is assigned a logical system name [Ahamad et al 88]. A
mapping function $\omega$ maps a system name onto a multicast address $A = \omega(S)$. Each station maintains a multicast address table for the objects stored at the node. To locate an object $O$, a user provides its system name $S$. The client host first calculates $O$'s multicast address $A = \omega(S)$ and uses $A$ as an index to search its local object directory. If it is not found, a multicast remote procedure call is invoked on the address $A$. Nodes which have $A$ in their multicast address table accept the remote procedure call, search the local object directory for the object with the system name $S$, and reply if they found it. In this way, all nodes share the overhead of maintaining, locating, and migrating objects.

Grouping transparency is an important property expected by both clients and servers. A client locates an object by submitting a name LookUp request to the name service. It is irrelevant to the client which server responds or whether a single or multiple servers handle the request. Each server manages its assigned portion of the name space and responds only to requests for objects it knows. It need not be aware that other servers exist.

Nondeterminism in distributed object naming stems from the fact that the global group state is partitioned among members; therefore, a client does not know which server will perform its request. A LookUp need not be multicast to name servers atomically, because it is an idempotent operation \(^{10}\) and does not alter the name servers' state. Note, however, that the name binding update in the name servers is a deterministic operation, and all replicated servers must execute the update operations atomically and in the same order that names are bound.

Grapevine [Schroeder et al 84] adopted another way of supporting a distributed name service. It exemplifies the nondeterministic DHO group in which the name space is fully replicated at each name server. A server may play primary or secondary roles in name updates, however, even though every server supports the same set of operations. The Grapevine update algorithm does not guarantee a consistent view at all name replicas because name update broadcasts are neither atomic nor ordered. A client deals only with its local name server without seeing the whole name server group. Clients can detect and correct inconsistent and stale names in an

\(^{10}\) An operation is idempotent if it can be executed zero or multiple times without changing the state of the server.
application-specific manner.

Both examples should make clear that failure of any name server does not stop the activities at other servers.

**Contract Bidding**

Contract bidding, another example of a nondeterministic OHO group, is a technique for resource sharing and load balancing among stations in a server pool. Clients submit job requests for servers to complete. The specific server station and the order of execution do not matter. Server group members do not maintain a global group state. All worker stations are functionally identical and respond to idempotent service contract bids only on the basis of their local state. Upon completing a task, servers return to the ready state for the next job assignment.

The number of available server stations is always changing, as is their current load. To optimize overall system throughput and mean response time, tasks should be scheduled to keep all servers equally busy. Scheduling is usually done in one of two ways. In a client-initiated scheme, a client posts its task requirement for server stations in the pool to bid on. An available server responds with its current load condition and the client then chooses the proper server to complete the task. In a server-initiated scheme, potential clients form a group and an available server posts its request for loading to the client group. Each client responds with its task requests and the server chooses the appropriate one to execute.

In both schemes a *competitive orphan* may be generated. In contrast to a *failure orphan*, generated when a server continues to execute the request of a dead client, a competitive orphan is generated when server group members are not properly coordinated.$^{11}$

In the server-initiated scheme, a competitive orphan request could be generated from a client that does not know whether its job request has already been carried out. Also, in the client-initiated scheme, multicasting a job request could trigger multiple concurrent executions

---

$^{11}$An orphan is a request that its sender either has dead or lost interest. Failure orphans are a general problem in the client-server model; competitive orphans are a special problem in the group communication context. We can view a failure orphan as resulting from a lack of coordination between the client and the server group; we can view a competitive orphan as resulting from a lack of coordination among group members.
at different servers, even when only a single execution is needed. The effect of competitive orphans must be reduced to a minimum and the competition losers must be notified so that they can participate in subsequent contract-bidding activities. Again, the communication overhead between clients and the members of a server group should be minimized; multicast should be efficient but need not be perfectly reliable. The failure of any one server should not terminate the whole system. The client whose job was being executed by the failed server must be notified to re-submit its request, however.

News Propagation

News propagation in USENET exemplifies the nondeterministic HET group. People subscribing to the same news group (a particular topic of discussion) constitute a group, and the state and the operations each human subscriber performs differ. A news poster never knows who the members in a group are. News propagations need be neither atomic nor ordered. Respondents to a news article can use either one-to-one personal correspondence or follow-up articles to the same or a different group. Each person then decides what to do with each news article and the follow-up comments. A subscriber can come and go at will; no synchronization is necessary.

Summary of Requirements

Nondeterministic groups are intended to improve service performance. Through grouping transparency, a group service strives for the same simple syntax and semantics found in one-to-one interprocess communication. Depending on the intended applications and the nature of the particular group, however, this transparency may be relaxed. Nondeterministic groups induce less overhead and generally have the following characteristics and requirements:

Communication transparency. The dominant communication pattern in nondeterministic group applications is request-response. Usually, applications do not require absolutely reliable message delivery or message ordering. Interactions with nondeterministic groups are inherently asynchronous because: (1) it is neither necessary nor realistic to expect
a client to wait until all server group members are synchronized and ready to receive a request; (2) server group membership normally is not known to clients; and (3) a server may not receive the request at all. Data consistency is not a problem for various reasons: the application requests may be idempotent, group consistency may not be critical to the application, or the application is able to detect and recover from inconsistency easily. Application programmers are given the flexibility — with the attendant complexity — of handling partial message failures.

Reply handling transparency. Multiple replies from a nondeterministic group may not be consistent. They may have to be handled by the clients in an application-specific manner rather than by the server group, as in deterministic groups; this sacrifices a certain degree of reply handling transparency. Also, an application must provide its own timeout value for each multicast request because the communication layer itself does not know how long to wait for the server group responses. When the timer expires, the application may decide whether to re-multicast or to perform alternative actions. In deterministic groups, these actions are normally performed at the communication layer.

Naming transparency. As in the case of deterministic groups, applications prefer to use a logical name to address a group service rather than having to know each individual member. To achieve naming transparency, both clients and servers prefer to handle requests and replies independently of the number of servers. The membership of a nondeterministic group can change dynamically, however, and is usually known neither to the active group members nor to the communication layer.

Failure transparency. Nondeterministic group member failure has different semantics from that of deterministic groups. When a member fails, other active members may transparently take over the uncompleted task to enhance availability, share service load and reduce global communication traffic.

Competitive Orphan Problem. In a nondeterministic group, a new type of orphan, the
competitive orphan. can be generated because of a lack of internal coordination among group members. Except for computer conferencing, most deterministic groups do not experience this problem because a group request must be handled by every member synchronously while the client waits for a reply from every server.

Table 2.1 summaries the differences between the deterministic and nondeterministic categories.

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Nondeterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naming</td>
<td>Complete group-view at communication layer is necessary. Membership changes need be synchronized with all other group messages.</td>
<td>Group-view usually is not known to anyone, even to the communication layer.</td>
</tr>
<tr>
<td>Reply handling</td>
<td>If required, expect all members to reply. Group members handle inconsistent replies without the involvement of the client.</td>
<td>Clients must handle inconsistent replies explicitly and applications have to provide the timeout parameter.</td>
</tr>
<tr>
<td>Failure handling</td>
<td>To enhance reliability, a partial failure is turned into a total failure in most cases.</td>
<td>To enhance availability, partial failures are usually hidden by active group members.</td>
</tr>
<tr>
<td>Others</td>
<td>In real-time systems, a timing fault may cause inconsistency.</td>
<td>Competitive orphan may arise due to a lack of proper coordination among group members.</td>
</tr>
</tbody>
</table>

Table 2.1: Deterministic vs. nondeterministic.

It is worth noting that although we classify groups into deterministic and nondeterministic categories, there is a grey area between the two in which many applications may fit in. An application of groups may have to be deterministic in some aspects, but need not be that way in other aspects. The purpose of this classification is not to draw a line between the two categories, but rather to provide guide-lines for understanding different aspects of group communication and transparency requirements from applications.
2.4 Discussion

The two classifications of groups previously discussed are based on different criteria, one on structure (data and operation homogeneous, operation homogeneous only, data homogeneous only and heterogeneous), the other on behavior (deterministic and nondeterministic). By orthogonally projecting one over the other, as Table 2.2 shows, we hope to better understand how various group applications fit the classifications.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Deterministic</th>
<th>Nondeterministic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data and operation homogeneous (DOH)</td>
<td>Fully replicated file systems (peer member scheme), replicated procedure call &amp; fault tolerant state machine</td>
<td>Applications such as the distributed time-of-day service in V system.</td>
</tr>
<tr>
<td>Operation homogeneous only (OHO)</td>
<td>Partially replicated file systems.</td>
<td>Clouds' distributed name-server group &amp; contract bidding.</td>
</tr>
<tr>
<td>Data homogeneous only (DHO)</td>
<td>Fully replicated file systems (primary and secondary scheme).</td>
<td>Grapevine's name-server group.</td>
</tr>
<tr>
<td>Heterogeneous (HET)</td>
<td>Distributed process-control, computer-conferencing &amp; E-mail distribution list.</td>
<td>News propagation in a news-group of USENET.</td>
</tr>
</tbody>
</table>

Table 2.2: Sample applications based on the classifications.

First, we can see that external uncertainties in most nondeterministic groups stem from the following facts:

- objects are distributed only to a subset of group members and the size and membership of this subset may not be known in advance; and
- even when objects are fully distributed at all group members, applications do not require their values to be always consistent or accurate and missing group messages can be tolerated.
Second, Table 2.2 shows that most applications of deterministic group require that messages are sent atomically and in order, regardless of the homogeneity of group structure. For instance, the communication support for a deterministic DHO group is the same as for a deterministic DOH group. No matter how differently each individual DHO group member functions at a high-level, to guarantee consistency among replicas, changes to the objects must be propagated atomically and in order.

Consider partially replicated file systems as an example of the deterministic OHO group in Table 2.2. File system reliability requires that for each update request, those and only those servers having the target file take action and respond. It is difficult to map a logical file onto an unknown number of file servers maintaining the physical replicas of the file. Thus, there exists a level of inherent nondeterminism in the group communication. If we had a separate server group for each replicated logical file object, we would end up with a fully replicated deterministic DHO server group for each file. This may not be necessary, however, and having many dynamically changing groups could be expensive.

It would be preferable to install software filters at the client and the servers to eliminate this structural nondeterminism. Each server filter would discard requests for nonlocal files. The client filter uses some mechanism (perhaps by consulting a name server) to determine the membership of the implied subgroup (those having a copy of the target file) and to guarantee atomic message transactions only with this subgroup. Once the implied subgroup membership is determined, the group transaction can proceed atomically.

An alternative would be to have every file server reply to every request; those not knowing the target file would simply reply with a "null" message. The client would work on non-null replies using the knowledge of the whole file server group membership to eliminate the nondeterminism. This scheme trades extra host loading cost for structural determinism to gain reliability. It differs from broadcast in that:

1. only file servers pay host loading costs for each file access, and

2. being a system program, file servers are generally more trustworthy and therefore file
transactions can be made secure.

Third, Table 2.2 shows that a DOH group can be used for both deterministic and nondeterministic applications. The same is true for the other three group structures. This suggests that it is the application, rather than the internal structure of groups, that dominate the requirements to group communication support.

Performance of group management, although important to certain applications, is one aspect that has not received enough attention. To support deterministic groups, an IGC protocol, besides delivering messages to members in the destination groups, must also guarantee ordered and atomic message transactions among the members in each receiving group. In this way, application level programming can be simplified. This extra effort is not necessary in an IGC protocol supporting nondeterministic groups only. This is because either the applications do not care about message reliability or ordering, or the applications will take care of message reliability and coordinate member actions themselves. When designing a group communication mechanism, trade-offs can be made based on the requirements of the intended applications, the message synchronization overhead in the IGC protocol, and the overhead of handling reliability and ordering issues at the application level.

There is yet another possible dimension of classification, based on the way group members cooperate internally to provide the expected behavior. For example in a deterministic DOH group, the single copy behavior may be guaranteed by using two-phase commit protocol, majority voting, weighted voting or primary-secondary scheme. Discussions along that dimension are beyond the scope of this chapter, however.

2.5 Related Work

Several existing systems support group communication. Isis [Birman and Joseph 87a] and Circus [Cooper 85] systems are primarily intended to deterministic replicated data objects or procedure module groups. [Birman and Joseph 87a] provides detailed design and analysis on protocols for atomic and ordered group communication. The V system [Cheriton 88b,
Cheriton and Zwaenepoel [85] and several other experimental systems [Hughes 88b] are examples of systems supporting nondeterministic groups.

Other work in understanding multicast includes [Mockapetris 83], which presents a general analysis of multicast mechanisms at the network level, rather than the application level. A multicast taxonomy is given in [Hughes 89a] on the basis of the number of replies to each multicast request. There is no general examination and classification of group communication requirements from the viewpoint of applications, however.

2.6 Chapter Summary

Before designing a general, coherent and integrated group communication system, we must understand how it will be used; that is, the basic application requirements. We have analyzed different types of groups, along with their potential applications and classified group applications into two major categories: deterministic and nondeterministic. One important distinction between the two is whether or not the group communication software needs to know the membership of groups to coordinate the actions of members. Orthogonally, according to the structure, process groups can also be classified as data operation homogeneous, operation homogeneous only, data homogeneous only or heterogeneous.

A basic conclusion drawn from this analysis is that grouping transparency is important and desirable. When integrated into the underlying group support, it simplifies the interface between the server groups and their clients by hiding from the clients, as much as possible, the membership of server groups and the interactions among group members. This enables the designers of clients and servers to concentrate on the problems to be solved, as they do in the unicast environment, without concern for coordinating multiple servers. Grouping transparency is manifested in group communication, group naming, multiple-reply handling, group-view change and partial failure. We hope this classification framework and analysis will enhance the understanding of process groups, group communication and some applications, thus aiding designers working with these mechanisms.
Chapter 3
Internet Group Name Resolutions

Groups exist naturally in the internet. Teleconferencing in wide area networks (WAN) and electronic mail distribution lists are two example applications. The second part of this thesis, starting from this chapter, will be focused on an important aspect of group communication: group naming support in the internet environment. The naming mechanism discussed in the rest of this thesis is intended to support deterministic groups. The communication support requirements for deterministic groups are stronger than those for nondeterministic groups. Therefore, more applications can be supported by deterministic group communication.

In this chapter, we investigate issues in name resolution for internet groups. In Section 3.1, the internet group naming structure is extended from the conventional flat structure to a nested structure to support efficient internet group communication and subdomain autonomy. In Section 3.2, a graph theoretic model of group naming is introduced. Following this is a brief overview of different name resolution schemes and the formulation of the resolution loop problem and the resolution duplication problem. A discussion of the correctness criteria that algorithms handling these problems must meet is included in this section as well. In Section 3.3, a brief taxonomy of various existing approaches to handling resolution loops and duplications is discussed and the shortcomings of these existing methods are analyzed. Related work in the area is mentioned in Section 3.4.

1A preliminary version of Chapter 3 and Chapter 4 is published [Liang et al 90b].
3.1 Nested Group Model

An internet consists of a set of subnets connected by internet links. An internet group is a group whose members are located in more than one subnet. Internet group communication refers to sending messages to internet groups.

A subnet may be any type of local area network, such as Ethernet [Metcalf and Boggs 76] or Token ring [Bux et al 82]. Although LANs generally provide efficient message transmission, they may not directly support multicast. In the remainder of this thesis, we do not assume multicast support in subnet.

Internet links usually do not directly support multicast [Deering 90]. Compared to LANs, communication across internet links is usually more expensive because of the low bandwidth and long delay in internet links. It is desirable for internet group communication to generate as little internet link traffic as possible. It is also desirable for subnets (or administrative domains) to be autonomous in the sense that changes made within a subnet (domain) have minimum impact on other subnets (domains). For these reasons, we introduce the nested group model to internet groups.

An nested group is a group whose members are either processes or subgroups. The process members usually are located in the same subnet. The subgroup members contain either processes or other subgroups. This nested structure need not be restricted to a single level. A group can have more than one subgroup and as many levels of nesting as necessary. When a group $x$ contains another group $y$, $x$ is a parent (or supergroup) of $y$ and $y$ is a child (or subgroup) of $x$. A group can have more than one parent groups. Each group is identified by a unique gid and appears in its parent group as a single member.

The nested group model minimizes the group communication traffic on internet links as well as the message delivery time to internet groups. Instead of one copy of each message per member process, one copy per subnet is sent across internet links. The message delivery time can be further reduced by allowing a subnet to use whatever multicast support available in that

\[\text{2There is no functional difference between a group, a subgroup and a supergroup. The terms subgroup and supergroup are used when we need to emphases the parent-children relationship between groups.}\]
 subnet to distribute messages to local members.

The nested group model supports subnet (subdomain) autonomy. By grouping members of the same subnet (subdomain) as a subgroup, only the identifier of the destination subgroup is required to send messages. Furthermore, membership changes in a subgroup can be sheltered within a subnet.

In summary, the model of nested group is useful to internet groups not only because it simplifies the user interface (by addressing each group using a single identifier) and improves system modularity (a nested group hides the internal details of a child group from its parents), but is also necessary as the membership of a group in a subnet may not be known to outsiders.

3.2 Group Naming Model

Addressing a group by a logical group identifier (gid) (as opposed to a list of individual member process identifiers (pid)) allows the group to be managed/accessed via its gid without referring to its membership. Since it is the individual member processes that eventually accept and process a message, the membership must be somehow maintained either explicitly or implicitly in a centralized or distributed manner [Birman and Joseph 87a, Chang and Maxemchuk 84b, Cheriton and Zwaenepoel 85]. At some point during the group communication process, the destination gid must be expanded into the set of member pids to deliver the message. This expansion process is called group name resolution.

For example in Isis, group membership is consistently replicated at every group member and cached by message senders [Birman and Joseph 87a]. A gid is resolved by the communication software at the sending machine to a list of receivers using a local membership cache. The list of receiver pids is then used in transmitting the message to the group members. Another example is the V system in which name resolution is performed at the destination machines [Cheriton and Zwaenepoel 85]. From a broadcast channel, a machine receiving a message with a recognizable gid expands the gid into a list of pids of the member processes on that machine and delivers a copy of the message to everyone in the list.
3.2.1 Group Name Graph

To better understand group naming problems in the internet, we develop the following name graph model. A grammar model for group naming is included in Appendix A.

Nested groups in a distributed system can be characterized by a simple directed graph called a name graph. Each node in the name graph is uniquely labeled by either a gid or a pid. A node labeled by a gid is a group node representing a group in the system; a node labeled by a pid is a process node representing a single process.

An arc < x, y > in a name graph goes from a parent node x to an immediate child node y if y is a member of x. A child can be either a group or a process node. Process nodes are leaf nodes with out-degree zero because they do not have children; group nodes are internal nodes with one or more children (assuming groups are not empty). A node may have multiple parents. In that case, it has multiple in-arcs, one from each parent. A group that is not a subgroup of any other group is a top-level group, and its corresponding node in the name graph is called a root node. Given a name graph, a partial ordering from ancestors to descendants is defined by the transitive closure of the relationship given by the arcs.

To ensure that the simple directed name graph model sufficiently represents the naming structure of nested groups in the real world, we assume that:

1. a group never includes itself directly as a subgroup (i.e., no self-loops), and
2. a group does not include the same subgroup more than once (i.e., no multiple edges).

Obviously, these two assumptions do not limit the way that groups may be structured.

Intuitively, in a name graph the member processes of a group x are the process nodes reachable from node x. A node y is reachable from a node x, denoted as x \rightarrow y, if there is a directed path from x to y in the name graph. The name graph of a group x is a subgraph of the global name graph. It is rooted at node x and consists of all nodes and arcs reachable from x.

\[\text{simple graph is a graph with no multiple edges or self-loops.}\]

\[\text{It is important not to confuse arcs in a name graph with physical links of the network. The former correspond to communication paths that may consist of several physical links.}\]
CHAPTER 3. INTERNET GROUP NAME RESOLUTIONS

Corresponding to the group management operations in real systems, the following operations can be defined on a name graph and executed at group nodes:

- **create(x)** — creates a node \( x \) in the name graph when a new group \( x \) is set-up;
- **delete(x)** — deletes node \( x \) from the name graph when group \( x \) is dismissed;
- **join(x, y)** — adds arc \( < x, y > \) into the name graph when a group/process \( y \) becomes a new member of group \( x \) (consequently, all members of group \( y \) become descendants of \( x \) when the operation completes);
- **leave(x, y)** — deletes arc \( < x, y > \) from the name graph when a group/receiver \( y \) relinquishes its membership in a group \( x \); and
- **name_resolution(x)** — obtains the set of identifiers of the member processes in group \( x \).

### 3.2.2 Group Name Resolution

Each arc in a name graph represents one level of indirection in group naming. Given a name graph, the *name resolution procedure* executed at a node \( x \) maps \( gid_x \) onto the set of identifiers of all immediate children of \( x \). The *name resolution process* for a group \( x \) can be viewed as recursively traversing the subgraph rooted at \( x \) along all out-going arcs (possibly in parallel) and executing the name resolution procedure at each reachable group node hop-by-hop until all process nodes are reached. The result of name resolution for a group \( x \) is defined as

\[
NR(x) = \{ pid | pid \text{ of the process reachable from } x \text{ in the name graph} \}.
\]

Paths traversed in resolving group \( x \) are called *name resolution paths* of \( x \).

To *send* a message to a group \( x \) is to deliver the message to every member in \( x \). Therefore, as part of the *send* operation, the name resolution process for \( x \) is invoked to obtain \( x \)'s membership. The basic requirement that the *send* operation imposes on the name resolution mechanism is:
**B1:** The result of a name resolution process for group $x$ is correct if it satisfies the condition that $NR(y) \subseteq NR(x)$ if and only if $x \rightarrow y$ in the name graph.

Assuming the absence of communication failures, B1 guarantees that each message to a group $x$ is delivered to every member of $x$.

