MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

COMPUTATION STRUCTURES GROUP MEMO 224

LOCATING MIGRATORY OBJECTS IN AN INTERNET

by

CECELIA E. HENDERSON

Submitted to the Department of Electrical Engineering and Computer Science, M.I.T., on August 18, 1982, in partial fulfillment of the requirements for the degree of Master of Science.

JANUARY 1983

Locating Migratory Objects in an Internet

by

Cecelia E. Henderson

Submitted to the
Department of Electrical Engineering and Computer Science
on 18 August 1982 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

An abstract object is composed of a name and a value and exists in the memory of a single node. Historically, an object was confined to a single node: created, manipulated and destroyed. Finding an object's location was simple if its location were incorporated in its name.

Consider allowing an object to leave the node on which it was created. An object allowed to move from one node to another is called *migratory* and its movement is called *migration*. Migration is not the same as copying an object, since the identity of an object migrates with it. Finding an object's location is no longer as simple as looking at its name. In a broadcast or multicast network, a broadcast query could be used to find an object. In an internet composed of several types of networks, a broadcast query is impossible. This thesis addresses the problem of finding a migratory object's location in an internet given the object's name.

Chapter One

Introduction

Distributed processing has prospered in recent years both in research and in applications of research results. Management of distributed databases has contributed to this prosperity. Another contributor has been the unbundling of processing from a single, expensive, large, shared mainframe to many cheaper, small, personal workstations connected through a network. New language designs have brought the ideas of abstract user-defined objects and distributed processing together. This thesis was written to investigate how to manage movable abstract user-defined objects in a distributed-processing environment.

This chapter presents a high-level description of the thesis problem and proposes a solution. It discusses design criteria which the developer of a solution should consider and compares the solution with a rival. Research related to the thesis problem is discussed to illustrate approaches to similar problems. The last section of the chapter explains the structure of the remainder of the thesis.

1.1 Problem Description

In this section, a few basic terms are defined and used in a brief description of the thesis problem. Operating conditions assumed for the proposed solution are presented. A short explanation of the proposed solution closes the section.

1.1.1 Definitions and Description

A node or site is a physical processor with computing capability and with access to stable and volatile memory. Data stored in stable memory survive crashes; data in volatile memory do not. A node may be connected to other nodes to form a network, which may be connected to other networks to form an internet. Individual nodes have a location in the internet and communicate among themselves by sending messages through the internet. A network partition occurs through selective network component failure which divides the nodes of a network into two or more subsets which cannot communicate.

An abstract object is composed of a name and a value which exist in the memory of a single node. Historically, an object was confined to a single node: created, manipulated and destroyed at that node. Finding an object's location was simple if its location was incorporated in its name.

Consider allowing an object to leave its birthsite, the node on which it was created. This might occur because the node is no longer willing to support the object, or because physical constraints required by the object are no longer met. In this paper, an object allowed to move from one node to another is called *migratory* and its movement is called *migration*. Migration is not the same as copying an object, since the identity of an object migrates with it. How an object migrates is an interesting problem which this thesis will not address.

This thesis addresses the problem of finding a migratory object's location in an internet given the object's name. Once an object can migrate, finding an object's location is no longer as simple as looking at its name. In a broadcast or multicast network, a broadcast query could be used to find an object. In an internet composed of several types of networks, a broadcast query is impossible at the hardware level. In this thesis, no assumptions are made about the network topology.

1.1.2 Operating Assumptions

Migration is rare. The great majority of objects spend their entire existence at their birthsite. An object which has migrated is unlikely to migrate again. An object which has migrated more than once is likely to spend a long time between migrations. Migration is atomic, guaranteed to be either successful or not. An object is never left in a state of partial migration.

Crashes are infrequent. The mean time between failures for any particular node is long. The great majority of nodes spend days or weeks executing between crashes. Stable storage survives crashes with extremely high probability but is expensive [Lampson 79].

An internet may lose, duplicate or interminably delay messages but may never undetectably scramble a message. A scrambled message is detected and thrown out. A node knows its own location in the internet and the locations of a subset of other nodes.

1.1.3 Proposed Solution

In this paper, the facility which tracks migratory objects is known as a *registry*. It maps a unique identifier (uid) of an object to the object's location. A registry is composed of a database, containing entries for objects, and processes to maintain the database. When an object is created, its name and birthsite are added to the registry. The object's birthsite is incorporated in a unique identifier for the newly registered object which the registry returns. Migration is coordinated with the registry. When an object is destroyed, the registry is notified. To acquire an object's location, the object's uid is given to the registry. The object's location is returned, if possible.