There are two approaches to doing name resolution: *centralized source name resolution* and *distributed name resolution*, depending on how and where name resolution is performed. In the centralized source name resolution, the destination gid is completely expanded into a set of receiver pids (called the *distribution list*) at a single location (either at the name resolution server or at the sender itself) before the group message is transmitted. Groups in Isis and Circus are examples of this approach [Birman and Joseph 87a, Cooper 85]. Distributed name resolution does not fully expand a gid at a single site before message transmission. Membership of a group is distributed across the network, and the group membership is only known to the name resolution server local to that group. A group is identified by a single id outside the subnet and is included as a single member in its parent groups. Group name resolution occurs incrementally as the message propagates to group members. A gid is expanded only when the message is going to the next hop. [Cheriton and Deering 85, Cheriton and Zwaenepoel 85] and [Frank et al 85] provide examples of this approach.

### 3.2.3 Resolution Loops

A *cycle* is a single path loop in a name graph. Two cycles are *chained together* if there is a path from some node in one cycle to some node in the other and visa versa. In a name graph, a *resolution loop* (RL) contains all cycles chained together. An RL is a *strongly connected component* in terms of graph theory.

Two groups are *deep equal* if their name resolution results are identical. Deep equal groups share exactly the same set of member processes although their name resolution paths might be different. Nodes that are deep equal because they are in the same RL (hence, are mutually reachable through the loop) have the relationship of *loop deep equal*. We shall denote this relationship by $\leftrightarrow$. 
Lemma 3.2.1 Loop deep equal is an equivalence relationship; it is reflexive, symmetric and transitive.

Proof:

This lemma follows from the fact that all loop deep equal nodes are residing in the same strongly connected component in a name graph [Even 79].

The name resolution process for a group \( x \) is non-terminating if and only if an RL exists in the name graph of \( x \). In this case, name resolution requests circulate around the RL and saturate the network. It is absolutely necessary to control RLs for group communication to function correctly. Besides requirement B1, a name resolution mechanism capable of handling RLs must:

**R1**: terminate each name resolution process in a finite number of steps, and

**R2**: preserve the loop deep equal relations defined by RLs.

Requirement R2 ensures that users are given the flexibility to take advantage of loop deep equal relation of RLs if they want.

As far as group name resolution is concerned, name expansion is not performed on a pid and no RL contains a process node. Therefore, we can ignore process nodes and the arcs leading to them to simplify the name graph. Such a simplified graph is called a reduced name graph. In the following sections, the term “name graph” means a reduced name graph.

### 3.2.4 Duplication Loops

In a name graph, a set of paths from a node to another node is called a *segment* if the last hop in these paths are the same. Two parallel segments from a node to another node are *distinct* if their last hop are different. A duplication loop \( DL(s, k) \) is a subgraph identified by two nodes — a *source* \( s \) and a *sink* \( k \). It consists of all the segments from \( s \) to \( k \) and satisfies the following conditions:

- the total number of distinct segments from \( s \) to \( k \) is greater than one, and
• at least two immediate children of \( s \) in these segments are different.

Note that a segment may have some RLs and DLs embedded in it. An RL/DL is *embedded* in a segment from \( s \) to \( k \) if all the nodes in that RL/DL are reachable from \( s \) and if \( k \) is reachable from all these nodes.

The existence of a \( DL(s, k) \) is a necessary and sufficient condition for resolution duplication to occur,\(^5\) because if no duplication suppression is enforced, a message to or routed through \( s \) will reach \( k \) through more than one \( s \rightarrow k \) segment, resulting in resolution duplications at \( k \) as well as at the descendants of \( k \). On the other hand, assuming RLs are properly handled and no DL exists, there will be at most one path between any pair of nodes in a name graph and each message can be resolved at a node at most once.

Although DLs are the natural results of subgroup sharing, duplication suppression improves the efficiency by preventing a message from being transmitted more than once in the subgraph rooted at the sink. Besides requirements B1, R1 and R2, other requirements for a name resolution mechanism capable of suppressing name resolution duplications are:

\[ D1: \] a message sent to or routed through the source of a DL should be resolved at most once at every node in the duplication loop (including the sink), and

\[ D2: \] a message sender should not be prevented from retransmitting its messages.

Requirement D2 comes from the observation that retransmission is one of the methods to guarantee *at least once* semantics in many protocols. Each retransmitted message should be delivered to all the members of the destination group independent of previous transmissions.

### 3.3 A Taxonomy on RL and DL Control

In this section, we survey existing methods of handling resolution loops and duplications and point out their shortcomings.

\(^5\)Duplications due to message retransmissions are not considered as resolution duplications and duplications due to RLs are considered separately.
CHAPTER 3. INTERNET GROUP NAME RESOLUTIONS

Essentially, the methods can be classified into centralized and distributed approaches depending on what name resolution scheme is used. The distributed methods can be further classified into dynamic and static approaches, depending on when resolution and duplication loops are detected and controlled.

We do not consider disallowing loops a reasonable solution, because it simply shifts the responsibility of loop control from the system to the users. Also, excluding DLs rules out the possibility of sharing groups in a system. It is worth noting that even though duplication suppression can be achieved at the end receivers using the traditional sequence number technique, propagating duplicates wastes internet link bandwidth.

3.3.1 Centralized Approach

The centralized approach uses centralized name resolution. Resolution loops can be easily detected since the site that performs name resolution has complete knowledge of the receiver group membership. Also, resolution duplications can be suppressed by removing all duplicated pids from the distribution list.

The problem is that either a separate copy per destination is sent, thus wasting internet link bandwidth, or the distribution list is included in each message, resulting in a variable message header and complex receiver and forwarder communication software [Waters et al 84]. Furthermore, the overhead of name resolution depends on how the name graph is maintained. If the name graph is stored in a centralized site, it is vulnerable to site failure. If the name graph is distributed in the network, it takes time to collect name bindings. For the purpose of efficiency, techniques like name caching or name server replication may be used to avoid overloading the central server. Consistency among name caches and replicated servers has to be maintained.

Another problem is related to security. To resolve a gid, the site that performs name resolution must have read permission to all the subgroup membership lists of the destination group [Deutsch 84]. This is a stronger requirement than a simple reference permission.\(^6\) In

\(^6\)Three types of permission can be defined for group membership access. (i) reference: the ability to refer to a
a network environment where personal workstations can be completely under the control of individual users. this may expose confidential group membership to bogus users.

3.3.2 Distributed Dynamic Approach

The distributed approach uses distributed name resolution. Many problems in the centralized approach can be simplified or avoided. For instance, if the nested group model is adopted, name resolution for a subgroup can be done within a subnet and the individual members of a local subgroup are not exposed to anyone outside the subnet. Also, it is only necessary to send a single copy of the message per subnet, incurring much less internet link traffic compared with the centralized approach. Furthermore, because name bindings are maintained by distributed name servers in the subnets, achieving consistency can be confined to the subnet. On the other hand, since none of the name servers has a complete and up-to-date picture of the global name graph, RL and DL control is more difficult than for the centralized approach.

In distributed dynamic methods, loops are detected and action taken at the time of message transmission. Basically, there are two types of dynamic methods: message-based and node-based. The former stores the name graph traversing history in the message itself; the later saves this information at each group node that the message has visited.

Message-Based Scheme

In a message-based scheme, a list is carried with each group message. This list contains the identifiers of the subgroups that have been expanded so far. Before expanding a group $x$, this list is examined. If $x$ is found in the list, an RL is detected and the message is discarded. This scheme requires extra communication bandwidth to carry the list with each message. Because the list is dynamically expanded at each node along the name resolution path, a variable length of group message header results. Also, since a node discards a message only if its gid

---

group through its gid without finding out the individual members in the group; (ii) read: the ability to determine the membership of a group identified by a given gid; and (iii) write: the ability to modify the membership of a group identified by a given gid.
is contained in the list in the message, duplicated copies reaching a node via different paths cannot be suppressed.

**Node-Based Scheme**

Compared to the message-based scheme, a node-based scheme trades memory cost for shorter and fixed length messages. Messages are uniquely identified and each node remembers the identifiers of the message that have been expanded. An RL is detected when the same message needs to be expanded again. The duplicated message is then discarded. Because this scheme depends on message identifiers, rather than on the path that a message has traveled, duplicate messages with the same message identifier will be suppressed at a sink.

The node-based scheme prevents message retransmissions, however, since retransmitted messages bear the same identifiers as their originals.\(^7\) A remedy to this problem is to allow a message to be resolved at most \(k(\geq 1)\) times through each node,\(^8\) or to discard remembered messages in a regular interval. In both cases, the possibilities of resolution loops and duplications are not completely eliminated. An alternative is to distinguish a retransmitted copy from the copies generated by the RL or DL. One way of doing this is to associate a retransmission counter with each message. This counter is set to zero in the original copy and is incremented by one at the message sender when the message is retransmitted. Under the assumption that message identifiers are globally distinct, a group member could use the message identifier to detect retransmissions of a received message. Nodes along the name resolution paths treat messages with the same identifier but different values of retransmission counter as different messages.

A more serious problem with node-based schemes is that it can only be approximated in practice, since theoretically it requires every node that has subgroups to remember the identifier of every message it has ever expanded. This information must be kept until notification of

---

\(^7\) Detecting duplications at leaf nodes so that duplicate messages are not seen by group members is a variation of the node-based scheme.

\(^8\) The integer \(k\) could be the maximum number of retransmissions allowed by the protocol before reporting an exception.
termination of the message transaction is received. Alternatively, if message lifetime can be bounded, the name expansion record of a message can be cleared after its life has expired. Unfortunately, message lifetime usually cannot be predetermined in practice. Furthermore, for efficiency reasons, message expansion records need to be kept in RAM for quick reference. In practice, it is unrealistic to have an arbitrary large RAM. RAM message buffers often have to be cleared on a regular basis, which requires an intelligent distributed garbage collection algorithm. This can be difficult as the algorithm must adapt to the dynamic changes of internet group configuration as well as the variable network traffic load.

One approximation of the node-based scheme is to maintain a FIFO queue of messages at each group node. A message is resolved if its identifier is not in the queue, and its identifier is saved at the end of the queue after the expansion. For a queue of size $k$, it can be guaranteed that the last $k$ messages will not be duplicated. The value of $k$ can be decided on the basis of experiment. As $k$ goes to infinity, one can approximate the original node-based scheme. The key problem is how to decide $k$ dynamically to adapt to the changing network traffic load and interconnections. Unless $k$ is very large, the possibility of resolution loop and duplications is not completely removed.

Besides these pitfalls in dynamic methods, a performance consideration is that a long list may have to be searched at every node for every message to detect resolution loops and duplications, causing run time overhead and delay.

The advantage of dynamic methods (over the static method discussed later) is that the name graph update operations are simple and their latency can be made small [Deering 90], since no special action needs to be performed and no global knowledge is required.

### 3.3.3 Distributed Static Approach

Static loop avoidance methods do not require every message to be buffered or examined at every node along the resolution path, they do not record in each message the set of nodes resolved so far either. Instead, each node statically saves some topology information about the name graph and detects RLs and DLs on the basis of this information when the name
CHAPTER 3. INTERNET GROUP NAME RESOLUTIONS

The name graph is modified via join and leave operations. To control RLs and DLs, the name graph is transformed into a special structure that, together with the loop avoidance and duplication suppression algorithms, meets the requirements specified in Section 3.2.

The rationales of the static approach are based on the following observations:

- As long as the final result of the result of name resolution process for the destination group is correct (i.e., satisfies B1), it does not matter to a group message sender how this result is obtained or what name resolution paths are taken.

- Name graph modifications generally occur much less frequently than name resolutions. According to statistics from CDNnet [CDNet 90], the ratio of electronic mail name resolutions to email distribution list updates was more than 60:1 in February 1990. Private communications with BITNET managers also confirmed this observation.

Static methods completely remove resolution loop and duplication effects, but dynamic methods can only approximate this. Static methods require no heuristic to estimate how long a message identifier has to be buffered at a node, or the management of a message buffer at each node for name resolutions. The name graph topology information for the purpose of loop detection can be saved on disk because it is needed only when the name graph is updated, rather than during message transport. With the static approach, group name resolution is less expensive as messages do not have to be checked at every node to determine whether they have been resolved as in the dynamic cases. Therefore, better group message transport performance can be expected than in the case of dynamic schemes.

In high speed networks, it is particularly desirable to reduce processing overhead of control information during data transfer, since in such networks, the performance bottleneck has shifted from the network to the nodal processing required to execute communication protocols in workstations and servers. A node with limited processing power must minimize the processing

---

9It has been observed that more updates occur during the early stage of a list's lifetime before it becomes stable. It is also observed that most updates in the early stage of an email distribution list add new subscribers. A sublist is formed only when a significant number of subscribers from the same organization are collected. These were observed from a large number of CDNnet email distribution lists.
overhead per message to avoid congestion and to take advantage of the large bandwidth of high speed networks. Therefore, the property of lower processing overhead during message transport makes static methods more suitable to high speed networks.

The negative side is that modifications to the name graph are expensive in static methods because of the overhead in loop detection and the overhead in propagating the name graph topology updates to descendants. Since modifications generally occur much less frequently than name references, the overall average overhead can be expected lower than in the dynamic schemes. In other words, the static scheme trades higher overhead in name graph updates for cheaper and faster group name resolution performance.

3.4 Related Work

[Birman and Joseph 87a, Benford and Onions 87, Chang and Maxemchuk 84b, Cooper 84b] and [Cheriton and Zwaenepoel 85] present some general discussions on existing support for multicast or group communication. In most systems, group communication is limited to single level groups. One of the contributions in this chapter is to organize internet groups according to the nested group model to achieve communication efficiency and subnet autonomy. Compared to the tree structure of host groups [Cheriton and Deering 85] and channel groups [Frank et al 85], our model is more flexible in allowing groups to be structured as an arbitrary directed graph. The name graph model is a global abstraction independent of how each node is physically supported; host and channel groups can be viewed as specific implementation techniques for maintaining group membership information in a subnet.

[Benford and Onions 87, Deutsch 84] describe some work in dynamic resolution loop detection in the context of electronic mail distribution lists.

3.5 Chapter Summary

Allowing nested groups in an internet environment provides a structured way of organizing internet groups, thus supporting subnet autonomy and reducing internet link traffic. The
problems of resolution loops and duplications result, however. We identified these problems using a formal group name graph model and analyzed the correctness criteria of the algorithms handling these problems. We also classified different methods into centralized and distributed approaches, depending on how and where name resolution is performed, and further classified distributed methods into dynamic and static categories, depending on when loops are detected and control actions taken. By analyzing the shortcomings of existing methods, we find that none of the existing centralized or distributed dynamic approaches work correctly or satisfactorily. We also expect that static distributed methods maybe more efficient than dynamic methods when group name resolution occurs much more frequently than changes in group membership. In this case, which is common in practice, making the frequent operations as efficient as possible reduces the overall system overhead and delay. These observations motivated us to work on the static method presented in the remainder of this thesis.
Chapter 4

Spanning Shadow Tree Algorithm

In this chapter, we recognize name resolution consistency as the correctness requirement for distributed static methods that handle RLs and DLs. We then design a spanning shadow tree algorithm that saves the name graph topology to detect RLs and DLs when the name graph is modified via the join and leave operations. While preserving name resolution consistency, update operations transform the system-level representation of the name graph into a special structure to avoid resolution loops and suppress duplications.

This chapter is organized as follows. Name resolution consistency is defined in Section 4.1. The basic idea of removing the effect of RLs and DLs are discussed in Section 4.2 and Section 4.3 respectively. In Section 4.4 and Section 4.5, we describe how to statically save the name graph topology and how to use the saved information in detecting RLs and DLs. The join and leave operations are presented in Section 4.6 and Section 4.7 respectively and their correctness is argued in Section 4.8. A brief overview of related work is given in Section 4.9.

We assume that no specific order (required from the user level) is imposed on the traversal of nodes in an RL. To simplify the discussions, we also assume that the network is reliable and that name graph updates do not occur concurrently. Removing the last two assumptions will be the subject of Chapter 6.
4.1 Name Resolution Consistency

Define the external group view (EGV) as the name graph perceived by omniscient users having global information, and the internal group view (IGV) as the internal representation of the name graph maintained by the naming system. The naming system implementing static RL/DL control algorithms must guarantee that the results of name resolution based on the EGV and the IGV are always consistent, so that users need not be aware of the existence of the IGV graph. This is called name resolution consistency. In the following discussion, we simply use the EGV (or IGV) to refer to the EGV (or IGV) graph.

A naming system generally has two parts: name resolution and name graph maintenance. The former interfaces with users and performs name resolutions using the IGV. The latter interfaces with system administrators through join and leave operations to change the name graph and maintain name resolution consistency between the IGV and the EGV.

In centralized or distributed name resolution that uses dynamic RL/DL control schemes, the EGV and the IGV are identical. Join/leave operations simultaneously modify both views as discussed in Section 3.2.1. In static schemes, however, the topology of the IGV may be different from that of the EGV since join/leave operations may have to modify the IGV in a way different from what the user perceives in modifying the EGV in order to avoid RLs. Let \((N, A)\) denote a di-graph with a node set \(N\) and an arc set \(A\). An IGV = \((N_{IGV}, A_{IGV})\) is name resolution consistent with respect to an EGV = \((N_{EGV}, A_{EGV})\) if:

**C1:** \(N_{IGV} = N_{EGV}\); and

**C2:** for any pair of nodes \(x\) and \(y\), there is an \(x \xrightarrow{\star} y\) name resolution path in the IGV if and only if there is an \(x \xrightarrow{\star} y\) path in the EGV.

When C1 and C2 are met, the results of name resolution for the same group in both views are identical. The only difference between the IGV and the EGV is that name resolution paths may traverse the same set of group nodes in a different order in each view. Because this order is unimportant to users as far as name resolution is concerned, the IGV is transparent.
4.2 Handling Resolution Duplications

Intuitively, a good place to enforce resolution duplication control (RD control) in a $DL(s, k)$ is at the sink node, where the duplications caused by $DL(s, k)$ are first observed. Since the effect of $DL(s, k)$ takes place only when a message originates at $s$ or at the ancestors of $s$, it is difficult to build a static structure in the name graph to meet requirements D1 and D2. We take the following semi-static approach:

- Every message carries the gid of its originator group — the group to which the message was first sent (from the user).

- In each DL, one of its segments is selected as the primary, the others as secondary. Accordingly, to shorten the notation, the immediate parents of the sink in these segments are called primary or secondary parents (in this DL).¹ To decide when to take the RD control action for a $DL(s, k)$, a node maintains an RD control record containing a list of ancestors of the source (including the source). During name graph updates, this list is computed by the sink (discussed later) and distributed to all secondary parents of the sink in $DL(s, k)$. When such an RD control record is maintained for an arc leading from a secondary parent in $DL(s, k)$ to the sink, we say that arc is RD controlled for $DL(s, k)$.

- The name resolution procedure is changed so that if the originator group of a message is found in the RD control record of an arc, the message is not forwarded through that arc.

Figure 4.1 shows some examples in which the dashed arcs are RD controlled. The notation $< DL(s, k) : a_1, \ldots, a_k >$ shows the RD control record for $DL(s, k)$, in which “$a_1, \ldots, a_k$” is the list of ancestors of the source node. In Figure 4.1.(A), $DL(a, f)$ is controlled at node $c$, $DL(a, e)$ is controlled at node $d$, and $DL(d, g)$ is controlled at nodes $e$ and $f$. The RD control records for $DL(a, g)$ at nodes $e$ and $f$ are not shown in the figure because they contain only a subset of the RD control record for $DL(d, g)$.

¹ The communication cost of the last hop in each segment may be taken into account in selecting the primary.
Figure 4.1: Examples of duplication loop control.

A $DL(s', k')$ is embedded in another $DL(s, k)$ if both its source $s'$ and sink $k'$ are in $DL(s, k)$. Figure 4.1.(B) shows an example where the $DL(c, g)$ is embedded in $DL(a, g)$. $DL(c, g)$ is RD controlled at node $f$, $DL(a, g)$ is RD controlled at nodes $f$ and $e$.

One must be careful in selecting the primary segment for DLs that have the embedding relationship and share the same sink node. For example in Figure 4.1.(B), if RD control for $DL(a, g)$ is performed at nodes $d$ and $e$ and RD control for $DL(c, g)$ is performed at node $f$, messages to or routed through node $a$ will never be delivered to node $g$. The reason is that the RD control record for an embedded DL contains the source node of the embedding DL. To avoid this problem, a sink $k$ must select its primary parent in $DL(s, k)$ as its primary parent in $DL(s', k)$ if $s \Rightarrow s'$ in the name graph.

This RD control scheme preserves name resolution consistency because it does not change the name graph. While traversing the name graph to perform name resolution, the RD control algorithm dynamically enforces the message propagation paths to be a spanning tree in the subgraph rooted at the message originator so that each node in that subgraph is visited exactly once. When a message is routed through the source of a DL in that subgraph, the RD control procedure removes the last hop leading to the sink from every secondary segment in that DL and only allows the message to be propagated to the sink once through the primary segment. In this
way, resolution duplications are suppressed. Furthermore, this scheme imposes no restriction on message retransmission.

It is worth noting that the secondary parents of the sink in each DL have to do a table look-up when resolving a message, and some amount of RAM is required at these nodes to store the RD control record for efficient name resolution. The length of the list, however, is finite (less than the total number of nodes in the name graph) and normally is small compared to the message list in the node based scheme since generally the name graph is not very complicated. Furthermore, except for the secondary parents of a sink node, other nodes do not pay such memory and run-time overhead.

4.3 Handling Resolution Loops

When adding a new arc \(<x, y>\), an RL is generated if and only if \(y \rightarrow^* x\). This RL consists of all the nodes along the \(y \rightarrow x\) path. When a DL or any existing RL is embedded in the \(y \rightarrow x\) path, multiple cycles will be contained in this new RL. These cycles are chained together.

The basic idea of removing the RL effect is to replace each cycle by a chain of two-node cycles and enforce control on accessing these two-node cycles. We define a shadow edge as a bidirectional arc and a shadow path as a path consisting of shadow edges only. Observe that nodes in a shadow path are loop deep equal because each shadow edge in fact is a two-node cycle and a shadow path is a chain of such cycles. A cycle with more than two nodes can be replaced by a shadow path with nodes in the cycle remaining loop deep equal. On the basis of this observation, an \(<x, y>\) arc is not added in the IGV if its addition generates any cycle. Instead, the existing \(y \rightarrow x\) path in the IGV is converted into a shadow path.

The name resolution procedure is modified to control the RL effect in a two-node cycle. Instead of resolving a message to all out-going arcs as before, each node in a name graph adopts the new resolution procedure shown in Figure 4.2.

If all cycles are replaced by shadow paths, then under the conditions that

\[\text{RD control is still performed on each out-going arc based on the message originator and the RD control record associated with the arc.}\]
<Assume a message coming in from an arc \( l \)>

for each shadow edge or outgoing arc \( l' \) in the IGV do

if (\( l \) is a shadow edge and \( l' \neq l \)) or

(the originator of the message is not in any RD control record maintained for \( l' \))

then forward the message across \( l' \)

Figure 4.2: The new name resolution procedure.

1. no other path exists between any two nodes on the same shadow path, and

2. everyone in the name graph follows the new resolution procedure,

we can proof the following theorem:

**Theorem 4.3.1** Messages to any node in a shadow path will be resolved at every other nodes in the path once and only once.

**Proof:** First observe that the new name resolution procedure does not have any effect on messages coming in or going out through normal arcs, that is, converting a cycle into a shadow path does not affect name resolutions at nodes outside the cycle. Second, the above argument can be proven by induction on the distance (number of arcs) between a pair of nodes, say \( x \) and \( y \), in a shadow path. When the distance is one (\( x \) and \( y \) are connected by a single shadow edge), a message to \( x \) will be delivered to \( y \) and \( y \) will not send it back to \( x \) through the same arc according to the new procedure. Because there is no other path between \( y \) and \( x \) (condition 1), both \( x \) and \( y \) resolve this message once and only once. Assume the statement is true when the distance between \( x \) and \( y \) is \( n \) (> 1). Consider the case when the distance is \( n + 1 \). In the shadow path between \( x \) and \( y \), there is a node \( z \) whose distance from \( x \) is \( n \), and the distance between \( z \) and \( y \) is one. According to the induction hypothesis, a message to \( x \) will be resolved at \( z \) once and only once. After it reaches \( z \), this message will be resolved at \( y \) once and only once as well according to the induction base. □

Given the above results, we only need to guarantee that all nodes involved in an RL are connected by a shadow path and that no other path exists between them after the shadow path
construction.

Multiple $y \overset{x}{\rightarrow} x$ paths are involved in a loop if at least one DL is embedded in the loop. If every arc in an embedded DL were shadowed, a shadow loop would be generated, since a DL consists of more than one segment between its source and sink and every segment becomes a bidirectional path after shadowing. On the other hand, when no DL is embedded in a loop, there is only a single path between any pair of nodes involved in the loop; thus, only this single path is shadowed. To handle the case when embedded DLs are involved, we extend the shadow path idea by constructing a spanning shadow tree among all nodes in all existing $y \overset{x}{\rightarrow} x$ paths such that there is only a single shadow path between any pair of these nodes.

The nature of a shadow edge guarantees that nodes in the tree remain loop deep equal and the one-way propagation property of the new resolution procedure prevents infinite resolution effects in the shadow paths. Therefore, requirements R1 and R2 are satisfied. The shadow tree algorithm preserves name resolution consistency. Because all nodes connected by an RL in the EGV are connected by a shadow tree in the IGV, these nodes are mutually reachable in both views. Also, the shadow tree algorithm does not add any arc, nor does it delete an arc if the arc is not in an RL. Therefore, the reachability between any nodes in the name graph remains unchanged and hence requirements C1 and C2 are satisfied.

Figure 4.3 shows an example of the shadow tree scheme. In Figure 4.3.(A), the addition of arc $< k, a >$ completes an RL consisting of all the nodes and arcs shown in the graph. There are four duplication loops $DL(b, g), DL(c, f), DL(d, g)$ and $DL(h, j)$ embedded in the RL. Figure 4.3.(B) shows the result of spanning shadow tree construction.

4.4 Representing Topology of Ancestors

RLs and DLs have to be detected before the above schemes can be used to deal with them. To detect an RL when adding an arc $< x, y >$, node $x$ must have some way of knowing if any $y \overset{x}{\rightarrow} x$ path exists in the name graph. If that is the case, it needs to know all the nodes in this path to construct the shadow tree. Also, after adding an arc $< x, y >$, a node in the subgraph of
\textit{y} has to determine whether it has become the sink of a new DL or whether a new segment has been added to a DL of which it is the sink. If so, it has to compute the RD control record and distribute this record to its secondary parents in the DL. In summary, RL and DL detections require a node to know the topology of its ancestors. A mechanism to record this topology is described in this section.

### 4.4.1 Virtual Node and Derived Name Graph

In a name graph, a \textit{physical} node refers to a single group node; a \textit{virtual node} (VN) refers to a strongly connected component in the graph. A VN consists of nodes that are loop deep equal and the arcs connecting these nodes. It corresponds to an RL in the EGV or a shadow tree in the IGV. A VN has a unique id (vnid) similar to a gid. Nodes in a VN form a \textit{loop deep equal set} and are called the \textit{components} of the VN. In this thesis, RL and VN are used interchangeably.

In a name graph, if each strongly connected component is substituted by a VN, the resulting
graph is called *derived name graph*. The properties of VN and the derived name graph are summarized below:

- A derived name graph is a directed acyclic graph (DAG).

- Since nodes inside a VN are loop deep equal, the ancestor-descendant relation in a name graph is completely defined by the arcs across different loop deep equal sets (i.e., the arcs in the derived name graph).

- The internal connection structure of a VN is immaterial as far as RL and DL detections are concerned, hence need not be known to anyone outside the VN.

- The components of a VN can be arbitrarily connected as long as they remain mutually reachable (for name resolutions).

The second and third points suggest that it is sufficient for a node to keep the topology information of its ancestors in the derived name graph to detect new RLs and DLs. The last point suggests that the name graph maintenance mechanism may choose whatever IGV structure to connect components in a VN as long as its derived name graph is the same as that of the EGV. Spanning shadow tree is an example.

In the following discussions, an arc is in the *normal* state if it connects two nodes that are not loop deep equal. A normal arc is unidirectional. There is a one-to-one correspondence between the normal arcs in the name graph and the arcs in the corresponding derived name graph. An arc is in the *shadow* state if it connects two nodes in an RL in the EGV and is part of the spanning shadow tree of the VN in the IGV. A shadow edge is bidirectional. Name resolution on a shadow edge follows the procedure in Figure 4.2. An arc is in the *fade* state if it connects two nodes in an RL in the EGV but is deleted from the IGV to avoid shadow loop (when spanning shadow trees are constructed). Fade arcs are ignored in name resolutions. A shadow edge or a fade arc is *restored* when its state is reset to normal.
4.4.2 Pathnames

A pathname scheme is adopted for a node to keep track of its ancestors’ topology in the derived name graph. A *pathname* $P$ is defined as a sequence of gids, $P = < x_1, x_2, \ldots, x_n >$, representing an $x_1 \rightarrow x_n$ path in the derived name graph. Given a name graph, pathnames are constructed and assigned to a node as follows:

1. Each node with in-degree zero has a pathname consisting of its gid/vnid only.
2. A node $x$ has a pathname $< P_{x_p}, x >$ if and only if an immediate parent $x_p$ of $x$ has a pathname $P_{x_p}$; that is, $x$ inherits the pathnames from its parents.
3. In pathnames, the nodes in an RL are represented by a single VN. Components of a VN share the same set of pathnames.

Identifying an RL as a VN hides the VN’s components and their connections from their descendants, hence, simplifies the pathnames of the descendants. The lengths of pathnames are finite because the derived name graph is a DAG with a finite number of nodes. As far as the descendants are concerned, there is no need to distinguish between a VN and a physical node in pathnames for the purpose of RL/DL detection.

Figure 4.4 provides examples of pathnames, in which gids/vnids are shown in bold face and pathnames are included in angle brackets. Figure 4.4.(B) shows how a shadow tree is represented by a VN $t$. The collection of pathnames maintained in a node is called the *pathname set* of that node. Obviously, all components in a VN share the same pathname set.

By constructing pathnames in this way, it is clear that:

1. The pathnames of a node $x$ terminate with the gid of $x$.
2. All components in a VN share the same set of pathnames (terminated by the vnid).
3. For each path from a root node to $x$ in the derived name graph, $x$ has a corresponding pathname.
4. If \( x \) is an ancestor of \( y \), \( x \) occurs in some pathname(s) of \( y \) and each pathname of \( x \) appears as a “prefix” in some pathnames of \( y \).

The pathname set at a node completely represents the topology of its ancestors in the derived name graph. The pathname set must be updated accordingly when the topology of ancestors is changed, since the correctness of RL/DL detections in future name graph updates will depend on the consistency of the pathname set. The distributed procedure of updating pathname sets of the nodes in a name graph is called pathname update relay.

For instance, after a normal arc \( < x, y > \) is added, all paths leading to \( x \) are extended to \( y \). Therefore, \( y \) and its descendants must update their pathname sets by adding the pathnames inherited from \( x \). To start the pathname update relay, an update request containing the pathname set of \( x \) is propagated through the subgraph rooted at \( y \). A node receiving an update request simply appends its gid (or its vnid if it is a component of a VN) at the end of each pathname contained in the request.\(^3\) includes the resulting pathnames in its pathname

\(^3\)If the node is in an VN and the update request is received from another component in the same VN, the
set, and relays the request to all of its immediate children after replacing the pathname set in the request by the inherited pathnames. Threads of pathname update relay terminate at leaf nodes (i.e., groups with process members only). Pathname update relay in other cases (such as adding a non-normal arc or deleting an arc) will be discussed in later sections.

4.5 Detecting RLs and DLs

RL detection is simple if a pathname set is maintained at each node. When adding an arc \(<x,y>\), node \(x\) learns that \(y\) is an ancestor if \(y\) occurs in any of its pathnames. Then all nodes in \(x\)'s pathnames from \(y\) to \(x\) (including \(x\) and \(y\)) are the components of the new RL.

For a node to detect that a new segment is leading to it from one of its ancestors, it has to do DL detection after every pathname update. A node \(k\) detects a \(DL(s,k)\) if

- \(s\) appears in at least two different pathnames of node \(k\), and
- in these pathnames, \(s\) is the common node with the shortest distance from \(k\) and is not an immediate parent of \(k\) in all of these pathnames.

When this occurs, \(k\) selects a primary segment as discussed in Section 4.2, computes and distributes an RD control record to its secondary parents in \(DL(s,k)\). When a new segment of the DL is added, the new segment becomes a secondary and the same RD control record is assigned to the parent of the sink in that segment as well. The RD control record of \(DL(s,k)\) can be computed by node \(k\) since every path that leads to \(s\) are extended to \(k\) and pathnames of \(s\) are embedded as prefixes in some pathnames of \(k\). Nodes in these prefixes are ancestors of \(s\) and their ids are contained in these pathnames before \(s\).

Instead of having the sink compute and distribute the RD control record, the sink may tell its secondary parents about the source of the DL and let them compute the RD control record themselves. In this way, the size of the messages from the sink to these parents can be reduced.

\footnote{In a pathname, \(m\) is before \(n\) if \(m\) identifies a node that is an ancestor of the node identified by \(n\).}
Because the RD control record is computed from pathnames and the vnid (rather than the gid of the originator) is recorded in the RD control record, a message originating at a component of a VN must have the VN’s vnid in its originator field for the purpose of RD control.

It should be noted that pathname update relay requests are not RD controlled because different copies of the request travel through different segments of the DL, bringing to the sink different pathname information. To increase efficiency, a sink may wait if the update request originates from the source or is routed through the source until an update request is received from every segment. Then the pathname update request is relayed.\(^5\)

### 4.6 Join Operation

A static algorithm consists of two name graph update operations: *join* and *leave*. In this section, we present a *join* operation that combines the above ideas. Before going into the detail, we briefly describe VN topology information and outline a construction algorithm of spanning shadow tree.

#### 4.6.1 VN Topology Information

The topology information of a VN consists of three parts:

1. The *EGV connection topology* of a VN is the interconnections among all the components of the VN in the EGV. This topology information can be represented by an adjacency matrix [Bondy and Murty 76].\(^6\)

2. A *parent list* containing information about all immediate parents of the VN,\(^7\) including

---

\(^5\)An alternative way of reducing message overhead is to RD control pathname update requests. The sink, upon receiving a update request from the primary segment, not only inherits the pathnames in the request but also generates the pathnames which would be passed down from the secondary segments. It can do so because it can find, from the local pathname set, the paths from the source node through the secondary segments to itself, and from the update request, the paths extended to the source node due to the arc addition. Both inherited and locally generated pathnames are relayed.

\(^6\)Adjacency matrix is a standard data structure used to represent graph topology.

\(^7\)A node is an immediate parent (or child) of a VN if it is an immediate parent (or child) of one of the components of the VN.
the gid of each physical immediate parent, the vnid of the VN in which that parent is a component (this vnid is undefined if the parent is not in any VN), and the component(s) to which that parent node is connected.

3. A child list containing the information about all immediate children of the VN in the same format as the parent list.

The topology information of a VN is collected when the VN is formed and is replicated at all the components. It is used when a new VN is formed or when an existing VN is broken by a leave operation.

4.6.2 VN Shadow Tree Construction

Once the EGV connection topology of a VN is collected, a number of existing algorithms maybe used to construct a spanning tree for the VN. We adopt the well known Kruskal algorithm [Bondy and Murty 76] because of its simplicity. Figure 4.5 shows the algorithm.

```
Put all the EGV arcs of the VN into an arc_list and sort the arc_list;

Remove the first arc (denoted by \(a, b\)) from arc_list;

\[
\text{tree\_nodes} = \{a, b\}; \quad /* \text{list of tree nodes} */
\]

\[
\text{tree\_arcs} = \{< a, b >\}; \quad /* \text{list of tree arcs} */
\]

\[
\text{while \((\text{sizeof} (\text{tree\_nodes}) < \text{sizeof} (\text{VN}))\) do begin}
\]

\[
< x, y > = \text{get\_next\_arc}(\text{arc\_list}, \text{tree\_nodes}); \quad /* x \in \text{tree\_nodes} */
\]

if \((y \text{ is not in } \text{tree\_nodes})\) then begin

\[
\text{tree\_nodes} = \text{tree\_nodes} \cup \{y\};
\]

\[
\text{tree\_arcs} = \text{tree\_arcs} \cup \{< x, y >\};
\]

end

end

return(\text{tree\_arcs});
```

Figure 4.5: Spanning tree construction.

In sorting the arc_list, a comparison order between a pair of arcs is defined as follows: \(< x, y > \prec \prec < x', y' >\) if \(x < x'\), or \(x = x'\) and \(y < y'\). Since gids are assumed globally unique and no parallel arcs between physical nodes are allowed in the EGV, this comparison order between
arcs is well defined. The get_next.arc() takes two parameters: arc_list — the list of arcs yet to be considered, and tree_nodes — the list of nodes already connected by the tree. It removes and returns the smallest arc in the arc_list originating from a node in the tree.

After a component has computed the list of tree arcs, the state of its adjacent arcs can be determined. An adjacent arc is shadowed if the arc is in the spanning tree. It is fade if the arc connects to another component in the same VN, but the arc is not in the list of tree arcs. The arc is set to normal if it connects to a node outside the VN.

A key point here is that every component must have exactly the same topology information of the VN and run the same algorithm for the tree construction, so that the same list of tree arcs is obtained. As will be seen in Section 4.6.3, this is guaranteed by collecting and distributing the topology information of the new VN through a single coordinator.

In practice, one may take the communication cost (delay, bandwidth, link utilization, reliability etc.) of arcs into consideration to construct a minimum spanning tree. The cost information may be collected while the VN topology is gathered and saved as part of the VN topology information. The modifications to the above algorithm would be:

- each arc is associated with a weight representing the communication cost of that arc and the arc_list is sorted according to the weight; and
- the arc returned from the get_next.arc() is the arc (originating from a node in the tree_nodes) with the smallest weight in the arc_list.

4.6.3 Operation Description

Consider a join operation that adds an arc < x, y > and let P_x denote the pathname set of node x. First, the RL detection algorithm in Section 4.5 is invoked at x using P_x. Depending on the result, we have the following cases:

No new RL is formed

If the new arc connects two nodes that are loop deep equal before the join (i.e., x and y
are components of an existing VN), all components in that VN are informed (by $x$) to update the VN topology information to include a fade arc $< x, y >$. No pathname update relay is necessary since no ancestor-descendant relationship is changed.

If $x$ and $y$ remain not loop deep equal after the join, a normal arc $< x, y >$ is added and a pathname update is relayed in the subgraph rooted at $y$ to inherit the pathname set of $x$ (refer to Section 4.4.2). If $x$ is in a VN, that VN becomes an immediate parent of $y$. Components in that VN have to update their VN child list to include $y$. Similarly, if $y$ is in a VN, components in that VN learn about the new immediate parent $x$ during pathname update relay and update their parent list to include $x$ accordingly.

**A new RL is formed**

The subset of pathnames $L = \{ p \mid p \in P_x \land y \in p \}$ identifies all the paths routed through $y$ to $x$. The node set $N = \{ n \mid \exists p \in L \text{ such that } n \in p \land y \not\sim n \}\cup\{y\}$ contains the nodes (both physical and VNs) in the new RL. We shall call this set the *component list* of the new VN.

Serving as the coordinator for the join operation, $x$ contacts every element in the component list to collect the topology information for the new VN. Every physical node contacted by $x$ returns its *local topology information* — the list of its immediate physical parents and children in the EGV (including the gid of the parent/child and the vnid of the VN in which that parent/child is a component). If an element in the component list is a VN, its membership is returned, so that $x$ can expand the component list (by substituting the returned member list for the VN in the component list) and then obtain the local topology information from each physical node member.\(^8\)

On the basis of the information gathered, an adjacency matrix recording the EGV connection topology of the new VN and the lists of parents and children of the new VN can be constructed. Node $x$ also assigns the new VN a vnid. All the paths to the components in the new VN can be recognized from pathname set $P_x$, since these nodes are ancestors of $x$ before arc $< x, y >$ is

---

\(^8\)An alternative that reduces the number of exchanged messages is to directly receive the topology information of an embedded VN from its member.
added and each path to these nodes is contained as a prefix in \(x\)'s pathnames. The pathnames of the new VN can be derived from \(P_x\) by substituting these components with the vnid.

A pathname update relay is launched in the subgraph rooted at \(x\). The above derived pathname set is contained in the pathname update relay request. \(x\) sends the pathname update request to other components in the new VN. The topology information of the new VN is piggybacked, therefore is replicated at every component.

Upon receiving a join pathname update, a node \(n\) does the following:

• If \(n\) is a component of the new VN, it executes the shadow tree construction algorithm described in Section 4.6.2 and sets the state of its adjacent arcs accordingly, adopts the new vnid and saves the topology information of the VN, and replaces its pathname set by the set of pathnames in the update request.\(^9\) The update request is then relayed only to children connected by normal arcs. If any DL for which \(n\) performs RD control is embedded in the new VN, the corresponding RD control record is discarded.

• If \(n\) is not a component of the new VN, it performs pathname relay as described in Section 4.4.2. Because nodes in the new VN are no longer visible in the derived name graph, pathnames containing the ids of these nodes (including the vnids of old VNs that are embedded in the new one) are eliminated.

After completing each pathname update, a node runs the DL detection procedure (refer to Section 4.5) and updates its RD control record. Within a VN, only the component with the smallest gid needs to do DL detection.\(^{10}\)

4.6.4 Handling Parallel Arcs

Although parallel arcs are not allowed in a name graph, they may exist in the derived name graph since it is possible that parallel arcs may exist between a physical node and a VN, or

\(^9\)If \(n\) was a component of an existing VN, the vnid and the topology information of the old VN are discarded, since the old VN disappears (it is embedded in the new VN).

\(^{10}\)The reason is that although all components in a VN can do the DL detection, their results are identical and their efforts in DL detections are duplicated.
between two VNs. Parallel arcs must be handled since pathnames represent the topology of the derived name graph. The following steps are included in the DL detection procedure:

1. Given the id of an immediate parent obtained from a pathname, a physical node, say $n$, can find out the number of arcs coming from that parent to $n$, or to the VN in which $n$ is a component. This is possible because the parent list replicated at $n$ provides all the necessary information if $n$ is in a VN. If $n$ is not in any VN, $n$ may find this from its local topology information. Any change to the mapping between the gid of an immediate physical parent and the vnid of the VN in which that parent is a component can be observed and recorded by $n$ when the pathname update request arrives.

2. When more than one arc from an immediate parent to $n$ or to the VN in which $n$ is a component is found, each arc is treated as a segment of a DL, consisting of only a source (the immediate parent found from the pathname) and a sink (node $n$ or the VN in which $n$ is a component). All but one such parallel arcs are RD controlled for the DL.

3. After all pathnames at a node are examined and all parallel arcs are taken care of as stated above, the DL detection procedure described in Section 4.5 is used to deal with DLs containing both parallel arcs and segments longer than one arc. Assume there are $m (> 1)$ parallel arcs from an immediate parent $p$ in a segment $S$ to the sink in a $DL(s, k)$. These parallel arcs form a $DL(p, k)$ which has been taken care of in step 2. If the DL detection procedure chooses $S$ as the primary segment of $DL(s, k)$, all but one parallel arcs from $p$ to $k$ are RD controlled; otherwise, all such arcs are RD controlled should $S$ be chosen as a secondary segment. As stated in Section 4.2, when $S$ is chosen as the primary, the parallel arc from $p$ to $k$ which is not RD controlled must be the primary of $DL(p, k)$ since $DL(p, k)$ is embedded in $S$.

4.7 Leave Operation

From the viewpoint of the users, `leave` is the operation that negates the effect of `join`. A
group $y$, which is an immediate member of another group $x$, quits its membership in $x$ by invoking $\text{leave}(x, y)$. This operation deletes arc $< x, y >$ from the EGV.

In static RL avoidance algorithms, however, the $\text{leave}$ operation may not be that simple if the deleted arc is contained in an RL. The RL containing the deleted arc is called the originating RL of the $\text{leave}$ operation. Depending on the topology of the originating RL and the location of the deleted arc, some nodes in the originating RL may no longer be loop deep equal; others may remain being connected by cycles in the originating RL. Each such cycle contains a subset of nodes and arcs in the originating RL. The cycle is not broken because it does not contain the deleted arc. A set of such remaining cycles is called a retained RL if (i) these cycles were part of the originating RL, (ii) they are not broken, and (iii) they remain chained together after the arc deletion. A retained RL is a new RL, it defines a new loop deep equal relation.