A registry is fully distributed across an internet. That portion of a registry maintained by a single node is called a nodal registry. At a minimum, a nodal registry contains information about objects existing at its node. When an object is registered or destroyed, an interaction with a nodal registry takes place quickly without involving other nodes. When an object migrates, coordination is required only between the registries of the two nodes involved in the migration. A nodal registry tries to return local information when an object's location is wanted. If this is inadequate, the nodal registry searches for the object's location by asking other nodal registries. Through migrations and searches, a nodal registry can obtain information on objects existing elsewhere, but this cached information may be inaccurate and the nodal registry is not required to keep it past its immediate use.

The approach is novel in performing a search on a fully distributed registry of abstract migratory objects in an internet. [Accetta 80] and [Halstead 79] rely on a central registry. [Kahn 78] and [Shoch 80] keep track of other kinds of special objects. [Birrell 80] uses a broadcast network. [Lindsay 81a] relies on the birthsite to be available and have reliable information. [Lindsay 81a] is discussed more fully later in this chapter in the section on related research. [Oppen 81] imposes a structure on the components of its analog of a registry whereas the nodal registry space in this thesis is flat. [Oppen 81] also allows the registry analog to store much more information than a location. [Oppen 81] is discussed more fully in the section on related research.

1.2 Design Considerations

The first half of this section discusses requirements for the proposed solution's reliability and robustness. The second half discusses the question of ensuring that the proposed solution terminates.

1.2.1 Reliability and robustness

A reliable algorithm produces accurate information, never false information. Accurate information can be either an expected result (for instance, a sorted list) or notification that the algorithm cannot produce the expected result (for instance, because the input is not in the correct format).

In this thesis, a registry returning accurate information produces an object's current location or says that the object no longer exists. A registry may return slightly inaccurate information because an object can migrate between the time the registry returns the object's former location and the time this information is used. A registry may also fail to find an object's location because of a crashed node or a persistent network partition. For these reasons, a registry is not strictly reliable. However, a registry will never convey completely false information. A registry will never report that an object has been destroyed if the object still exists nor produce a location for an object that was destroyed before the registry began searching for the object.

A robust algorithm performs gracefully in the face of adverse conditions, providing a gradually degrading quality of service until a breakoff point is reached past which no service is offered. For instance, a retrieval algorithm, given a unique key, will return that key's corresponding record. Given an ambiguous key, it may return a set of records. Given no key, it may refuse to attempt retrieval.

A robust registry functions despite transitory disappearance or duplication of messages, node failures and network partitions. Its performance should degrade gracefully in the presence of persistent disappearance or duplication of messages, node failures and network partitions.

1.2.2 Guaranteed termination

The proposed solution should be guaranteed to terminate under all circumstances. [Francez 80] uses a spanning tree to control termination of disjoint processes by propagating success or failure information up the spanning tree and propagating termination information down. [Martin 81] uses a broadcast mechanism to control termination of an algorithm to determine a directed path from a source node to a destination node in a strongly connected directed graph. When a nodal registry is forced to search for an object's location, the search must terminate if the registry learns the object's location, if it learns that the object is destroyed or if it cannot be sure of either. Termination cannot depend on a broadcast mechanism or on an internet remaining strongly connected or on a sought object standing still.

1.3 Distributed Update vs. Distributed Search

A rival solution to that proposed by the thesis is to use a distributed registry kept accurate through distributed updates to a small set of known copies of registry information. Several of the approaches discussed in the next section use distributed updates. It was decided that a distributed search would be used as a research vehicle to find out whether the space required for replicate entries could be traded for time and effort spread across the nodes of the internet. The final chapter will return to this question.

1.4 Related Research

A problem quickly encountered in managing objects in distributed processing is that of naming the objects so that different objects may be distinguished and particular objects found and manipulated. Catalog management for a distributed database management system presents many problems similar to management of abstract user objects. A database relation becomes the object whose location must be found, though such an object moves very rarely and is usually replicated for speed of access and reliability. The first section below discusses how one distributed database management system, R*, performs catalog management. (SDD-1 [Rothnie 79] and distributed INGRES [Held 75] are other examples.)

A distributed message-handling system must keep track of the location of possible recipients of messages. Historically, a recipient's move from one location to another has been done by hand and the message system notified afterwards. The second section below describes how Grapevine [Birrell 80] locates message recipients.

The Xerox clearinghouse [Oppen 81] grew out of research on Grapevine. The clearinghouse maps names to sets of properties, a much richer mapping than the one supposed for this thesis. The third section below describes the clearinghouse.

Packet handling in a conventional packet-switched network can rely on static location tables augmented by dynamic performance information in routing packets from source to destination; not so a network composed of mobile nodes. The fourth section below describes the packet radionet, which relies on dynamic path-finding mechanisms to discover the current location of portable packet radios.