A $\text{leave}$ operation may also generate new DLs or change existing DLs. DLs embedded in the originating VN are hidden but will be re-exposed if its source and sink are no longer loop deep equal after the arc deletion. Re-exposed DLs need to be detected and properly controlled. Also a segment of a DL may be removed after an arc deletion. If the removed segment was the primary, a new primary needs to be chosen. A DL disappears when only one segment is left.

The $\text{leave}$ operation has to reorganize the IGV connections among nodes in the originating RL to maintain name resolution consistency. The ancestor-descendant relation among nodes that are no longer loop deep equal must be reestablished in the IGV. Nodes in a retained RL have to be mutually reachable in the IGV to maintain their loop deep equal relation. Furthermore, pathname sets at the descendants of the originating VN have to be properly updated, so that DLs can be detected and future name graph updates can be performed correctly.

Before discussing the solution to the $\text{leave}$ operation, the following points are observed:

- If the deleted arc is not in any RL, it is a normal arc and its removal does not change any loop deep equal relation. No change is necessary in the IGV except removing the arc.

- An arc deletion affects at most one loop deep equal relation in the name graph. Only IGV connections among nodes in the originating RL need to be reorganized.
When the deletion of an arc \(< x,y >\) breaks a cycle, all nodes in that cycle will become the descendants of \(y\) after the arc removal, irrespective of their ancestor-descendant relation before the cycle was formed. The new ancestor-descendant relation is defined by the \(y \rightarrow x\) paths in the EGV since this is what a user perceives from the EGV.

After deleting a normal arc \(< x,y >\), only those nodes reachable from \(y\) need to be informed of the topology change as far as pathname update relay is concerned. These nodes have to drop the pathnames of the pattern \(< \cdots xy \cdots >\) since paths to \(x\) are no longer directly extendible to \(y\). If arc \(< x,y >\) is not normal and its deletion breaks the originating VN, all the descendants of the originating VN have to drop their pathnames containing the broken VN.

The basic idea of the solution to the leave operation is simple. When a leave operation breaks a VN, the following steps are taken:

1. determine new loop deep equal relations among nodes in the originating VN according to the EGV topology of the originating VN (the result is a set of retained RLs);
2. in the IGV, restore the arcs between retained RLs and construct a spanning shadow tree connecting the nodes in each retained RL; and
3. relay pathname update according to the reorganized name graph topology, and detect DLs and update RD control record according to the new pathname set.

Figure 4.6 shows an example of the leave algorithm. In the example, deleting arc \(< x,y >\) within an RL breaks that RL. Two retained RLs are generated after the removal of \(< x,y >\), one contains nodes \(u, y \) and \(z\), the other contains nodes \(x, v \) and \(w\). The arcs connecting the two retained RLs are restored. \(DL(z,x)\) embedded in the originating RL is exposed after the arc deletion and RD control record for this DL is assigned to arc \(< z,x >\).

**4.7.1 Determining Retained VN**

After deleting a shadow edge or a fade arc, the connectivity among all nodes in the originat-
ing VN can be obtained from the EGV connection topology of the originating VN (represented by an adjacency matrix). Assume that the components in the originating VN are numbered from 1 to \(N\). By definition [Bondy and Murty 76], in an adjacency matrix \(A\), element \(a_{i,j} = 1\) if there is an arc from node \(i\) to node \(j\); otherwise, \(a_{i,j} = 0\). The reachability matrix \(R = \sum_{i=1}^{N} A^i\). A reachability matrix element \(r_{i,j} > 0\) indicates that in the EGV at least one path exists within the originating RL from node \(i\) to node \(j\) after the given arc is removed.\(^{11}\)

With the reachability matrix \(R\), components in the originating VN can be partitioned into equivalence sets, each constitutes a retained VN. Node \(i\) and node \(j\) are in the same set if and only if \(r_{i,j} > 0\) and \(r_{j,i} > 0\); that is, nodes in each set are mutually reachable in the EGV and remain loop deep equal.

\(^{11}\)As a matter of fact, finding retained RLs is equivalent to finding strongly connected components in a graph. Other algorithms for this purpose can be found in [Aho et al. 74].
4.7.2 Operation Description

Consider a leave operation that deletes an arc \( < x, y > \). Depending on whether any loop deep equal relation is changed, we have the following cases:

**No VN is Broken**

When a normal arc is deleted, no RL is broken and no loop deep equal relation in the name graph is changed. A pathname update relay is launched in the subgraph rooted at \( y \) to delete pathnames of the pattern \( < \cdots, x, y, \cdots > \). On the basis of the updated pathname set, a node updates its RD control record to exclude the nodes that are no longer the ancestors of the source of the concerned DL. At the sink of a DL, it is also necessary to check if deleting \( < x, y > \) removes a segment of the DL. If less than two segments is left, the DL disappears and its RD control record at the secondary parents are removed. Otherwise, a new primary segment has to be selected if the primary segment is removed by the leave. If \( x \) is in a VN, other nodes in the VN have to update their VN child lists.

In the case that the deleted arc is not normal and the arc removal does not change any loop deep equal relation (i.e., \( r_{x,y} > 0 \) and \( r_{y,x} > 0 \)), the update is internal to a VN. Only the nodes in that VN need to be informed to exclude arc \( < x, y > \) from the EGV connection topology of the VN. The spanning shadow tree for that VN may have to be recomputed, but no pathname update relay is necessary.

**The Originating VN is Broken**

When deleting \( < x, y > \) breaks an RL (i.e., \( r_{x,y} = 0 \) or \( r_{y,x} = 0 \)), \( y \), serving as the coordinator for the leave operation, informs other nodes in the originating VN of the arc deletion. Each node in the originating VN does the following reorganization work:

- computes the membership of the retained VN in which the node is a member (refer to Section 4.7.1), and runs the spanning shadow tree construction algorithm (refer to
Section 4.6.2) on this retained VN to shadow/restore its adjacent arcs:  

- cleans up the pathname set by removing the pathnames representing paths to nodes other than itself or to any component of the retained VN; and  

- sets the last element of the remaining pathnames to the vnid of the retained VN in which the node is a component (or its gid if the node is not in any retained VN).  

After node $y$ completes its reorganization work and receives confirmation that other nodes in the originating VN have also done so, it launches a leave pathname update relay. The update request contains the deleted arc, the vnid of the broken VN, and the complete pathname set of $y$. Pathname update requests are propagated along normal arcs and shadow edges in the subgraph rooted at $y$ and are not RD controlled.

Upon receiving a leave pathname update, a node does the following:

- Inherits the pathnames in the request and relays the update request. If the node was a component or a child of the originating VN, changes made to immediate parents are observed from the received pathnames and are recorded in the parent list.

- Discards the RD control record of a DL if the vnid of the originating VN is found in that record or if the originating VN is part of the DL. This is because after the originating VN is broken, its components or retained VNs become visible. The RD control record has to be recomputed to reflect these changes.

- Invokes the DL detection procedure (refer to Section 4.5) if the node is not in a VN or if the node has the smallest gid among all components in a VN.

### 4.8 Correctness

In Section 4.2 and Section 4.3, we presented a correctness argument for the shadow tree

---

12 If the node is not in any retained VN, all its adjacent arcs are restored.

13 Alternatively, the coordinator may compute the retained VNs, assign them vnids and distribute the result to other components.
scheme and the RD control scheme. In this section, we briefly argue the correctness of the join and leave operations with respect to the name resolution consistency requirements in Section 4.1. Since both operations do not add/delete any node in the name graph, C1 is always satisfied. According to requirement C2, an update operation in the shadow tree algorithm is correct if after adding/deleting an arc, (1) nodes in each RL in the EGV are connected among themselves by a spanning shadow tree in the IGV, and (2) the ancestor-descendant relationship in the IGV conforms to the derived name graph of the EGV. We show that C2 is satisfied in two steps:

1. Since a join operation constructs a spanning shadow tree for the new RL and a leave operation constructs a separate spanning shadow tree for each retained RL, nodes that are loop deep equal in the EGV are also loop deep equal in the IGV. When the name resolution procedure in Figure 4.2 is used, a message resolved at a tree node will be eventually resolved at all other nodes in the same shadow tree. The order that nodes are visited during name resolution may be different from that expected from the EGV, but this should be of no concern to the users.

2. Except for restoring existing fade arcs or shadow edges across loop deep equal sets, a leave operation does not add or delete any arc between different loop deep equal sets. A join operation does not add or delete any arc outside the new VN. The set of arcs in the IGV between nodes that are not loop deep equal is identical to that in the EGV, therefore, the IGV and the EGV have the same derived name graph and the ancestor-descendant relation among nodes in the IGV are consistent with that in the EGV.

Furthermore, both operations launch pathname update relay after IGV reorganization. The propagation of pathname update requests follows the direction of normal arcs between loop deep equal sets. Hence, pathnames correctly record the ancestor-descendant relation in the derived name graph. When a VN is created or changed, all the descendants of that VN are involved in the pathname update to bring their view of the name graph up-to-date. DLs are detected as soon as information about the topology of ancestors becomes available (i.e., after a node completes each pathname update).
We therefore conclude that both update operations preserve name resolution consistency between the IGV and the EGV.

4.9 Related Work

[Benford and Onions 87, Deutsch 84] describe some work in dynamic resolution loop detection in the context of electronic mail distribution lists. As pointed out in Section 3.3, these schemes do not perform correctly in certain circumstances. The static algorithm not only guarantees correct name resolution, but also provides potentially better name resolution performance because it is not necessary to perform list searching at every node during message transport to prevent resolution loops and duplications.

[Dally and Seitz, Merlin and Schweitzer 80a, Merlin and Schweitzer 80b] discuss deadlock avoidance in message routing in multiprocessors and in message buffer management for store-and-forward networks using deadlock graph transformations. The requirements of graph transformation for deadlock avoidance is different from that for resolution loop and duplication avoidance. As far as we know, there has been no other research on the problem of static resolution loop and duplication avoidance.

4.10 Chapter Summary

In this chapter we designed a spanning shadow tree algorithm that removes the effect of RL and DL while preserving name resolution consistency between the EGV and the IGV. This distributed static algorithm uses a pathname structure for a node to record the topology of its ancestors. With this knowledge, a node can detect if any RL will be formed when an arc is added. The virtual node concept is used to encapsulate a set of nodes that are loop deep equal. This concept is helpful in defining the amount of name graph information required for static algorithms to work.

In a join operation, VN topology information is collected by the coordinator when the VN is formed. The EGV connection topology of a VN is replicated at all its components. Based
on this information, a component constructs a spanning tree and shadows/fades adjacent arcs to/from other nodes in the same VN depending on whether the arcs are in the tree. In a leave operation, each node in the originating VN computes the retained VN based on reachability analysis and runs the spanning tree algorithm to determine the state of its adjacent arcs. DLs are detected after each pathname update and RD control records are assigned to all secondary parents of the sink in each DL to suppress resolution duplications. Name resolution procedure is modified to take shadow edges and fade arcs into consideration for RL control and to check RD control record for DL suppression.

The correctness of the algorithm is argued by showing that the derived name graphs of the IGV and the EGV are the same and components in a VN are mutually reachable. Thus the name resolution consistency requirements C1 and C2 are satisfied.
Chapter 5

Analysis and Experiments

In this chapter, we analyze the communication complexity of the shadow tree algorithm in Section 5.1, describe a prototype implementation in Section 5.2, and report some experiments conducted on the prototype in Section 5.3.

5.1 Communication Complexity

The complexity of the static algorithm is measured in number of messages. It depends on the name graph configuration, the location that the update is performed in the name graph, the distribution of group nodes in a physical system, and multicast support at group nodes. Since these factors are system-specific, we shall estimate the worst case communication complexity for the algorithm.

5.1.1 Worst Case Assumptions

Consider adding/deleting an arc in a name graph $G = (N, A)$ containing $|N|$ nodes and $|A|$ arcs (a shadow edge is counted once because a message traverses each arc in a shadow tree only once). In the worst case,

1. groups are completely distributed, one group node per subnet, so that every message between group nodes must be counted:
2. multicast is simulated by one-to-one interprocess communication at every node; and

3. the name graph topology is organized as shown in Figure 5.1, where each node has a path from each of its ancestors.

![Figure 5.1: The worst case name graph topology.](image)

We justify the worst case topology as follows. An update can be decomposed into two phases. In *join*, the first phase consists of the collection of topology information of the new VN from each of its components. In *leave*, the first phase consists of the notification of the deleted arc so that each component of the originating VN can reorganize the state of its adjacent arcs and clean up its pathname set. When only a normal arc is added or deleted, the first phase is not needed. The second phase consists of the pathname update relay. In this section, the part of the name graph in which the pathname update relay is to be performed after an update operation is called the *to-be-updated subgraph*. In an update operation, pathname update relay request must travel through every normal arc and shadow edge in the to-be-updated subgraph, because the correctness of pathname update relay is propagation path dependent.

The message cost in the second phase of an update is proportional to the number of arcs in the to-be-updated subgraph, the number of new DLs resulted from the update, and the number of secondary segments in each new DL. Therefore, to justify the worst case topology, we only need to show that any DAG is a subgraph of the graph that has the same number of nodes and the topology shown in Figure 5.1.
We use a greedy construction strategy to show that the topology in Figure 5.1 contains the maximum number of arcs. Let us number the nodes in the graph by 1, 2, \ldots, n where \( n = |N| \). Observe that (a) node \( n \) can have at most \(|N| - 1\) in-arcs since the name graph is a simple graph; and (b) given that there is an arc from every other node to \( n \), it is impossible to have an arc from \( n \) to any of the other nodes, since the graph is a DAG. Recursively applying this construction strategy to the other \(|N| - 1\) nodes results in the topology shown in Figure 5.1 where \(|N| = 5\). Because adding an arc into this graph would either generate a loop or make the graph non-simple, this topology gives the maximum number of arcs in a DAG: \( |A| = |N|(|N| - 1)/2 \).

### 5.1.2 Message Complexity

In the special cases that adding an arc joins two nodes already in the same RL or deleting a nonnormal arc does not break an RL, the second phase of relaying pathname update is not necessary. Suppose the involved VN has \( k \) components, only \( k - 1 \) messages are necessary to inform the other \( k - 1 \) components to update their EGV connection topology of the VN. In the worst case, \( k = |N| \).

With the worst case assumptions described in Section 5.1.1, the two phases of an update in non-special cases are analyzed as follows.

Given a to-be-updated subgraph consisting of \( m \) nodes (where \( 1 < m \leq |N| \)), the message overhead of the second phase consists of two parts:

1. \( OH_{pn-update}(m) \) — the number of messages for pathname update relay in the to-be-updated subgraph: Assuming no RL in that graph, the maximum number of messages in this part is \( m(m - 1)/2 \), since one message per arc is required. This indicates that the maximum overhead is incurred when an update occurs close to the root of the name graph and contain as many nodes in its to-be-updated subgraph as possible.

2. \( OH_{rd-assign}(m) \) — the number of messages for assigning or removing the RD control record at the secondary parents of the sink in each DL: The number of messages in this part depends on the topology of the to-be-updated subgraph. This part of the overhead reaches
the maximum when (i) the to-be-updated subgraph contains the maximum number of DLs, (ii) each DL contains the maximum number of segments, and (iii) the to-be-updated graph does not contain any RL since DLs embedded in an RL need not to be RD controlled. The worst case topology maximizes this part of the overhead to $m(m - 1)/2 - (m - 1)$, as every arc in the graph will be RD controlled except the path from $G_1$ to $G_m$ which contains $m - 1$ arcs.

Next, let us analyze the first phase of join. Consider the graph in Figure 5.2, where $|N| = 5$. If a normal arc is added (i.e., no new RL is formed), only a pathname update is necessary. Let us ignore arc $< G_2, G_1 >$ for the moment. From the above discussion, if arc $< G_1, G_2 >$ is the arc added by the join, the maximum number of messages will be required in the second phase; that is, $1 + OH_{pn-update}(|N| - 1) + OH_{rd-assign}(|N|) = |N|^2 - 3|N| + 3$ since the RD record assignment is performed in the subgraph rooted at $G_1$ which contains $|N|$ nodes, and pathname update is performed in the subgraph rooted at $G_2$ which contains $|N| - 1$ nodes.

Suppose that an RL of $k$ components is formed. Now consider the complete graph in Figure 5.2 including arc $< G_2, G_1 >$. The first phase requires $3(k - 1)$ messages to collect and to distribute the new VN topology information. The number of messages in the second phase can be maximized to $(|N| - k)^2 + 2(k - 1)(|N| - k)$ if the new VN includes the first $k$ nodes (including $G_1$). The first part of this expression is the number of messages required in the

\footnote{Assume that each node in the to-be-updated subgraph does not relay pathname update request to its children}
pathname update relay and the RD control record assignment for a graph of $|N| - k + 1$ nodes with the worst case topology. The second part is the number of additional messages for all but one components in the new VN to send their pathname update requests to the immediate children connected by normal arcs and for RD control record assignments to these arcs.

The total number of required messages when an RL of $k (\geq 2)$ nodes is formed is $3(k - 1) + (|N| - k)^2 + 2(k - 1)(|N| - k)$. By setting its derivative with respect to $k$ to zero, the maximum number of required messages equals to $|N|^2 - 2|N| + 3$ is obtained when $k = 2$ or 3; that is, when arc $< G_2, G_1 >$ or $< G_3, G_1 >$ is added.

In summary, given the worst case topology with $|N|$ nodes, the maximum number of messages required in a join is

\[
Overhead_{join} = \begin{cases} 
|N|^2 - 3|N| + 3 & \text{a normal arc is added,} \\
|N| - 1 & \text{a nonnormal arc is added without creating new VN,} \\
|N|^2 - 2|N| + 3 & \text{a new VN consisting of } k (2 \leq k \leq |N|) \text{ nodes is formed.}
\end{cases}
\]

Finally, let us analyze the first phase of leave. Consider the graph in Figure 5.3, where $|N| = 5$. If a normal arc is deleted, only a pathname update is necessary. Let us ignore arc $< G_{|N|}, G_1 >$ for the moment. From the above discussions, if arc $< G_1, G_2 >$ is the arc deleted until the request is received from all immediate parents in that subgraph. This assumption simplifies the message counting and reduces the total number of messages required.
by the \textit{leave}, the number of messages required in the pathname update relay is maximized to $|N|^2 - 3|N| + 3$ since the to-be-updated subgraph is the subgraph of $G_2$ which contains $|N| - 1$ nodes and has the worst case topology.

Suppose that an RL with $k$ components is broken by the \textit{leave}. Consider the complete graph in Figure 5.3 including arc $< G_{|N|}, G_1 >$. The first phase requires $2(k - 1)$ messages to announce the arc deletion to the components in the originating VN. Obviously, this part of the overhead is maximized if $k = |N|$. The number of messages required in the second phase depends on the topology of the to-be-updated subgraph after the RL is broken. From Figure 5.3 it is easy to see that if $< G_{|N|}, G_1 >$ (here $|N| = 5$) is the arc to be deleted, the pathnames of the whole name graph have to be updated. Therefore, the overhead in this phase is $OH_{\text{pn-update}}(|N|) + OH_{\text{rd-assign}}(|N|) = (|N| - 1)^2$. The maximum number of messages required when an RL is broken by an arc deletion is $2(|N| - 1) + (|N| - 1)^2$.

In summary, given the worst case topology with $|N|$ nodes, the maximum number of messages required in a \textit{leave} is

$$\text{Overhead}_{\text{leave}} = \begin{cases} |N|^2 - 3|N| + 3 & \text{a normal arc is deleted}, \\ |N| - 1 & \text{a nonnormal arc is deleted without destroying VN}, \\ |N|^2 - 1 & \text{a VN consisting of } k \ (2 \leq k \leq |N|) \text{ nodes is broken.} \end{cases}$$

5.1.3 Discussion

Note that although the worst case message complexity of both operations is in the order of $O(|N|^2)$, the average complexity is expected to be much less, because:

- Name graphs tend to be sparse in the real world since unrelated groups usually are not connected. This suggests that an update activity (including its pathname update relay) only involves a small portion of the name graph instead of all $|N|$ nodes.

- When a VN of size $m$ exists in the name graph, only $m - 1$ messages are needed to propagate a pathname update to the components of the VN. This is one order of magnitude smaller than $m(m - 1)/2$ messages when these nodes are not in a VN. Also, there is no need for RD control assignment for DLs embedded in a VN.
• If multicast is properly supported, a node may multicast pathname update relay to its children, reducing the number of messages.

• Multiple group nodes may reside on the same machine. If these nodes are consecutively connected in the name graph, only one intermachine message is needed in updating their pathnames.

• In practice, majority updates occur at the leaf-level in the name graph. Also, the number of secondary parents to be assigned RD control records is typically small.

5.2 Prototype Implementation

A prototype implementation of the internet group model and the static name graph update algorithms has been built. This prototype, serving as an existence proof, is used to estimate the amount of implementation effort and to demonstrate and investigate the model behavior in various situations.

5.2.1 Environment

The prototype is implemented using Threads — a sub-kernel running inside a Unix process [Neufeld et al 90]. Threads supports light-weighted cooperative processes sharing a single Unix process address space and provides efficient interprocess communication between the Threads processes. In Threads, concurrency can be achieved by creating multiple execution threads in a Unix process.

Threads is chosen as the test-bed of the prototype for the following reasons:

• Easy to use. Threads provides a very simple user interface. Connectionless message passing between different Threads processes is supported by synchronous Send(), Receive() and Reply() primitives. Message receivers are simply addressed by their pids. All existing Unix system calls and library routines are available in Threads. The Threads sub-kernel has been well tested and ported on several architectures.
• Easy to debug. Since Threads processes are in a single Unix process, Unix dbx can be used to debug the prototype.

• Easy to move the prototype to a real system since a program written to run in the Threads environment has few differences from one written to run directly under Unix.

5.2.2 Prototype Structure

The internet group name graph model in Chapter 3 and the name graph update operations described in Chapter 4 are implemented. Each group node in the name graph is implemented as a separate Threads process and all nodes in the name graph reside in the same Unix process address space. The arcs between nodes are recorded in an adjacent arc list data structure maintained at each node. A node uses this list when it takes part in name resolution or pathname update relay activities. To be realistic, communication between different nodes are restricted to message passing; that is, instead of passing pointers, messages between Threads processes are copied. A node coordinates a name graph update activity (with other group nodes) when a user request (join or leave) is received.

There is a book-keeping server, implemented as a thread, which keeps track of the user commands and maintains the EGV. It also keeps track of name graph update activities coordinated by group nodes and maintains the IGV. Its purpose is to provide a global view of the name graph. When requested by users during debugging, it can print the adjacency matrices of the IGV and the EGV, and compute reachability matrices in both views for comparison.

There is a user interface thread which is responsible for reading commands from the console and creating and deleting group nodes in the name graph. It dispatches join and leave user requests to group nodes in the name graph, or the user requests for the global state of the name graph to the book-keeping server.

The implementation is in C. The total number of lines of code is about 7,800, including blank lines, comments and debugging routines. Many data structures could have been simplified and many routines with similar functions could have been combined and optimized.
5.2.3 Prototype Limitations

We briefly outline the limitations of the prototype in this section. Removing these limitations is left as future work.

Timing performance refers to the execution time of name graph updates. Since the prototype is implemented within a single Unix process, its timing performance does not reflect the real execution time and message delay in physical networks. To obtain meaningful timing performance, the prototype has to be moved to a real network, its code optimized, and the measured performance data have to be collected from there.

Concurrency is another issue not addressed. Since there is only one user interface thread that reads user requests from the console and dispatches the requests to the group nodes in the name graph, update requests from the user are executed in a sequential order. To execute updates concurrently, the prototype has to be implemented using a multiple Unix process structure (preferably on multiple machines) and concurrency control and failure handling protocols have to be implemented. This will allow users to send their requests directly to the originating groups (rather than having all requests funneled through a single process as in the prototype) to update name graph concurrently. The concurrency control and the failure handling protocols in Chapter 6 are not implemented in this prototype because they are extensions of protocols that are known to work.

5.3 Testing Experiments

We report the testing experiments conducted on the prototype in this Section. Two types of testing experiments were conducted on the prototype: random testing and special-case testing.

5.3.1 Random Testing

The purpose of the random testing is to detect if there is any design flaw in the algorithm or any unexpected bugs in coding the prototype.

During random testing, graduate students were invited to attack the prototype. Group
nodes were added to the name graph and arcs between nodes were added and deleted randomly. After each update, name resolution consistency was verified by invoking name resolutions on group nodes to check if the name graph was correct and if RL and DL effects were properly controlled. Users were supposed to know the EGV graph, but the IGV must be transparent to them. If a message sent to a group node could be received by that node and each of its descendants (according to the EGV) once and only once, name resolution consistency between the IGV and the EGV was achieved and the execution of the update was considered correct.

Except for a few small bugs which were easily fixed, no major error in the algorithms was found. The largest name graph tried by students had more than 20 nodes. Name resolutions were performed as expected according to the EGV and no inconsistency was found. In other words, the IGV name graph transformations in static updates were transparent.

5.3.2 Special-Case Testing

The purpose of the special-case testing is to see whether the internet group model and the static name graph update operations react to various situations as expected. We focused our attention on testing updates that generate, modify or destroy RLs and DLs, because it is in these cases that the static RL/DL control algorithm takes effect to change the IGV.

As in the case of random testing, name resolutions were invoked on nodes in the name graph after each update was completed in order to verify the correctness of the update. The execution of a static update was correct if name resolution consistency between the IGV and the EGV was preserved. Furthermore, the internal state of each node, including its pathname set, its adjacent arc list and the topology information of the VN in which the node was a component, were carefully examined using the Unix dbx.

Appendix B describes details of the experiments conducted on the prototype. These experiments were designed according to the flow of control in the prototype program. They cover all the situations listed below. Although nothing formal can be said about the coverage of the special-case tests, the experiments caused every line of the code to be executed at least once.²

²Each line of code executed during the experimenting process was marked by hand to ensure that all lines
The situations that need to be tested are summarized below:

**Test Cases for Join**

We list the situations which need to be tested for the `join` operation in this section. These test cases include RL generation/modification, DL generation/modification, and combinations of both. Examples are shown in Figure 5.4, where a dotted line represents the arc added by the `join` and circles and shaded areas represent VNs.

1. The trivial case — an arc addition generates no RL/DL. This allows us to check if path-name update relay is performed properly in `join`.

2. RL detections. This can be further divided into the following cases:
   
   (a) The detection of a trivial RL — an RL consisting of two physical nodes only (case J.2.a in Figure 5.4).
   
   (b) The detection of a nontrivial RL whose component list contains physical nodes only (case J.2.b in Figure 5.4).
   
   (c) The detection of a nontrivial RL whose component list contains physical nodes as well as VNs (case J.2.c in Figure 5.4); that is, existing RLs may be embedded in the new RL. This allows us to check if the component list of the VN can be expanded properly during VN construction.
   
   (d) Changing the connection topology of a VN by adding an arc that connects two components in the same VN (case J.2.d in Figure 5.4). The effect of this update should not be seen by nodes outside the VN.

The above were tested in `join` experiments 1 and 2 in Appendix B.

3. DL detections. This can be further divided into the following cases:

were executed.
Figure 5.4: Test cases for the *join* operation.
(a) The detection of a simple DL — a DL containing no VN or embedded DL (case J.3.a in Figure 5.4).

(b) The detection of a DL that contains embedded DLs (case J.3.b in Figure 5.4).

(c) The detection of a DL that is embedded in another DL (case J.3.c in Figure 5.4).

(d) The detection of a DL with a VN source, or a VN sink, or both (case J.3.d in Figure 5.4).

(e) Updating RD control record when ancestors of the DL source are changed.

(f) The effect on RD control when adding an arc connecting two nodes already in the VN source or the VN sink.

(g) The detection and control of parallel arcs between two VNs, or between a physical node and a VN (case J.3.g in Figure 5.4).

The above were tested in join experiments 2 – 8 in Appendix B.

4. The detection of an RL containing embedded DLs (case J.4 in Figure 5.4). This was tested in join experiments 2, 3 and 5 – 8 in Appendix B.

5. The detection of an RL that contains nodes in different segments in the same DL (case J.5 in Figure 5.4). In this case, the DL involved is changed and the resulting new DLs have to be detected and controlled. This was tested in join experiment 5 in Appendix B.

Test Cases for Leave

We list the situations which need to be tested for the leave operation in this section. These test cases include DL modification/destruction, and RL modification/destruction and its effect to DLs. Examples are shown in Figure 5.5, where the deleted arcs are crossed and shaded areas represent VNs.

1. The trivial case — deleting a normal arc. This includes the following cases:
Figure 5.5: Test cases for the *leave* operation.
(a) The deleted arc is not in any RL/DL. This allows us to check if pathname update relay is performed properly in leave.

(b) Deleting the arc removes a secondary segment of a DL (case L.1.b in Figure 5.5).

(c) Deleting the arc removes the primary segment of a DL. This allows us to check:

1. If a new primary can be properly selected when there are more than one secondary segment left in the involved DL (case L.1.c.1 in Figure 5.5).

2. When only one segment is left (case L.1.c.2 in Figure 5.5), if the RD control record for the involved DL is removed at the immediate parent of the sink.

These were tested in leave experiments 1 and 5 in Appendix B.

2. The effect of deleting a shadow edge or a fade arc within a VN without changing any loop deep equal relation (case L.2 in Figure 5.5). Nodes outside the VN should not be aware of the arc deletion, but components of that VN should properly update the VN topology information. This was tested in leave experiment 2 in Appendix B.

3. The deletion of parallel arcs between two VNs or between a physical node and a VN (case L.3 in Figure 5.5). Effectively, this is similar to the case when a segment of a DL is removed. This was tested in leave experiment 3 in Appendix B.

4. The effect of breaking a VN due to an arc deletion. This includes:

(a) Construction of retained VNs (case L.4.a in Figure 5.5).

(b) Re-establishment of the ancestor-descendant relation among all the nodes in the broken VN (case L.4.b in Figure 5.5).

(c) While a DL may be removed when its VN source is broken, new DLs may be formed and should be detected (case L.4.c in Figure 5.5).

(d) While a DL may be removed when its VN sink is broken, new DLs may be formed. The new DLs should be detected and RD control records have to be assigned (case L.4.d in Figure 5.5).
(e) DLs embedded in a broken VN may be re-exposed and therefore should be detected.

These were tested in leave experiments 4 – 7 in Appendix B.

5. When a VN is the sink of a DL and the source of another DL at the same time, breaking the VN may result in both DLs being destroyed and new DLs generated (case L.5 in Figure 5.5). This was tested in leave experiment 7 in Appendix B.

5.4 Chapter Summary

The communication complexity of the shadow tree algorithm is $O(|N|^2)$ in the worst case. In practice, a better average case complexity can be expected. The effort of prototype implementation is reasonable. The testing experiments show that the algorithm is robust and practical.
Chapter 6
Concurrency and Resiliency

Because message delays in networks are inherently unpredictable, the execution result of an update may be affected by the concurrent executions of other updates or by some unexpected node or communication link failures. A practical name graph update protocol has to deal with these situations to produce correct results.

This chapter is organized as follows. In Section 6.1, we discuss the interference between updates and analyze why and how interferences may occur. We also briefly outline different concurrency control policies on name graph updates and their implications. In Section 6.2, an update ordering protocol that serializes concurrent updates is described and its correctness is argued. In Section 6.3, the failure mode is defined and the basic approach to dealing with failures is outlined. In Section 6.4, the update ordering protocol is extended to provide resiliency. Failure handling activities to maintain name resolution consistency are described in Section 6.5 and reliable name resolution protocols are outlined in Section 6.6.

6.1 Concurrency Control Issues

As Table 6.1 shows, for each update operation \( op(x,y) \), we define its originator to be the node that coordinates the update operation; its working arc to be the arc added or deleted by the update in the EGV; and its working set to be the set of nodes along the \( y \rightarrow x \) path(s).
The subgraph rooted at $y$ is called the subgraph of update $op(x,y)$.

<table>
<thead>
<tr>
<th>$op(x,y)$</th>
<th>originator</th>
<th>working arc</th>
<th>working set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$join(x,y)$</td>
<td>$x$</td>
<td>$&lt;x,y&gt;$</td>
<td>$\phi$ if no RL is detected. Otherwise, all components in the new VN.</td>
</tr>
<tr>
<td>$leave(x,y)$</td>
<td>$y$</td>
<td>$&lt;x,y&gt;$</td>
<td>$\phi$ if $&lt;x,y&gt;$ is normal. Otherwise, all components in the originating VN.</td>
</tr>
</tbody>
</table>

Table 6.1: Originator, working arc and working set.

Without loss of generality, assume that there is a name graph manager process per node. The name graph is collectively maintained by all the name graph managers. A name graph manager maintains the following topology information about the name graph:

1. information for name resolutions — the adjacent arc list specifying the type (in or out) and the state (normal, shadow or fade) of the arcs incident at this node, and RD control records for out-going arcs that are RD controlled for some DL;

2. information for RL and DL detections — the pathname set representing the topology of ancestors in the derived name graph, and

3. topology information of the VN in which this node is a component (the adjacency matrix of the VN and the lists of parents/children of the VN defined in Section 4.6.1).

Each update operation takes two steps to complete: (i) RL detection and VN construction, and (ii) pathname update relay (and DL detection). The first step changes the name graph and the second step propagates the effect of the update to descendants. Multiple name graph managers cooperatively take part in an update operation to change the state of the name graph (parts 1–3 listed above) based on their local knowledge about the current state of name graph (parts 2 and 3 listed above).

Local knowledge about the name graph state may be obsolete, however, as changes to the name graph by an update take unpredictable amount of time to propagate. Inconsistent result
could be produced if an update uses obsolete state information. An update \( op_2 \) affects another update \( op_1 \) if the concurrent execution of \( op_2 \) obsoletes the local knowledge of the name graph state used by \( op_1 \). Updates are independent if they do not affect each other when executed concurrently.

Examples

A \( \text{join}(x,y) \) operation uses \( P_x \), the pathname set at the originator \( x \), to check for new RL and obtain its working set. If \( P_x \) has not been properly updated to reflect the fact that other concurrent updates have changed the \( y \rightarrow x \) path(s), \( \text{join}(x,y) \) may make an erroneous decision on RL detection. Figure 6.1 shows some examples:

- An undetected RL is an RL generated by a set of concurrent \( \text{join} \) operations, but is
detected by none of them (Figure 6.1.a).

- An incomplete RL is an RL that does not include all the nodes in the same loop deep equal relation. $\text{Join}(x,y)$ detects an incomplete RL if when it is executed, $P_x$ does not reflect the fact that some concurrent $\text{join}$ operation(s) has introduced new branch(es) into the $y \rightarrow z$ path(s) (Figure 6.1.b).

- A phantom RL is a falsely detected RL that no longer exists.

- An incorrect RL is an RL containing nodes that are not loop deep equal. A $\text{join}(x,y)$ detects an incorrect RL if when it is executed, $P_x$ does not reflect the fact that some concurrent $\text{leave}$ operation(s) has removed some branch(es) from the $y \leftarrow z$ path (Figure 6.1.d). When no $y \leftarrow z$ path is left, a phantom RL is detected (Figure 6.1.c). Case (a) shows that $\text{join}$ operations may affect each other even though their working sets do not overlap (their working sets are $\phi$ in Figure 6.1.a).

The IGV has to be reorganized when a shadow edge or a fade arc is deleted. Instead of using pathnames to determine the retained RLs, a $\text{leave}$ reorganizes the IGV according to the topology information of the originating VN, which is replicated at every component when the VN is formed. The reorganization of the IGV may be incorrect if the knowledge of the originating VN topology information at the originator has not been properly updated to reflect the changes made to the VN by other concurrent updates.

Unlike the $\text{join}$, a $\text{leave}$ is not affected by any other update if it deletes a normal arc, since it does not need any VN topology information for the IGV reorganization.

**Discussion**

The following theorems state the conditions under which updates affect one another.

**Theorem 6.1.1** Given two updates $op_1$ and $op_2$, the concurrent execution of $op_2$ affects $op_1$ if and only if $op_2$ changes the working set of $op_1$.
**Proof:** The if part has been shown by the above examples. We prove the only if part by contradiction as follows. Assume that the working set of \( op_1(x_1, y_1) \) is not changed by \( op_2(x_2, y_2) \). Then working arc \( < x_2, y_2 > \) of \( op_2 \) must have nothing to do with the loop deep equal relation created or modified by \( op_1 \); that is, if \( op_1 \) is a join, adding or deleting \( < x_2, y_2 > \) does not generate, change or break any \( y_1 \rightarrow x_1 \) path; or if \( op_1 \) is a leave, either \( x_2 \) or \( y_2 \) is not in the originating VN of \( op_1 \). Hence, the IGV transformation in \( op_1 \) is independent of \( op_2 \). □

According to the definition, \( op_2 \) does not affect \( op_1 \) if the changes made by \( op_2 \) is known to the originator of \( op_1 \) when it is executed, because in that case their executions are sequential.

**Lemma 6.1.1** A necessary condition for an update \( op_1 \) to affect another concurrent update \( op_2(x, y) \) is for \( op_1 \) to occur in the subgraph of \( y \).

**Proof:** According to the definition, the working set of the update \( op_2(x, y) \) is completely embedded in the subgraph of \( y \). If \( op_1 \) does not occur in the subgraph of \( y \), it cannot change this subgraph, nor is it possible for \( op_1 \) to change the \( y \rightarrow x \) path(s) to affect \( op_2(x, y) \). □

**Theorem 6.1.2** Given two updates \( op_1(x_1, y_1) \) and \( op_2(x_2, y_2) \), their concurrent executions affect each other if and only if either their working sets overlap or they are join operations occurring in the subgraphs of each other.

**Proof:** This theorem states how updates affect each other (as defined by Theorem 6.1.1). It is proven in two cases:

**case 1.** According to the definition, the working set of an update contains all the nodes which are going to have a loop deep equal relation after the update, or which are in the loop deep equal relation to be changed by the update. Updates having overlapping working sets will affect each other if executed concurrently, because they may change the same loop deep equal relation without knowing the changes made by the other. On the other hand, if the working set of the two updates do not overlap and one of the updates is leave, interference cannot occur. Because the working set of the leave is outside the working
set of the other update, deleting the working arc of the leave does not affect the loop
deep equal relation being created or modified by the other concurrent update. The other
update does not affect the leave either since it does not change the working set of the leave
and the correctness of the leave does not depend on the pathname set at its originator.

**case 2.** The only case left to be examined is when the working sets do not overlap and the two
updates are join operations. If they occur in the subgraph of each other as in Figure 6.1.(a)
(where we have paths $y_1 \rightarrow x_2$ and $y_2 \rightarrow x_1$), they affect each other by creating an
undetected RL. On the other hand, if one of the join operations is not in the subgraph of
the other, according to Lemma 6.1.1, it is impossible for it to change the working set of
the other, hence there will be no interference. □

**Concurrency Control Policies**

Maintaining a name graph is similar to maintaining a distributed database. An execution
of a set of updates on a given initial name graph is *serializable* if the resulting name graph is
produced as if the updates were executed on the initial name graph in some sequential order.
Serializable execution results are *consistent* because update executions are *commutable* — the
final resulting name graph is independent of the execution order of the updates.

The goals of concurrency control in distributed systems are to schedule the concurrent
operations to produce a serializable result, and to maximize the parallelism among independent
operations to increase efficiency. Different policies of concurrency control on the name graph
updates and their implications are briefly outlined below:

- *No concurrency control at all:* Because of the potential interferences between concurrent
  updates, name resolutions may observe the intermediate states of updates and the re-
  sulting IGV may be inconsistent. This is not acceptable in general unless updates occur
  infrequently and the inconsistency between the IGV and the EGV can be detected and
  corrected during name resolutions.

- *Concurrency control among name graph updates only:* Although name graph updates
are serialized and the name graph is correct after each update, name resolutions may still observe some transit states of updates. This is acceptable only if applications do not care about temporary inconsistency or can detect and recover from the temporary inconsistency by themselves.

- **Concurrency control on all accesses to the name graph:** Although name resolutions are not commutable with respect to name graph updates, it is acceptable to some applications if the result of a name resolution is produced on the basis of a consistent snapshot of the name graph. A *snapshot* is a state of the name graph at a particular point in time. It is *consistent* if it shows the result of a serializable execution of completed updates.

- **Ordered group communication:** Not only are name graph updates serializable, but name resolutions are also carried out in a *consistent order*; that is, if a message is received before another message at a member process, this order is preserved at all the other members in the same group. Ordered group communication guarantees that members in the same group receive messages in the same order, and thus are synchronized. ABCAST and GBCAST in the ISIS system [Birman and Joseph 87a] are examples.

The degree that concurrency control should be supported in a group communication system depends on the intended applications. Once this is determined, a variety of techniques may be used to implement the policy.

### 6.2 Update Ordering Protocol

In this section, an update ordering protocol for concurrency control is described. This protocol is similar to the ISIS GBCAST [Birman and Joseph 87a]. It schedules an execution order among concurrent updates that may affect each other before their executions start.

To support the protocol, every name graph manager maintains:

- A *priority counter*. A priority is defined as a tuple \(< t, s >\), where \(t\) is the value of the counter and \(s\) is the gid of the node. Priority \(< t, s >\) is lower than priority \(< t', s' >\)
if \( t > t' \), or \( t = t' \) and \( s > s' \). The larger the priority value, the lower the priority is. The priority counter value is always adjusted to be larger than the priority value of any message that the node has ever sent or received. The value of a priority is globally unique because the gid is globally unique.

- A message queue. Messages in the queue are sorted according to their priorities. Message \( m_1 \) is ordered before (closer to the queue head) message \( m_2 \) if \( m_1 \) has a smaller priority value than \( m_2 \). Messages in the queue are marked deliverable or undeliverable.

When an update request is received from a user, it is assigned a globally unique transaction id by the originator. This id is carried by all the messages exchanged among the nodes participating in the update transaction. Messages correspond to the same update transaction if they carry the same transaction id.

6.2.1 Description

The name graph update ordering protocol consists of two rounds:

Round One — Order Determination

The operation originator determines the order of an update \( op(x,y) \) by sending an order request to the subgraph of \( y \), and every node in this subgraph votes a priority on the basis of its local priority counter. The originator takes the highest priority vote as the final priority for the update request. Because in a name graph, a node generally does not know the membership of others, this voting round has to be conducted by relaying messages hop-by-hop through the subgraph of \( y \). Specifically:

1.1 When an update request is received from a user, it is assigned a priority according to the value of the priority counter, marked undeliverable, and appended at the end of the message queue at the originator. The originator then prepares an order request which contains the transaction id and the type (join or leave) of the update, the ids of the two
nodes connected by the working arc (i.e., \( x \) and \( y \)), and the initial priority proposed by the originator. The order request is propagated hop-by-hop in the subgraph of \( y \).

1.2 Upon receiving an order request, a node votes a priority according to the value of its priority counter. The order request is marked undeliverable and is appended at the end of the message queue at the receiving node.

1.3 If the receiving node is a leaf node, it returns its priority vote to the parent from which the order request was received. Otherwise, the receiving node relays the order request with its priority vote to its immediate children and awaits their votes. After all immediate children have responded, the node changes its priority vote to be the highest vote returned from its subgraph and the message queue is re-sorted. This new priority vote is then returned in an acknowledgment to the parent from which the order request was received.\(^1\)

1.4 The order determination round terminates when the priority vote collection terminates at the update originator. At this time, the original update request is assigned the Max\{priorities returned from the subgraph of \( y \)\} and marked deliverable. The message queue at the originator is re-sorted.

Update requests and order requests are control messages. They are exchanged between name graph managers only. In the following, messages exchanged during the second round are called transaction instructions. Transaction instructions carry the transaction priority determined during the first round.

**Round Two — Update Execution**

Once the priority of an update request is determined at its originator, its execution order with respect to the other updates is determined. Its execution cannot start, however, until

\(^1\)If there exists a \( y \rightarrow x \) path, the originator \( x \) may receive the order request for \( op(x,y) \) during the order determination round. In that case, \( x \) performs steps 1.2 and 1.3, but does not put the order request into its message queue since the update request is already there (as determined by the transaction id).
updates ordered before it are completed; that is, until the request reaches the head of the message queue at the originator. Specifically:

2.1 When a deliverable update request \( op(x,y) \) reaches the head of the message queue at the originator, the update is allowed to start. RL detection and IGV reorganization can be performed based on the name graph topology information saved at the originator. The update is performed as if it were invoked in a sequential execution without concern of interference from other concurrent updates.