1.4.1 Catalog management in a distributed database management system

R* is a distributed version of System R [Astrahan 76, Lindsay 81b], a relational database management system. It was designed to support graceful growth, preserve local administrative autonomy and provide transparent data distribution. Its environment is a homogeneous set of relational database management systems or a nonhomogeneous set with the same external interface and a common communication protocol. The network interconnecting the database management systems may be slow, unreliable and expensive. Site autonomy is ensured through isolation and controlled access to shared information. Data accessability depends only on site availability. The entry of a new site is done by hand by the database administrators at the mutually agreeable sites. Objects (that is, database relations) can be created at any site and may infrequently migrate from one site to another site. A particular object is stored entirely at one site.

The designers of R* [Lindsay 81b] wished to allow a user the greatest possible freedom in choosing a *print name*, the name by which an object is manipulated in an application program. At the same time, they wished each object to have a uid within the distributed database management system which would be user-visible and indicate where a catalog entry for the object could be found. To resolve a print name into a uid, a System Wide Name (SWN) is used. An R* SWN has the following format:

(user@user_site.object_name@object_site)

where *user* is the userid of the user who created the object; *user_site* is the network site of this creator; and *object_site* is the network site at which the object was created.

The database catalog for R* [Lindsay 81a] supports a mapping from user-specified object references to low-level object identifiers and object locations. It is represented as a relation in its own database, taking advantage of the available concurrency control and recovery facilities. Each site maintains its own piece of the distributed catalog. The catalog at an object's birthsite will always know the object's current location. Each local catalog contains an entry for each locally stored object. A site may cache a catalog entry from a remote site in order to improve response, but these cached entries are not maintained consistently. Version numbers are used to detect discrepancies. The version number of catalog entries at an object's birth and storage sites are changed only when the object's location or access paths changes. No provision is made for locating an object if the local cache has no entry and the birthsite is unavailable.

The names handled by this paper's registy are abstract objects which users of the registry cannot examine. No structure such as that built into R* SWNs is apparent. A registry name is a uid, associated with an object when a user requests that the object be registered. Thereafter any request made by the user to the registry concerning that object must be accompanied by the object's registry uid.

1.4.2 Grapevine

Grapevine [Levin 76, Birrell 80] is a distributed message transport and registry system built in an internet environment with high bandwidth local networks [Metcalfe 76]. It allows for naming recipients of messages and an explicit mapping from names to current locations. A user may move from one naming authority to another provided he makes the move himself. Grapevine will not move a user by itself.

Recipients of messages are located in a multistep procedure. A broadcast reference locates a name lookup server which recognizes the naming authority of the recipient. The name lookup server supplies the network address of a registration server for the naming authority. The registration server supplies the network address of a message server willing to store the message for the recipient. When a user wishes to read his mail, he logs in at any workstation and asks the message server for his mail.

In contrast, this paper's registry recognizes that objects move and provides a mechanism for an object to report that it has moved. This mechanism requires coordination only between the two nodes involved in the migration, the source node and the destination node. No attempt is made to explain how an object migrates other than requiring this notification to the registry.

1.4.3 The Xerox clearinghouse

The Xerox clearinghouse [Oppen 81] is a sort of supercatalog of any person, device, service or resource that some one or some program might want to know something about. In addition to an object's location, the clearing house also knows what the object represents (a person or a printer, for example) and individual attributes (the person's phone number or the printer's linespeed). As such, it goes beyond the task set for this thesis of keeping track solely of an object's location.

In structure, a clearinghouse is a hierarchy of clearinghouses for (at the top) organizations and (next down) domains. Beneath domains are local names of objects. An object has a three-level name of the form:

localname@domain@organization

which is unique across the internet and is assigned by a system administrator. A client of the clearinghouse, requesting the location of a named object, may cause a hierarchical search of clearinghouse servers; first its own domain server, then the server of the named object's organization, and finally the server of the named object's domain, which will know the object's location.

By way of contrast, the name space of nodal registries and registry uids in this paper is flat. Registry uids are assigned in a distributed manner by the registry rather than by a system administrator. A registry cannot perform a hierarchically-structured search the way the Xerox clearinghouse does.

1.4.4 The packet radio network

The packet radio network [Kahn 78, Shoch 78a] is composed of three types of entities: stations, packet radios, and repeaters. Stations initiate all network control protocols which can have a global effect. Packet radios originate and receive messages and are slowly mobile. Repeaters are essentially relay links and will be ignored in the following discussion because the radios are able to subsume their functions.