2.2 When an order request reaches the head of the message queue, it is not processed immediately. As a result, it blocks the queue and messages ordered after it in the queue are prevented from being processed. When a transaction instruction is received, the order request corresponding to the same update is removed. The received instruction is marked deliverable and is inserted into the message queue according to its priority. It will be processed when it reaches the head of the queue.

6.2.2 Discussion

When groups are not nested, the above protocol works exactly as the Isis GBCAST, which has been shown to work correctly. When groups are nested, it is important for all nodes in the subgraph of an update to receive the corresponding order request before executing the update (refer to Section 6.2.3). Since an update \( op(x,y) \) consists of two rounds and the execution round is delayed, problems arise when other updates change the subgraph of \( y \) between the two rounds. We discuss these problems and propose solutions to them in this section. Their correctness is argued in Section 6.2.3. Since we only have two types of update (join and leave) and an update can only occur either inside a VN or outside any VN, we have the following cases to cover in the following discussions:

1. a join operation adds a normal arc;

\(^2\)Note that only the operation originator puts the update request into its message queue, other nodes in the operation subgraph only get an order request during the order determination round.
2. a join operation creates a new VN;

3. a leave operation deletes a normal arc; and

4. a leave operation breaks a VN.

Adding a Normal Arc

A join(u,v) scheduled to be executed between the order determination round and the actual execution of op(x,y) brings the subgraph of v into the subgraph of y if u is in the subgraph of y. When this occurs, some nodes in the subgraph of v may not have learned the ordering of op(x,y) (since they were not reachable from y during the order determination round), therefore, they could have started some concurrent updates that may affect op(x,y). To avoid this situation, the following is included in the execution round of a join when a normal arc is added:

2.3 Before a join(u,v) that adds a normal arc < u, v > terminates, the message queue at u is merged into the queues at all the nodes in the subgraph of v. This can be achieved by piggybacking the message queue at u onto the pathname update relay requests in the execution of join(u,v).

When join(u,v) terminates at a node, the node not only learns of its reachability from u, but also receives every message that its ancestors (including u and the ancestors of u) have received and ordered after the join(u,v) update.

Before a message queue is sent out for other nodes to merge, each update request in the queue is replaced by an order request carrying the same transaction id and priority.\(^3\) When an order request m is to be merged into the message queue at a node, if the queue contains a message carrying the same transaction id as m does, m is ignored. Otherwise, m is inserted into the queue according to its priority to make the ordering of m known to that node.

\(^3\)This ensures that only the operation originator has the update request in its queue. Descendants of the originator only get an order request.
Merging Missing Messages during Leave Pathname Update

Consider a node $k$ in the subgraph of a leave. As discussed in Section 4.7.2, if $k$ is the sink of a DL and if the leave breaks the primary segment of the DL, a new primary segment will be selected. It is possible, however, that some messages received by the nodes in the new primary segment may not have been received by node $k$ (because of a longer propagation delay in the old primary segment). These messages are denoted as missing messages.

When the primary segment of the $DL(s,k)$ is broken by a leave, the following step is included in the leave pathname update executed at sink $k$ to capture the missing messages:

2.4.1 Assume that node $k$ has selected one of its secondary parents in $DL(s,k)$, say $z$, as the new primary parent. It contacts $z$ to delete the RD control record for $DL(s,k)$ and ask for $z$'s message queue. The returned message queue from $z$ is merged into the message queue at $k$ and is piggybacked onto the leave pathname update request that is relayed to the descendants of $k$, so that the descendants can also merge the missing messages in their message queues while relaying the leave pathname update.

If the DL affected by the leave has a VN sink, step 2.4.1 is only executed by the component that has the smallest gid among all the components of the VN sink.

Dropping Irrelevant Messages Resulted from Leave Update

A message $m$ in the message queue at a node $n$ is irrelevant if there is no path from the originator of $m$ to $n$ in the name graph. As Figure 6.2 shows, if a leave($u,v$) is executed in the subgraph of $y$ after the order determination round but before the actual execution of $op(x,y)$, some nodes in the subgraph of $v$ may no longer be part of the subgraph of $y$ after leave($u,v$) completes. These nodes, having received an order request for $op(x,y)$ during the first round of $op(x,y)$, will not receive any transaction instruction for $op(x,y)$. The order request for $op(x,y)$ becomes irrelevant at these nodes and blocks them from processing further updates. Pathname update requests can become irrelevant too. Although such a message does not block the message queue, its execution produces erroneous pathnames of non-existing paths.
Figure 6.2: Generation of irrelevant messages after deleting \( <u,v> \).

The following step is included in the execution round of each update to drop irrelevant messages:

2.4.2 When a message for \( op(x,y) \) reaches the head of the message queue, the node checks if node \( w \), where \( w = y \) if \( op = \text{join} \) or \( w = x \) if \( op = \text{leave} \), is an ancestor according to its current pathname set.\(^4\) If not, this message is deleted from the message queue; otherwise, the node awaits the transaction instruction for \( op(x,y) \) as specified in step 2.2.

While a node is waiting, it has to perform this check after each \textit{leave} pathname update until either a transaction instruction corresponding to the same update arrives or the message becomes irrelevant.

**Forming a New VN**

To ensure the consistency resulted from loop deep equal relations, nodes in a VN must start with the same message queue when the VN is formed and execute the same sequence of operations.

\(^4\)If node \( w \) is a component of a VN, it is not visible in the pathname sets at its descendants. In that case, the descendant needs to ask \( w \) for the vnid of the VN in which \( w \) is a component.
messages during the life time of the VN. To ensure the consistent initial message queue, the following steps are included in the execution round of a join that creates a new VN:

2.5.1 While collecting the local topology information from the nodes in the working set, their message queues are collected and merged as well by the originator. The result is the initial message queue. It is distributed together with the topology information of the new VN to the components.

2.5.2 After submitting the local message queue, a component of the new VN stops propagating messages (if the message does not carry the id of the current update transaction) until it finishes the pathname update relay for the join. Because during this period, the IGV connection between the nodes in the new VN is in a transit state.

2.5.3 The initial message queue at the components of the new VN is piggybacked onto the pathname update relay request for the join. The descendants of the new VN merge this piggybacked queue VN into their queues.

Once the IGV spanning shadow tree for the new VN is constructed, all its components will receive the same set of messages and order them consistently according to the protocol.

**Breaking a VN**

When a leave breaks a VN, messages propagated half way through the VN may see the transit state of the IGV reorganization and their propagation may be incorrect. Solutions to this problem must ensure that VN modifications (in name graph updates) and message propagation through a VN (in name resolutions) are serializable. To achieve this serializability, the following steps are included in the execution round of a leave that breaks a VN:

2.6.1 When an IGV reorganization request is received, a component of the originating VN stops propagating messages until it finishes the pathname update relay for the leave.

2.6.2 The message queues at the components of the originating VN are returned when they confirm to the originator that they have completed the IGV reorganization for the leave.
These queues are merged and the result is piggybacked onto the pathname update request for the leave. Nodes in the subgraph of the leave merge this piggybacked queue into their queues while performing the pathname update relay. Only relevant messages are merged.

Because (i) the piggybacked message queue contains all the messages received by the components in the originating VN before the leave execution starts, (ii) pathname update relay of the leave is launched from the originator, and (iii) after the leave, all the nodes in the originating VN are the descendants of the originator, any message received by the nodes in the originating VN is received by all the nodes in that VN and their descendants. Furthermore, nodes in the same retained VN will start with the same message queue.

### 6.2.3 Correctness

In this section, we argue that the above update ordering protocol serializes concurrent update executions if they may affect each other.

A node participates in an update if the node has to relay the pathname update resulted from the update. A message arrives at a node late if the priority of the message is higher than that of the message being processed by the node.

**Lemma 6.2.1** The ordering protocol guarantees that message queue merges are never late.

**Proof:** During the execution of an update, the messages to be merged are from the queues of the nodes in the working set. Because messages in a queue are processed according to their priorities, messages to be merged always have priorities lower than that of the update being executed. Since in each update, participating nodes merge message queues before the update completes, the merged messages are never late.  

**Lemma 6.2.2** The ordering protocol guarantees that a node receives a relevant order request for an update if it participates in the update.

**Proof:** Consider an update \( op(x, y) \). A node \( n \) participates in \( op(x, y) \) if and only if \( n \) is in the subgraph of \( y \).
CHAPTER 6. CONCURRENCY AND RESILIENCY

- If \( n \) is in the subgraph of \( y \) when the order determination round of \( op(x,y) \) is executed, \( n \) must have received the order request for \( op(x,y) \) since \( n \) has to vote for the priority.

- Otherwise, \( n \) must be brought into the subgraph of \( y \) by a \( join(u,v) \) ordered before \( op(x,y) \) and \( v \) is an ancestor of \( n \). If \( join(u,v) \) adds a normal arc, \( u \) must be in the subgraph of \( y \) and knows the order request for \( op(x,y) \). Step 2.3 ensures that all messages received by \( u \) are also received by the nodes in the subgraph of \( v \). If \( join(u,v) \) generates a VN, one of the nodes in the working set of \( join(u,v) \) must be in the subgraph of \( y \). Steps 2.5.1 — 2.5.3 ensure that messages received by the components of the new VN are also received by the descendants of the VN. Because of lemma 6.2.1, \( n \) will receive the order request for \( op(x,y) \) in both cases before it completes \( join(u,v) \).

- Step 2.4.1 ensures that messages received by the source of a DL are also received by the corresponding sink (and its descendants) even though the DL is changed by a leave during the order request propagation. Steps 2.6.1 and 2.6.2 ensure that messages received by the components of a VN are also received by the descendants of the VN even though the VN is changed by a leave while the order request is propagated half way though.

- If a node is moved out from the subgraph of the update and the corresponding order request becomes irrelevant at the node, step 2.4.2 ensures that that order request will be dropped and message processing at the node is not blocked. \( \square \)

**Theorem 6.2.1** If two concurrent updates may affect each other, their executions can be serialized by using the update ordering protocol.

**Proof:** According to Theorem 6.1.2, it is sufficient to show that this update ordering protocol serializes updates that have overlapping working sets, or join operations that occur in the subgraphs of each other.

Given two concurrent updates \( op_1(x_1, y_1) \) and \( op_2(x_2, y_2) \), where \( op_1, op_2 \in \{join, leave\} \), and assume that \( op_1 \) has a higher priority than \( op_2 \). If the two updates occur in the subgraphs
of each other, there exist the paths \( y_1 \rightarrow x_2 \) and \( y_2 \rightarrow x_3 \). That means there is an order request for \( op_1 \) ordered before the \( op_2 \) update request in the message queue at the originator of \( op_2 \). Update \( op_2 \) cannot start until the order request for \( op_1 \) is removed; that is, until the originator of \( op_2 \) finishes the IGV reorganization and pathname update for \( op_1 \). Therefore, when \( op_2 \) starts execution, the changes made by \( op_1 \) become known to the originator of \( op_2 \) (i.e., reflected in its pathnames).

If the working sets of the two updates overlap, there exists a node \( z \) in the intersection such that \( y_1 \rightarrow z \rightarrow x_2 \) and \( y_2 \rightarrow z \rightarrow x_1 \); that is, they occur in the subgraphs of each other, and the above result holds. □

If the originator of an update is not in the subgraph of another update, according to Lemma 6.2.2, its message processing can never be blocked by the order request of the latter, hence both updates can be executed in parallel. The nodes participating in both updates execute the updates in a consistent order.

### 6.2.4 Message Complexity

The message overhead of the name graph update operations is increased when the ordering protocol is used. This overhead consists of two parts: the order determination overhead in round one and the execution overhead in round two. Given the same worst case assumptions as in Section 5.1, the update execution overhead remains unchanged since all the information required in round two can be piggybacked. The additional overhead for concurrency control is the messages exchanged in the order determination round. It consists of propagating the order request and collecting priority votes in the subgraph of the update. An order request propagation follows the name resolution procedure, the arcs it traversed form a spanning tree in the subgraph of the message originator. In the worst case, every node in the name graph may have to receive the order request. Therefore, the total number of messages required in round one is upper bounded by \( 2(|N| - 1) \), where \(|N|\) is the total number of nodes in the name graph. This part of overhead does not change the message complexity of update operations,

\(^5^\)If \( op_2 = leave \), \( y_1 \rightarrow x_2 \rightarrow y_2 \).
since the worst case message overhead in round two remains in the order of $O(|N|^2)$.

### 6.3 Resiliency Support Issues

In this thesis, the terms site/link are used to refer to the machine/communication link in a physical network and node/arc refer to the logical node/arc in a name graph. Arcs in the name graph are mapped onto paths in the network by the underlying routing algorithm. Site/link failures in the network are reflected as node/arc failures in the name graph. We use the term remaining EGV to refer to the connections in the EGV among the active nodes after a failure.

Node/arc failures need not be hidden from applications. Some applications, such as email distribution lists and news propagations, do not take any action when a node/arc fails. Messages are simply lost or queued until the failed node/arc recovers. Other applications that require continuous communication support, such as teleconferencing, require a reorganization of the IGV connections among active nodes to maintain name resolution consistency with respect to the remaining EGV. This IGV reorganization activity after a failure is called failure handling. A name graph management mechanism is resilient if it has this auto-reorganization capability.

Node/arc failures may affect the result of an ongoing update or a name resolution. An update operation is resilient if it produces a consistent IGV irrespective of the failures occurred during its execution. A name resolution protocol is resilient if it delivers messages based on a consistent name graph snapshot and preserves the required semantics (such as message atomicity and ordering) irrespective of failures. Resilient name graph update protocol, failure handling in the resilient name graph management, and resilient name resolution protocols are the subjects of the rest of this chapter.

**Failure Mode Assumptions**

The following assumptions are made:

- A node $x$ monitors another node $y$ if there is an arc between them in the EGV, or if $x$ is expecting a message/reply from $y$. We assume the underlying system provides
some monitoring mechanism similar to the process alias [Malcolm and Vasudevan 84], watchdog [Ravindran and Chanson 86b], timer [Bernstein et al 88] or the *receive-specific* primitive in the V system [Cheriton 88b].

- A node \( y \) detects that an adjacent arc \( < x, y > \) fails and declares that node \( x \) is *unreachable* from \( y \) when \( x \) crashes or when all the physical communication paths between \( x \) and \( y \) are broken (i.e., \( x \) and \( y \) are in different partitions). A failure is reported to applications, so that users may take certain actions to amend the EGV when the site/link failure recovers and to redo the failed name graph updates or name resolutions.

- Messages from the same source are received in the same order as they were sent. Lower layer protocols are responsible for handling lost and out of sequence messages.

- The update ordering protocol in Section 6.2 is used for concurrency control.

- Node failures are fail-stop [Schlichting and Schneider 83]. Once a node crashes, it loses all information about the name graph, including its adjacent arcs (parent and children lists), its pathname set, the topology information of the VN of which the node is a component, and its message queue.

- Messages to unreachable nodes are dropped. Also nodes that failed and then recover while failure handling is in progress are not allowed to participate in the ongoing failure handling activity. Users are told about the failure and are responsible to reconnect the recovered nodes with other active nodes.

In distributed systems, a node generally cannot distinguish a network partition from a site failure. To simplify failure handling, the nodes that have become unreachable due to network partitions are modeled as if they have failed. This allows us to deal with the two different types of failures in a similar manner. As a result, when network partition occurs, nodes in a partition may forget about nodes in the other partitions.
Basic Approach to Resiliency

Although failures and user initiated updates both remove arcs from the name graph, they are different in a number of ways:

- A failure may simultaneously delete more than one node/arc in the EGV. For example, a node failure removes all arcs incident at the failed node since nodes are fail-stop. Multiple nodes may be cut off from the name graph by a network partition.

- Failures are asynchronous. The changes to the EGV resulted from a failure take effect as soon as the failure occurs. Failure handling cannot be delayed or name graph updates and name resolutions will be affected. On the contrary, user initiated updates are delayed (in the update ordering protocol) in order to be synchronized.

Furthermore, note that:

1. Failure handling activities and user initiated updates are *commutable*.

2. A sequence of consecutive failure handling activities are *combine-able*. Given an initial name graph, deleting failed nodes/arcs one after another in a sequence of consecutive failure handling activities and deleting the same set of nodes/arcs all together in a single failure handling activity should produce the same resulting name graph.

The basic approach to resiliency is simple. To produce a serializable result, a two-phase commitment protocol is integrated into the execution round of each user initiated update that creates/modifies a VN. An update fails and is aborted if any node in its working set or any arc connecting nodes in the working set fails before the update is completed. A failed update is reported to the user so that it can be re-invoked later.\(^6\) The two-phase commitment protocol is also used to guard each failure handling activity that changes a VN. Failure handling is aborted and restarted immediately if further failure occurs before its completion.

\(^6\)Alternatively, an aborted update can be automatically re-scheduled by the system to be executed after the failure handling.
6.4 Name Graph Update Protocol Resiliency

Consider the update ordering protocol in Section 6.2. Failures may occur in both the order determination round and the execution round of an update.

6.4.1 Making Ordering Determination Resilient

To make the order determination round resilient, we have to ensure that irrespective of failures, all nodes in the subgraph of an update operation receive the order request for the update before the update execution starts. This assertion is required in Lemma 6.2.2 to ensure the correctness of the update ordering protocol.

A simple way of dealing with the failures that occur during the order determination round is to report to the originator of the update when a node expecting a priority vote from its children detects that a child has failed. The originator may simply abort the update (by issuing an abort message to the subgraph) and tell the user to re-invoke the update later.

Once the above assertion is established for an update, it is not affected by failures, as failures can only cut nodes away from the subgraph of the update. The order requests for the update at the nodes that are cut away are treated as irrelevant and dropped later (refer to Section 6.2.2).

6.4.2 Making Execution Resilient

The execution round of an update is viewed as an atomic transaction if a VN is to be created or modified. A two-phase commitment protocol is used to guard the transaction against failures. The originator of the update is the transaction coordinator, the other nodes in the to-be-created/modified VN are the participants.

The following steps are performed in a join transaction:

1. The coordinator asks the participants for their local EGV topology information and their message queues.

---

7 The two-phase commitment protocol is not necessary when an update only adds/deletes a normal arc. Such an update is unaffected by failures since it does not require any IGV reorganization.
2. Participants send their local EGV topology information and their message queues to the coordinator. They also stop resolving messages and start to monitor the coordinator until the update transaction commits or aborts.

3. After collecting and computing the topology information of the new VN and merging the collected message queues, the coordinator distributes the result to the participants.

4. Upon receiving the result from the coordinator, a participant saves the topology information of the new VN and the merged message queue, temporarily sets the state of its adjacent arcs based on the spanning shadow tree construction, computes the pathname set of the new VN, and acknowledges a ready-to-commit to the coordinator.

5. Depending on the acknowledgments from the participants, the coordinator issues a commit/abort instruction to the participants.

6. Participants commit or abort the join update transaction according to the instruction received from the coordinator. If abort, whatever result produced by the join is discarded. If commit, the node launches a pathname update relay to its immediate children connected by normal out-going arcs after committing the results obtained in step 4. The order request for the committed update is removed from the message queue also.

Clearly, steps 4–6 constitute a two-phase commitment. Similar steps are taken in a leave transaction:

1. The coordinator informs the participants of the deleted arc.

2. The participants reorganize the IGV and send their message queues to the coordinator.

3. The coordinator merges the collected message queues and starts pathname update relay within the to-be-modified VN. The merged message queue is piggybacked.

4. A participant returns a ready-to-commit after the merged message queue is obtained and its pathname set is updated.

Steps 5 and 6 are the same as in join.
6.4.3 Dealing with Failures during Update Execution

To show the correctness of the update protocol resiliency, detecting and dealing with failures in the execution round are summarized in this section. After an update is aborted, a failure handling activity (discussed in Section 6.5) is initiated (by the system) immediately.

Participant failure between steps 2 and 4

This failure can be detected at step 5 when the coordinator expects acknowledgments from the participants. The coordinator then issues an abort instruction to the participants to cancel the update. As a result, failure handling is ordered before the update.

Coordinator failure between steps 3 and 5

This failure can be detected at step 6 by a participant expecting the final instruction from the coordinator. The participant then consults with other participants for the state of the update. If any reachable participant has committed/aborted the update, it also commits/aborts; otherwise, the update is aborted.\(^8\)

Effectively, if the update is committed, the failure appears to the active components in the new/modified VN as if it had occurred after the update. If the update is aborted, the update request appears as if it were never issued because it does not make sense to add/delete an arc connecting a failed node. In both cases, the resulting name graph is consistent since updates and failure handling activities are commutable.

Participant failure after step 4

This failure cannot be detected by the two-phase commitment protocol. The effect of such a failure, however, is equivalent to the case when the participant fails immediately after completing step 6. Being totally unaware of such a failure, active participants follow the

\(^8\)In a join, the participants can be found from the component list of the new VN obtained at step 4. In a leave, they can be found from the component list of the to-be-modified VN.
instruction from the coordinator to commit/abort the transaction. This failure will be detected by the node monitoring mechanism and will be dealt with after the update completes.

**Discussion**

A coordinator or participant failure before the completion of step 3 can be detected at step 4 (by the participants) or step 5 (by the coordinator). Such a failure results in the update transaction being aborted unilaterally.

A network partition may leave an update being committed in one partition but aborted in another, since the two-phase commitment protocol does not block uncertain participants when the coordinator fails. This result is not harmful, however, since the name graph is consistent within each partition.

The system initiates a failure handling activity after an update is aborted. Until the pathname update relay resulted from the failure handling is completed, nodes in the subgraph of the aborted update cannot participate in any other update or name resolution as they are blocked by the order request for the aborted update. This order request will be considered as irrelevant and removed when the pathname update relay resulted from the failure handling is performed at these nodes.

**6.4.4 Message Complexity**

In this section, we analyze the additional message overhead resulted from the two-phase commitment. Consider an update that creates/modifies a VN of size \(k\), where \(k\) is upper bounded by the size of the name graph \(|N|\). If no failure occurs, the additional message overhead resulted from the two-phase commitment are \((k - 1)\) ready-to-commit messages plus \((k - 1)\) commit/abort messages.\(^9\) If a failure occurs during update execution, participants have to exchange more messages to determine the fate of the affected update. In the worst case, the coordinator fails and each participant has to consult with every other participant before deciding to abort the update, resulting in \((k - 2) \times (k - 1)/2\) messages exchanged among the participants.

\(^9\)Messages exchanged for reachability monitoring are not counted.
remaining \((k - 1)\) participants. More message overhead is required if further failures occur during the consultation.

In summary, making the update ordering protocol resilient requires an additional message overhead which is upper bounded by \(O(|N|^2)\) per failure. The complexity of update operations remains \(O(|N|^2)\).

6.5 Name Graph Resiliency

Failure handling refers to the activity of re-organizing the IGV after each failure to preserve name resolution consistency. Depending on whether the failure changes any RL, different failure handling actions are taken.

6.5.1 Handling Normal Arc Failures

Consider the failure of a normal arc \(< x, y >\). The name graph manager at \(y\) initiates an activity to delete the failed arc as soon as \(x\) is detected unreachable. This system initiated activity to handle a normal arc failure is called \(NA\)-failure-handling. Since only a normal arc \(< x, y >\) is removed, no reorganization in the IGV is required. Instead, a \(NA\)-failure-handling\((x, y)\) request is relayed in the subgraph of \(y\) to inform the descendants to remove the pathnames that contain the unreachable node \(x\).

A \(NA\)-failure-handling cannot be affected by any other updates or concurrent \(NA\)-failure-handling activities since it does not reorganize the IGV. On the other hand, \(NA\)-failure-handling activities and user initiated updates are always serializable, because a user initiated update is aborted if the two-phase commitment protocol in the update execution round detects that a node in the working set becomes unreachable. Due to these reasons, a \(NA\)-failure-handling\((x, y)\) need not be safeguarded by a two-phase commitment protocol or be scheduled by the update ordering protocol. It can be initiated as soon as the failure is detected since only one pass of message propagation in the subgraph of \(y\) is performed. A \(NA\)-failure-handling\((x, y)\) request carries the same kind of information as that in a \(leave(x, y)\) pathname update request, and a node in the subgraph of \(y\) does the same thing as if a \(leave(x, y)\) pathname update request were
received. The node also removes the order requests for the updates aborted due to the failure (refer to Section 6.4.3).

Handling Node Failures

A node failure causes all the arcs incident at the node to fail simultaneously. Handling a node failure is equivalent to handling the concurrent failures of the arcs incident at the failed node. Each normal arc incident from the failed node can be deleted by a separate $NA$-failure-handling (initiated at the immediate child). The following $VN$-failure-handling can be invoked to delete shadow edges or fade arcs since loop deep equal relations in the name graph have changed.

6.5.2 Handling VN Partial Failures

A VN partial failure occurs if some (but not all) components of the VN become unreachable. The activity of handling a VN partial failure is called $VN$-failure-handling. Similar to a user initiated leave that breaks a VN, all the active members of a broken VN in the same partition take part in $VN$-failure-handling to reorganize their connections in the IGV to preserve name resolution consistency with respect to the remaining EGV. The differences are:

- The coordinator and the participants of each $VN$-failure-handling have to be determined, because a VN partial failure may be detected by more than one node.
- Possibly more than one arc will be deleted in a single IGV reorganization activity.
- A $VN$-failure-handling activity does not need an order determination round. It is initiated (by the system) as soon as a VN partial failure is detected.
- A pathname update relay has to be launched from every node whose immediate parent (in the EGV) becomes unreachable due to the failure.
Determining the Coordinator and the Participants

Members in a VN can be ranked by their gid (from high to low). This ranking is called the coordinator-rank (c-rank). When a VN is partitioned, the active member with the highest c-ranking in a partition of the broken VN is chosen as the coordinator for the VN-failure-handling activity. The distributed coordinator election algorithm in [Bernstein et al 88] (page 254) can be used for this purpose.

Once a coordinator is elected, it has to determine the participants of the transaction because more than one node may become unreachable in a VN partial failure. The coordinator sends a probe to each component of the broken VN. The components that acknowledge the probe with their message queues are the participants of the VN-failure-handling transaction.

Reorganizing the IGV and Updating Pathnames

After the participants are determined, the coordinator deletes all inactive members and the arcs incident at the inactive members from the adjacency matrix of the broken VN. The resulting adjacency matrix reflects the remaining EGV connections among the participants. The coordinator then computes the reachability among the participants, assigns a vnid to each retained VN, and merges the message queues collected. It then sends to the participants a VN-failure-handling request which contains a list of inactive members, the member list and the vnid of each retained VN, as well as the merged message queue.

Upon receiving a VN-failure-handling request, a participant $p$ runs the same procedure as in a leave operation, plus:10

- All nodes in the inactive member list and arcs incident at these nodes are deleted from the adjacency matrix of the broken VN.
- If $p$ is in a member list of a retained VN, it adopts the retained VN id assigned by the coordinator.

---

10 If $p$ is waiting for a commit/abort instruction, it continues to wait until the update is committed/aborted. No pathname update relay needs to be performed even though the update may be committed, since the failure obsoletes the pathnames generated by that update.
• Pathname representing paths to nodes in the inactive member list or to the participants from which \( p \) is no longer reachable in the remaining EGV are deleted. Each remaining pathname that represents a path to another participant \( q \) is appended with the \( q \rightarrow p \) path(s) if \( p \) and \( q \) are not loop deep equal and if any \( q \rightarrow p \) path exists in the remaining EGV.

**Discussion**

A \textit{VN-failure-handling} cannot be affected by any other concurrent updates or failure handling activities because they are safeguarded by a two-phase commitment protocol. A user initiated update is aborted if it may be affected by any failure, its effect to the name graph is thereby canceled. A \textit{NA-failure-handling} deals with a failure outside any VN, therefore cannot affect a \textit{VN-failure-handling}. Concurrent \textit{VN-failure-handling} activities never interfere with each other since they reorganize the IGV connections of non-overlapping node sets.

After a failure, a message in the message queue at a participant (or a descendant of a participant) may become irrelevant, because the failure may cut the node off from the subgraph of the message originator. After pathname update relay, a node may use the algorithms in Section 6.2.2 to eliminate irrelevant messages in its queue.

**6.6 Name Resolution Resiliency**

A name resolution is consistent if it is performed on a consistent snapshot of the name graph irrespective of concurrent updates and failures. A snapshot is consistent if it is the result of a serializable execution of name graph updates and failure handling activities. A name resolution protocol is resilient if the semantics of name resolution operation can be preserved irrespective of concurrent changes to the name graph. In group communications, name resolution semantics has two aspects: \textit{delivery order} and \textit{delivery coverage} [Garcia-Molina and Kogan 88, Garcia-Molina and Spauster 91]. Delivery order is specified by some timing requirements in delivering messages; delivery coverage is defined by the ratio of the number of group members
that received the message to the size of the receiving group.

The update protocol in Section 6.4 and the failure handling protocol in Section 6.5 guarantee that the name graph will be consistent eventually. Although transit states of the name graph may be used when no resiliency support is provided by the name resolution protocol, inconsistent results are only temporarily. A name resolution protocol with no resiliency support is simple and cheap as it does not synchronize receivers for message delivery. Some applications may choose to live with it if they can tolerate temporary inconsistencies.

6.6.1 Name Resolution Consistency

Besides resolving messages on a consistent name graph snapshot, further ordering requirements may be specified to meet the requirements from various applications. For example:

- Members in the same group have to receive messages in a consistent order. This requires name resolutions to be serializable among themselves [Birman and Joseph 87a].

- The message delivery order must be consistent with respect to some causal relation defined by applications [Birman and Joseph 87b].

Supporting name resolution consistency (and other ordering requirements) requires not only a serializable execution of updates and failure handling activities (to produce consistent name graph snapshots), but also a serializable execution of name resolutions with respect to changes to the name graph. A synchronization mechanism has to be built into the name resolution protocol to ensure that *a node does not deliver a message to the application unless it is assured that all the active receiving members will deliver the message in a consistent order.*

Assuming the update ordering protocol in Section 6.2 is used, two protocols that support name resolution consistency are outlined below.

**Group Datagram Service**

In the internet environment, group datagram service is an approximation to the multicast service such as that provided by the V system. When resolving a message, a node checks if its
message queue is empty. If this is the case, the message is resolved; otherwise, the message is dropped. Because the queue only holds update requests and order requests, an empty queue indicates that this node is not involved in any update and its adjacent arcs are part of a consistent name graph snapshot.

This approximation is reasonable when the name graph does not change frequently. The probability that a message is not delivered to an active member of the receiving group depends not only on the probability of machine and communication link failures or network congestion, but also on the frequency of name graph updates. This approximation tends to drop more messages than necessary, because a message may be dropped even though the node is only involved in some pathname update relay.

**Ordered Name Resolution Protocol**

The same two-round ordering strategy in Section 6.2 can be used to order name resolutions. As a result, name resolutions are ordered with respect to name graph updates (so that messages are delivered based on consistent name graph snapshots) as well as among themselves (so that messages from different sources to overlapping groups are delivered in a consistent order).

**Discussion**

Group datagram service and ordered name resolution protocol ensure that name resolutions are ordered with respect to name graph updates. A simple way to order name resolutions with respect to failures is to abort name resolutions affected by failures. User messages are discarded at the nodes (and their descendants) involved in failure handling activities and the failure is reported. The application is expected to deal with message loss (e.g., retransmit the message) and duplications generated from retransmissions.

**6.6.2 Name Resolution Atomicity**

Atomicity support addresses the delivery coverage aspect of name resolutions. With atomicity support, not only are name resolution results consistent, but message deliveries are also
atomic with respect to failures: that is, a group message is received either by all the active members or by none of them in failure situations. By "exaggerating" a partial failure (occurred during a message transmission) as a total failure, atomicity turns a group into a single entity and leaves applications with a simple failure mode to deal with. This is particularly desirable in distributed database systems [Lampson 83a].

An atomic name resolution protocol has to guarantee that a receiving node does not deliver a message to the application until it is assured that all the other active members will deliver the message. To reach a consistent decision on message delivery when the originator of a message fails, the complete membership of the destination group is often required to allow active members in the receiving group to consult with each other.

Reaching consistent decision across network partitions is difficult. When communication failure (such as network partition) is possible, every atomic commitment protocol may result in blocking [Bernstein et al 88]. Because our goal is to provide continuous communication support within each partition, we have to either restrict the failure mode (e.g., assuming network partition never occurs) or compromise the definition of atomicity (e.g., only guarantee the consistency and atomicity within each partition). Users requiring consistent message delivery across partitions have to either use blocking protocols or build their own consistency assurance mechanism at the application level. In that case, the goal and approaches of failure handling and resiliency support for name graph updates and name resolutions are different from those discussed in this chapter. These topics are out of the scope of this thesis.

Even though it is assumed that communication failures never occur, one has to assume that all information at failed nodes are lost. This may not be realistic because messages may have been delivered to a human being. But blocking may occur without this assumption, since until the failed node comes back, the active nodes cannot tell if the failed node has delivered the message.

The ordered name resolution protocol in Section 6.6.1 does not provide atomicity because after a node learns the final priority of a message, it is still not sure whether other members in the receiving group have received the final priority and will deliver the message. When a
failure occurs, a member cannot consult with the other members because the membership of the receiving group is unknown. A third phase can be added to the protocol in Section 6.6.1 to achieve atomic delivery (with the above failure mode assumptions) as follows:

**Phase One:** The user message is propagated hop-by-hop to the nodes in the subgraph of the message originator and priority votes from these nodes are propagated hop-by-hop back to the originator.

**Phase Two:** After collecting priority votes from the descendants, the originator propagates the final priority hop-by-hop in its subgraph. Upon receiving the final priority, a node reorders the user message in its message queue accordingly, marks the message *to-be-committed* and relays the final priority to its descendants. After collecting *ready-to-commit* acknowledgments from all immediate children, the node returns a *ready-to-commit* acknowledgment to the parent from which the final priority was received.

**Phase Three:** After the originator has finished collecting *ready-to-commit* acknowledgments from all immediate children, it knows that all nodes in its subgraph have received the final priority and are ready to commit to delivery of the user message. The originator then propagates a *commit* instruction hop-by-hop in its subgraph. A node in the subgraph marks the user message *deliverable* when the *commit* instruction is received. A *deliverable* user message is delivered to the application when it reaches the head of the message queue and remains relevant.

Note that the third phase is necessary because the *commit* instruction announces that all nodes in the destination group have confirmed reception of the final priority of the message and are ready to deliver the message.

The failure of a group member before returning a *ready-to-commit* acknowledgment can be detected by its immediate parent and can be reported to the originator. The originator then issues an *abort* instruction in the third phase and reports the failure to the sender. The message is dropped by the descendants when the *abort* instruction is received.
The failure of a node (including the originator) before completing the forwarding of the commit/abort instruction in the third phase can be detected by its immediate children expecting the instruction. Handling such a failure requires knowledge of the complete membership of the message destination group. The member list of the destination group could be gathered when the priority votes were collected in the first phase. That is, together with its priority vote, a node also returns its member list. The returned member list includes the node itself and the union of the member lists returned from its immediate children. When the final priority is determined at the originator, the complete membership of the destination group is also obtained. This member list may be piggybacked onto the final priority message in the second phase, so that when a node sets the user message state to to-be-committed, the complete membership of the destination group of the message is also ready at the node. With this list, the node is able to consult with other active members in the receiving group for the delivery status of the message when a parent fails. If any active member in the receiving group has received a commit instruction from the originator, the other receiving members also commit and deliver that message. If the user message status at all active members is to-be-committed, the message is dropped. The latter case occurs when the originator fails before the third phase starts.

### 6.7 Related Work

Synchronizing group membership updates and group messages is not a new subject. The protocol in [Chang and Maxemchuk 84a] orders group messages through a single site in a broadcast type network. The ordered group communication protocol in [Navaratnam et al 88] uses similar concept. These protocols are not suitable for the internet because the internet is not broadcast oriented.

A propagation protocol is proposed in [Garcia-Molina and Spauster 91] for ordering group messages in the internet environment. Groups are structured into a tree called a propagation graph. There is a primary site for each group and a unique path from the primary site to each member. Messages to a group are submitted to the primary site and are routed to other
members through intermediate sites located in the group intersections. Messages are ordered along the way by merging messages to different groups at the sites in the group intersections. All messages eventually end up at their destinations already ordered. It is the requirements of the propagation graph that limits the applicability of this propagation protocol to the nested group model. The propagation path of a message to a nested group is determined by the topology of the name graph, which may have an arbitrary topology. Also, messages to nodes in a VN are not necessarily routed through the same path. Therefore, messages cannot be ordered consistently by simply adopting the propagation protocol. Furthermore, without complete knowledge of group membership, it is very difficult to construct a propagation graph.

In Isis [Birman and Joseph 87a], a set of protocols for synchronized group communications were proposed. This protocol suite was designed for non-hierarchical groups. We have adopted the basic idea of the Isis protocol in the update ordering protocol and the atomic name resolution protocol. But several aspects of the nested group model make our extension non-trivial:

- In Isis, groups are structured as one-level trees and the membership of the group is known to every member. In the internet environment, the structure of a nested group can be arbitrary and the complete membership of a group are usually not known even to group members. As a result, messages to nested groups must take a set of multi-hop resolution paths.

- Because of the multi-hop communication topology, dynamic changes to the name graph complicates the synchronization in serializing updates and name resolutions.

Two-phase commitment protocols and three-phase nonblocking protocols have been well studied in the literature. Ideas, arguments and techniques from [Bernstein et al 88] are applied to our problem setting in this chapter.

6.8 Chapter Summary

This chapter deals with the concurrency control and resiliency aspects in name graph updates and name resolutions. Concurrency control and update/message ordering are achieved
by a two-round priority voting; failure handling and update resiliency are achieved by using a two-phase commitment protocol.

Updates affect each other if their working sets overlap, or if they are join operations that occur in each other’s subgraph directly or indirectly. In the update ordering protocol described in this chapter, updates with overlapping subgraphs are ordered before their execution start. Updates are allowed to be executed in parallel as long as they do not affect each other. It is the arbitrary topology of name graph and the incomplete group membership information that make synchronization among group members more complicated than existing group communication protocols.

The goal of resiliency support is to provide correct group communication continuously among active nodes based on the remaining EGV. A two-phase commitment protocol is embedded in the execution round of updates (or failure handling) that change the loop deep equal relations in the name graph. An update is aborted when it is affected by failures during the execution. This is acceptable because updates and failure handling activities are commutable.

Different degrees of resiliency support in name resolutions are briefly outlined in this chapter. It is argued that with the update protocol and failure handling protocol discussed earlier, name resolution inconsistency is only temporary even if the name resolution protocol does not provide any resiliency support. More synchronization overhead is involved when a higher degree of consistency and resiliency is required. Due to the lack of complete membership information, atomic group communication in the internet environment is difficult and expensive (requiring a three-phase protocol). Various assumptions concerning failure modes and membership information have to be made in designing atomic group communication protocols. When a network partition is possible and consistent delivery across partition is required, group communication may be blocked.

Our goal was not to provide a reliable name graph update protocol (or a reliable name resolution protocol) that always successfully carries out user requests irrespective of failures. Instead, operation failures are reported to applications so that users can take the appropriate actions.
Chapter 7

Conclusions and Future Research

We conclude the thesis by summarizing the major results and listing some future research.

7.1 Summary of Results

This thesis has two parts. The first part provides a comprehensive classification of process groups; the second part focuses on group name management in the internet environment.

Classifications of Process Groups

To design distributed systems that support process group and group communication, it is important to understand how these mechanisms are used and what are the communication requirements expected from the applications.

For the purpose of classification, a group is defined as a set of cooperative processes that maintain objects to provide a service. Based on the homogeneity of the internal structure of group members (i.e., the state of the objects that they maintain and the operations that they perform on the objects), groups are classified into four categories: data and operation homogeneous, operation homogeneous only, data homogeneous only and heterogeneous.

Based on their external behavior, groups are classified into deterministic and nondeterministic categories. The main difference between the two is the degree of grouping transparency in
communication, reply handling, naming, and failure handling. Deterministic groups hide their internal structure (e.g., membership) from applications as much as possible and do most of the coordination work (e.g., membership change, atomic/ordered group message delivery) at the system level, so that users are provided with a simple interface to groups. Nondeterministic groups provide users with the flexibility of not having to pay the overhead of intrinsic coordination among group members when it is not needed, so that interacting with groups is more efficient (but may also be more complicated). The major concerns in deciding which one is more suitable for a certain application are transparency, efficiency and flexibility.

Internet Group Naming Models

In the internet, subnets/subdomains are connected by internet links that have relatively low bandwidth and long delay. The nested group model is proposed to reduce the traffic on internet links and to maintain autonomy in subnets/subdomains.

The nested group model allows group members in a subnet/subdomain to form a group. It also allows a group to be included in another group as a member. Organizing internet groups in this structured manner hides membership changes within a subnet/subdomain from observation outside the subnet/subdomain. Also, only one message across an internet link needs to be sent to all members of a subgroup.

Two formal models for the nested naming structure are developed: name graph model and name grammar model. With the name graph model, problems such as name resolution loops (RLs) and duplication loops (DLs) are recognized, and loop deep equal relationship is identified as an intrinsic characteristic of RLs. Correctness criteria for the algorithms solving these problems are formulated in Sections 3.2.3 and 3.2.4. Various approaches to handling RLs and DLs are surveyed in Section 3.3. It turns out that none of the existing approaches meets the correctness criteria.

Static Approach to Handling RLs and DLs

A static algorithm maintains a system view (IGV) of the name graph and imposes control
structure in the IGV to avoid the RL and DL effects (in a user transparent manner). The topology of the IGV may be different from that of the users' view (EGV). Update operations must preserve name resolution consistency between the IGV and the EGV.

The virtual node (VN) concept characterizes the loop deep equal relation among nodes in an RL. It defines the amount of global knowledge of the name graph that a node needs to maintain for static name graph update operations to function correctly. Nodes within a VN are connected by a spanning shadow tree in the IGV. A new name resolution procedure is designed to guarantee that a message arriving at a node in a shadow tree will be resolved to every other tree node at most once so that the RL effect is removed. DLs are controlled by assigning RD control record to the secondary parents of the DL sink. A message is not resolved to a child if the arc leading to the child is RD controlled and if the message is routed through the DL source. To detect RLs and DLs, a pathname representation is proposed for each node to record the topology of its ancestors in the derived name graph.

The shadow tree algorithm not only provides a correct solution handling RLs and DLs, but also reduces the run-time overhead in name resolutions. The worst case message overhead of update operations in the shadow tree algorithm is bounded by \( O(|N|^2) \) where \( |N| \) is the total number of nodes in the name graph. A prototype of the shadow tree algorithm has been implemented as an existence proof of the algorithm.

To complete the model, concurrency control and failure handling protocols for static name graph updates are designed. Name graph updates that share nodes in their subgraphs are ordered using a two-phase update ordering protocol before they start to execute. Synchronized name graph update operations are safeguarded against asynchronous failures by using the well-known two-phase commitment protocol in their executions, so that an update is either completed correctly or aborted when failure occurs during execution. Various name resolution protocols supporting ordered group communication, consistent group datagram, and atomic group communication are briefly outlined. As a result, the communication, naming and failure handling transparency requirements for deterministic groups are met (refer to Section 2.3.1).1

1The transparency requirements in reply handling and real-time were not considered in this exercise.
Conclusions

The following conclusions can be drawn from this thesis:

- Process groups can be classified based on their internal structure. They can also be classified based on their external behavior depending on the degree of grouping transparency expected from their intended applications.

- Nested group model can be used to reduce group communication traffic on internet links and support subnet/subdomain autonomy.

- RL and DL effects can be controlled using distributed static approach. The shadow tree algorithm is an example.

- The shadow tree algorithm completely removes RL and DL effects. It provides name resolution consistency and is transparent to users.

- The communication complexity of the update operations in the shadow tree algorithm has an upper bound and the average case complexity can be expected to be much better. The algorithm can be implemented with a reasonable effort and is robust as demonstrated by the prototype experiments.

- The design of concurrency control and failure handling protocols shows that although atomicity and ordering are achievable in internet group communications, hiding membership of subgroups makes them difficult. The protocol can be complex and its overhead can be high. Trade-off has to be made in transparency (autonomy), reliability and efficiency based on the intended applications.

7.2 Limitations

A few assumptions made in this thesis limit the applicability of the static algorithms:

- The static approach assumes that updates occur much less frequently than name resolutions and trades higher update overhead for lower name resolution overhead to obtain a
better overall group communication performance. When group memberships are highly
dynamic in some applications, this assumption no longer valid and the performance of
the static approach may deteriorate. The reason is that a static update blocks name res­
olutions while an update is performed (otherwise, the result may be inconsistent). Also
a number of messages are exchanged during an update, increasing the network traffic.

- The shadow tree algorithm assumes that users do not care about the order that messages
are resolved at the nodes within a destination group. Certain applications structure groups
according to some administration hierarchy and require messages always be passed from
the top-level down. The shadow tree algorithm is not suitable in this case.

Even though the assumption of low bandwidth internet link will no longer be true when
high-speed optical links become widely available, the nested group model for internet groups
remains valuable because (i) it reduces the traffic in internet data highway and supports sub­
net/subdomain autonomy, and (ii) it also reduces the group message distribution time since
only a single message is sent to members in a subgroup. Furthermore, compared to the dynamic
approach, the static approach reduces the overhead of name resolution, hence, is preferred in
high-speed networks where node processing speed is the bottleneck.

7.3 Future Research

There are a number of areas that future research may lead to useful results:

- A systematic study focused on the implications and applications of the name grammar
model proposed in Appendix A may lead to some new insight to the naming problem.

- An optimistic approach to concurrency control and failure handling is to start execution
of an update when it is invoked. Before committing the update, a certification procedure
is conducted to verify that the update is not affected by other concurrent updates or
failures. The update is aborted if the certification fails. This approach is preferred than
the conservative approach in Chapter 6 if concurrent updates and failures do not occur
frequently, since the overhead involved in each update may be made small when it is not affected. The design and analysis of optimistic name graph update operations is an interesting future research direction.

- The name resolution protocols for atomic group communication, ordered group communication and group datagram in this thesis either require a large amount of overhead in message deliveries or provide poor delivery coverage when updates or failures occur. It is desirable to design new name resolution protocols that support the required communication requirements (atomicity and ordering) with as small an overhead as possible.

- We have assumed that update failures and message transmission failures are reported to users and handled at the application layer. It will be interesting to examine how communication failures are actually handled in various applications, and try to build some mechanism at the system level to ease this task. Furthermore, to support applications that always require consistent message delivery (such as distributed database transactions), group message transmission needs to be blocked when network partition occurs. To support this type of applications, recovery algorithms need to be designed so that the EGV and the IGV can be restored after the failed components have recovered and the blocked messages can be delivered consistently.

- To improve the overall system performance, the static approach trades the expense of less frequent update operations for a cheaper but much more frequent name resolution operation. For applications where group membership is more dynamic, it is desirable to have some quantitative methods to determine to what extent the above trade-off is worthwhile. For highly dynamic groups, a new group management approach with smaller update latency and less message overhead will be needed.

- In the case that users do care about the order of name resolutions at the group members, the name resolution procedure needs to be redesigned to take this into consideration while preserving name resolution consistency.
Bibliography


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH


Chapter 7. Conclusions and Future Research


Appendix A

Group Name Grammar Model

Group naming structure can also be characterized by a grammar. For each top-level group $S$, there exists a grammar $L_S = (V, T, P, S)$, where $V$ is a finite set of gids (non-terminals), and $T$ is a finite set of pids (terminals). $P$ is a finite set of production rules, representing the name expansion from a gid to all of its immediate children. The name resolution for a group, say $S$, returns a set of pids that corresponds to a string derived from the starting symbol $S$ by applying the production rules recursively. It should be noted that a name expansion grammar is context free (CFG); that is, the expansion of any gid does not depend on its expansion context. Also note that in the context of group name resolution, the order of symbols in the right hand side of every production rule is immaterial. Therefore, the CFG $L_S$ of the group $S$ can be written as a right-linear grammar; that is, $L_S$ can be a represented by a regular grammar. Because regular sets are closed under union operation [Hopcroft and Ullman 79], the grammar corresponding to the global naming system (which is the union of all top-level groups) is regular.

Given a name grammar $L = (V, T, P, S)$, a name grammar automata $D = (V, T')$ can be generated. Assuming $L$ has been written in Chomsky Normal Form (CNF), an arc $(A \xrightarrow{x} B) \in T'$ if and only if $(A \xrightarrow{x} B) \in P$, where $x \in T$. Since $L$ is a regular grammar, $D$ is a finite-state machine (FSM).

In a grammar representation, a name expansion corresponds to a sentence derivation of

---

$^1$ Although the number of processes and the number of groups in a distributed system vary in time, they are finite at any particular time instance.
the grammar, and group membership update corresponds to adding or deleting production rules. It is the latter that makes the model distinct from general regular language grammar manipulations.

It is easy to see that the group name graph model is equivalent to the group name grammar model; that is, for each name graph, there exists a name grammar, such that a set of processes can be resolved from a group name in the graph if and only if the same set of processes can be derived from the grammar. The argument for this statement is trivial if one notes the fact that a production rule

$$G_i \Rightarrow G_{i_1}G_{i_2} \cdots G_{i_k}p_1 \cdots p_i,$$

exists in $P$ if and only if the group $G_i$ contains $G_{i_1}, G_{i_2}, \cdots, G_{i_k}, p_1, \cdots, p_i$ as its immediate children in the corresponding name graph, where $G_{i_1}, \cdots, G_{i_k} \in V$ and $p_1, \cdots, p_i \in T$.

This name grammar model exposes another dimension of research — is there any case in practice that the grammar is not CFG? if yes, what are the implications and issues? New insight may well be obtained by exploring this dimension. However, we leave this as future research since it is out of the scope of this thesis.
Appendix B

Prototype Experiments

This appendix describes the experiments conducted on the prototype implementation. There are eight experiments for join and seven for leave. All the graphs given in this appendix are EGV graphs. The state of each arc in the IGV is highlighted by a different line type — a thick line represents a shadow edge, a dotted line represents a fade arc, and a thin line represents a normal arc. User requests are numbered in the order that they are executed. After each update, name resolutions are run at nodes in the name graph to verify name resolution consistency between the IGV and the EGV. These name resolution requests are not shown in the diagrams. The adjacency matrices of the EGV and the IGV were printed by the prototype and the diagrams were drawn (by hand) directly from the output of the prototype. In the following, the labels in square brackets indicate the special cases tested in each experiment (refer to Figure 5.4 and 5.5).

Join Experiments

1. Join experiment 1 in Figure B.1 is used to test VN construction, including:

   (i) the construction of a trivial VN ([J.2.a] at step 7),

   (ii) the construction of a nontrivial VN whose component list contains physical nodes only ([J.2.b] at step 3),
(iii) VN topology modifications when adding an arc that connects two nodes in the same VN ([J.2.d] at step 4), and

(iv) the construction of a nontrivial VN whose component list contains both physical nodes and VNs ([J.2.c] at step 9).

1. join (a, b)
2. join (b, c)
3. join (c, a)
4. join (a, c)
5. join (e, c)
6. join (d, f)
7. join (f, d)
8. join (b, d)
9. join (f, e)

Figure B.1: Join experiment 1.

2. Join experiment 2 in Figure B.2 is used to test DL detection and control, including:

(i) the detection of a simple DL that contains no embedded DL or VN ([J.3.a] at step 6, \( DL(b, d) \) is generated and detected),

(ii) the detection of DLs that share some segments ([J.3.b and J.3.c] at step 7, \( DL(a, f) \) and \( DL(a, d) \) that share some nodes with \( DL(b, d) \) are generated and detected),

(iii) the generation of a VN source ([J.3.d] at step 9) or a VN sink ([J.3.d] at step 13),

(iv) the construction of a VN that contains embedded DLs ([J.4] at step 14, \( DL(a, d) \), \( DL(b, d) \) and \( DL(a, f) \) are embedded in the new RL),

(v) the detection of parallel arcs between VNs ([J.3.g] at step 16), and
(vi) the effect of update to the ancestors of the source on the RD control records ([J.3.e] at steps 8 – 10 and 12).

1. join (a, b) 10. join (u, x)
2. join (b, c) 11. join (u, v)
3. join (c, d) 12. join (v, u)
4. join (e, f) 13. join (d, f)
5. join (f, d) 14. join (d, a)
6. join (b, f) 15. join (c, e)
7. join (a, e) 16. join (v, e)
8. join (x, a)
9. join (a, x)

Figure B.2: Join experiment 2.

3. Join experiment 3 in Figure B.3 is used to test:

(i) the detection of parallel arcs between two VNs ([J.3.g] at step 8),
(ii) the effect of topology change within the VN source/sink on a DL consisting of parallel arcs ([J.2.d and J.3.f] at steps 9 and 10), and
(iii) the detection of an RL that contains an embedded DL having parallel arcs between two VNs ([J.2.c and J.4] at step 11).

4. Join experiment 4 in Figure B.4 is used to test:

(i) the effect of changing a physical DL sink to a VN sink ([J.3.d] at step 4).
(ii) the detection of parallel arcs between a physical source node and a VN sink ([J.3.g] at step 5),
Join experiment 3.

(iii) the detection of a DL that has a VN source and a physical sink node ([J.3.g] at step 12), and

(iv) the detection of parallel arcs between two VNs ([J.3.g] at step 13).

In case (i), the DL is replaced by a new DL with a VN sink and the RD control record is re-assigned.

5. Join experiment 5 in Figure B.5 is used to test:

(i) the construction of a VN that contains nodes in different segments of a DL and its effect on DLs ([J.5] at step 8),

(ii) the effect of topology change internal to a VN that is the sink of a DL and the source of another DL ([J.3.f] at step 9), and

(iii) the detection an RL that contains embedded DLs ([J.4] at step 10).
1. join (a, c)
2. join (b, d)
3. join (c, d)
4. join (d, c)
5. join (a, d)
6. join (e, f)
7. join (f, e)
8. join (c, e)
9. join (f, g)
10. join (g, h)
11. join (h, i)
12. join (e, h)
13. join (d, f)

Figure B.4: Join experiment 4.
APPENDIX B. PROTOTYPE EXPERIMENTS

1. join (a, b)
2. join (b, c)
3. join (c, d)
4. join (e, f)
5. join (f, d)
6. join (b, f)
7. join (a, e)

8. join (f, b)

9. join (f, e)

10. join (d, b)

Figure B.5: Join experiment 5.
6. *Join* experiment 6 in Figure B.6 is used to test the detection of parallel arcs leading to the same component in a VN sink ([J.3.g] at steps 8 and 9), and *Join* experiment 7 in Figure B.7 is used to test the detection of parallel arcs leading to different components in a VN sink ([J.3.g] at steps 8 and 9). Both experiments also detect RLs containing embedded DLs ([J.4] at step 10).

\[
\begin{align*}
&1. \text{join } (a, b) \\
&2. \text{join } (b, c) \\
&3. \text{join } (c, a) \\
&4. \text{join } (d, e) \\
&5. \text{join } (e, f) \\
&6. \text{join } (f, d) \\
&7. \text{join } (a, d) \\
&8. \text{join } (b, d) \\
&9. \text{join } (c, d) \\
&10. \text{join } (f, a)
\end{align*}
\]

Figure B.6: *Join* experiment 6.

7. *Join* experiment 8 in Figure B.8 is used to test:

(i) the detection of a DL containing a VN source and a VN sink ([J.3.d] at step 8),

(ii) the detection of parallel arcs in a segment of a DL and the RD control assignment for the new embedded DL ([J.3.c] at step 9), and

(iii) the detection of an RL that contains embedded DLs ([J.4] at step 10).

**Leave Experiments**

1. *Leave* experiment 1 in Figure B.9 is used to test:
1. join (a, b)
2. join (b, c)
3. join (c, a)
4. join (d, e)
5. join (e, f)
6. join (f, d)
7. join (a, d)
8. join (b, e)
9. join (c, f)
10. join (f, a)

Figure B.7: Join experiment 7.

1. join (a, b)
2. join (b, c)
3. join (c, a)
4. join (d, e)
5. join (e, f)
6. join (f, e)
7. join (a, d)
8. join (b, e)
9. join (c, f)
10. join (f, a)

Figure B.8: Join experiment 8.
(i) a normal arc deletion without breaking any RL or removing any segment of a DL ([L.1.a] at step 9),

(ii) the removal of a secondary segment when a normal arc is deleted ([L.1.b] at step 11), and

(iii) the removal of the primary segment due to a normal arc deletion, and the selection of a new primary segment for this DL ([L.1.c] at step 14).

Figure B.9: Leave experiment 1.

2. Leave experiment 2 in Figure B.10 is used to test the effect of deleting VN internal arcs without changing the loop deep equal relation among all the components ([L.2]). The deletions of fade arcs are tested at steps 7, 8 and 9. The deletions of shadow edges (with the same initial name graph) are tested at steps 7, 8, and 9.

3. Leave experiment 3 in Figure B.11 is used to test parallel arc deletions:

   (i) the deletion of a parallel arc between a physical source node and a VN sink ([L.3] at step 11).
1. join (a, b)  
2. join (b, c)  
3. join (c, a)  
4. join (a, c)  
5. join (c, b)  
6. join (b, a)  
7. leave (a, c)  
8. leave (c, b)  
9. leave (b, a)  

Figure B.10: Leave experiment 2.

(ii) the deletion of a parallel arc between two VNs ([L.3] at step 12), and
(iii) the deletion of a parallel arc between a VN source and a physical sink node ([L.3] at step 13).

4. Leave experiment 4 in Figure B.12 is used to test VN breaking:

(i) braking a VN completely so that no retained VN needs to be constructed ([L.4.b] at steps 14 and 15), and
(ii) constructing retained VNs after a VN is broken ([L.4.a] at step 13). Re-establishing ancestor-descendant relationship among all nodes (including retained VNs) in the broken VN is also tested ([L.4.b]).

5. Leave experiment 5 in Figure B.13 is used to test the effect of VN breaking on DLs:

(i) breaking a VN does not break any segment of a DL and all the nodes in that VN remains in the segment of the DL (at step 13), and
(ii) breaking a VN also breaks a DL ([L.1.c and L.4.c] at step 15).

6. Leave experiment 6 in Figure B.14 is used to test the effect on a DL when its VN source or VN sink is broken:
1. join (b, c)
2. join (c, b)
3. join (d, e)
4. join (e, d)
5. join (a, b)
6. join (a, c)
7. join (b, d)
8. join (c, e)
9. join (d, f)
10. join (e, f)

11. leave (a, c)
12. leave (b, d)
13. leave (d, f)

Figure B.11: Leave experiment 3.
1. join (a, b)
2. join (b, c)
3. join (c, a)
4. join (d, e)
5. join (e, f)
6. join (f, d)
7. join (b, e)
8. join (f, c)
9. join (g, b)
10. join (f, h)
11. join (a, x)
12. join (y, e)

13. leave (b, e)
14. leave (f, d)
15. leave (a, b)

Figure B.12: Leave experiment 4.
Figure B.13: Leave experiment 5.
(i) the VN source is broken ([L.4.c] at step 11) and the DL is replaced by a new DL with a physical source node, and

(ii) the VN sink is broken ([L.4.d] at step 12) and the DL is replaced by a new DL with a physical sink node.

Figure B.14: Leave experiment 6.

7. Leave experiment 7 in Figure B.15 is used to test DL detections in leave:
(i) the detection of re-exposed DLs that were embedded in the broken VN ([L.4.e] at step 11),

(ii) the effect of modifications to VN source/sink ([L.4.c and L.4.d] at step 12), and

(iii) breaking VN source/sink and replacing the affected DL by a new one that has a physical source/sink node ([L.5] at step 13).

---

Figure B.15: *Leave* experiment 7.
Appendix C

Glossary

In this appendix, we provide a glossary to help readers to clearly understand some of the terms used in this thesis.

**Adjacency matrix and adjacent arc list:** Adjacency matrix is a standard data structure representation of graph topology. In this thesis, it is used to store the EGV connectivity information of a VN. This information is required when an RL is broken. In this thesis, an adjacent arc list is a data structure used by a node to store the state information about its adjacent arcs, or the adjacent arcs of the VN in which the node is a component. This information is required in name resolution and pathname update.

**Client and server:** A client invokes an operation to request certain service, a server or a server group receives and processes the request to provide the service.

**Concurrency control:** Synchronize the execution of concurrent updates that may affect each other to produce a serializable result. An execution of a set of updates on an initial name graph is serializable if the resulting name graph is produced as if the updates were executed one after another on the initial name graph.

**Deterministic and nondeterministic group:** Deterministic group requires all members to be synchronized at the system-level in receiving and processing messages, nondeterministic
group allows this requirement to be relaxed to certain degree, relying on the application-level software to handle the inconsistency in an application-specific manner.

**Duplication loop:** A segment is a set of paths from a source node to a sink node. Paths in a segment share the same last hop leading to the sink. Two segments between a pair of nodes are distinct if their last hop to the sink are different. A duplication loop (DL) consists of multiple distinct segments between a pair of nodes. A segment in a DL is called the primary segment if no RD control for that DL is performed at the last hop of that segment. Other segments are called secondary segments.

**EGV and IGV:** The EGV is the external group view of the name graph perceived by omniscient observers with global information, the IGV is the internal group view of the name graph maintained by the underlying naming system. The two views are name resolution consistent if the results of name resolution performed on both views always match.

**Failure handling:** The activity of reorganizing the IGV to preserve name resolution consistency with respect to the remaining EGV after a failure. NA-failure-handling refers to failure handling when a normal arc fails; VN-failure-handling refers to failure handling when a VN is broken (i.e., when shadow edges or fade arcs fail).

**Group:** A set of receivers that are put together to receive group messages and to provide service cooperatively. A nested group is a group that contains other groups as its members.

**Group Communication:** Sending a message from one sender to a group of receivers. At network-level, this one-to-many message exchange is referred as multicast. Unicast refers to one-to-one message exchange and broadcast refers to one-to-all. Exchanging messages between an external sender and a receiver group is called inter-group communication. exchanging messages within a group is called intra-group communication. Ordered group communication ensures that message delivery satisfies certain ordering requirements defined for all the receiving members. Atomic group communication
ensures that a message to a group is received either by all the active members or by none in failure situations.

**Internet:** A network that inter-connects a set of sub-networks (e.g., LANs). The links connecting sub-networks are called internet links.

**Irrelevant message:** A message becomes irrelevant at a node if the message is sent to a group in which the node is no longer a member at the time of message delivery.

**Loop deep equal:** Two groups are deep equal if their name resolution results are the same. Two groups are loop deep equal if they are in the same RL. Loop deep equal is an equivalence relation.

**Missing message:** A message is missing at node if that message has been received by one of its immediate parents, but has not by that node.

**Normal arc, shadow edge and fade arc:** A normal arc connects two nodes that are not loop deep equal. Normal arcs always show up in both the EGV and the IGV. A shadow edge or a fade arc connects two nodes that are loop deep equal in the EGV. Fade arcs are excluded from the IGV and shadow edges are bidirectional. To avoid loop effect, a special name resolution procedure is executed in accessing shadow edges. A shadow tree is a tree consisting of only shadow edges in the IGV.

**Name graph:** A directed graph model that represents the naming structure of nested groups. A derived name graph is a DAG that represents the ancestor-descendant relationship among nodes in a name graph. Nodes that are in the same strongly connected component in the name graph are represented as a single node in the derived name graph. No ancestor-descendant relation is defined among these nodes.

**Name resolution:** A process that maps a group identifier to the group member process identifiers by traversing through a name graph. The paths in the name graph traversed to resolve a group identifier are called the name resolution paths of that group.
Originator or coordinator: The node that receives an update request from the user is the originator or the coordinator of the update transaction. The node which receives a group message (i.e., a name resolution request) from the user is called the originator group of that message.

Parallel arcs: Multiple arcs from one node to another node in a graph.

Parent and child: When a normal arc $a \rightarrow b$ exists in the name graph, node $a$ is an immediate parent of $b$ and node $b$ is an immediate child of $a$. When a path from node $a$ to node $b$ exists in the name graph and $a$ and $b$ are not loop deep equal, node $a$ is an ancestor of $b$ and node $b$ is a descendant of $a$. Unlike a tree, a node can have more than one immediate parent in the name graph. In a DL, a node is called the primary parent of the sink in that DL if no RD control is performed at that node for the DL. Other immediate parents of the sink in that DL are called secondary parents.

Pathname: A sequence of nodes representing an incoming path to a node in the derived name graph. This representation is internal to the naming system at that node and is used for RL/DL detections. All the pathnames at a node form the pathname set of the node. Pathname inheritance is a method to derive pathnames from the pathname set of an immediate parent. Pathname update relay is the process of updating pathnames at the descendants when a name graph update is performed.

RD control: The action of resolution duplication control to suppress name resolution duplications. It should not suppress duplicated messages resulted from retransmissions.

Resolution loop: A cycle is a single path loop in the name graph. A resolution loop (RL) contains all the cycles that are chained together. An RL is a strongly connected component in the terminology of graph theory. The existence of an RL indicates that nodes in the RL contain themselves as members.

Root node and leaf node: In a name graph, a node with in-degree zero is called a root node and a node with an out-degree zero is called a leaf.
Simple graph: A simple graph that has no self-loop or multiple edges.

Strongly connected component: In a graph, a strongly connected component is a subgraph in which all nodes are mutually reachable.

Subgraph: A subgraph rooted at a node consists of all the nodes and arcs that are reachable from that node. The subgraph of an update is the subgraph rooted at the update originator. The pathname sets at the nodes in this subgraph have to be updated after the update.

Virtual node: A virtual node (VN) contains all the nodes in an RL. Nodes in an VN are called components of the VN. VN does not physically exist. It is a logical concept used by the naming system to maintain the connectivity information among nodes that are loop deep equal.

Working arc and Working set: The working arc of an update is the arc to be added or deleted. the working set of the update contains all the nodes among which the loop deep equal relation is to be changed by the update.