In a small radionet, there is a single station that is aware of all radios in the network and determines the best route from one radio to another. It discovers new radios as they enter the range of the network and determines when radios leave the network. Best routes minimize cost and complexity. Each packet radio collects information on those radios within its broadcast range, summarizes this information and sends it to the station. The station uses this information to deduce overall connectivity. It determines a good route to itself from each radio and distributes this information. When a packet radio wishes to send a message, it requests a route from the station. On receiving the routing information, the radio broadcasts a route setup message, followed by the message.

If the number of radios in the network becomes large enough, a single station is no longer sufficient. Each station individually controls a group of contiguous radios. Each packet radio can remember any number of stations which try to gather it into their respective domains. A station is a neighbor of another station if they share at least one radio. Stations communicate directly only with neighboring stations. Stations do not coordinate with each other to adjust apportionment of radios to stations.

Routing a message to a radio outside the local station's domain begins when a routing query is sent from an originating packet radio to its local station. The local station asks its neighbor stations if they own the destination radio. These stations in turn ask their neighbors, building a route as the queries progress. The same query seen twice by the same station is discarded. The destination station passes the routing query to the destination radio, which initiates a route setup. The message is then sent from the originating radio directly to the destination radio. Thus the packet radionet uses dynamic source routing with relatively frequent updating of routing tables. The message source specifies all of the intermediate routing decisions.

This method of routing is analogous to one of the search methods used by a nodal registry to find the location of an object when the local information is inadequate.

1.5 The Remainder of the Thesis

Chapter Two introduces terminology on which discussions later in the thesis are based. A procedural interface for the registry system is presented followed by a description of two strategies that a nodal registry could use when searching for an object. Why guardians were thought relevant to the thesis problem, a model of a guardian and how the model effected the structure of a nodal registry are explained. Chapter Two closes with a description of two lower level system, a routing system and a neighbor system.

Chapters Three and Four present implementations for each of the two search strategies introduced in Chapter Two. Simple search conditions begin each presentation. Conditions grow more complicated as they allow for changes in neighbors, an unreliable network, object migration, concurrent searches, and crash recovery.

Chapter Five summarizes the thesis work.

Chapter Two

Environment

The first section of this chapter introduces common terminology on which to base the discussion of later sections. The second section presents a procedural interface for the registry system. The third section describes two strategies that a nodal registry could use when searching for an object. The fourth section explains why guardians were thought relevant to the thesis problem, gives a model of a guardian and explains how this model effected the structure of a nodal registry. The final section describes two lower level systems on which the registry depends, a routing system and a neighbor system.

2.1 Terminology

A *handle* is a registry-wide unique identifier returned by the registry when an object is added to the registry. The object's birthsite is incorporated in its handle.

An object is *registered* if an entry exists for it in a nodal registry. State information is kept on registered objects. A *resident* object exists wholly in the volatile or stable memory on the registry's node. All manipulation of the object takes place at that node. A *nonresident* object may exist on a different node. A *migrating* object is migrating to or from the registry's node, existing wholly on neither node. An object may be *sought*, to determine its location. A *destroyed* object exists nowhere.

An object *resides* on a node if and only if the nodal registry contains an entry marked as resident. This implies that a registry will know for certain, when asked, if a particular object resides at its node. It will neither give an incorrect negative answer, responding that the object is not resident when it is, nor give an incorrect positive answer, responding that the object is resident when it is not, nor be unable to answer.

A nodal registry contains an object entry marked as *nonresident* and giving a location if and only if the object currently resides at the given location or the object resided at the given location at some past time and has not resided at the registry's location between that time and the present. No more recent information on the object has been received. This means that any information on a nonresident object will at worst not advance a search, may help a search, but will never sabotage a search.

A pair of nodal registries each contain an object entry marked *migrating* if and only if the object is currently migrating from one of the nodes to the other. The entry location is the object's location before migration and the entry migration destinaton is the object's location after successful migration.

An object is *sought* if and only if the registry is conducting a search for the object's location or the nodal registry was asked for the object's location by another nodal registry at some point in the past.

An object entry is marked *destroyed* if and only if the object was resident and destroyed on the registry's node, or another nodal registry, when asked, responded that the object is destroyed. This guarantees that record of the destruction of an object may be obtained only from its last residence or in a chain of information exchange which originates with its last residence.

No entry at all may exist for an object. The object is *nonexistent* from the viewpoint of the registry. A nonexistent object may have entries in the database, but all these entries must be marked *nonresident*. Such object entries can exist in the fully distributed registry if and only if the object has been registered, destroyed and all records of its destruction removed.

A *seeking node* is a node the registry of which has initiated a search for an object. A *sought object* is an object of a search. A *queried node*, in relation to a seeking node and a sought object, is a node whose registry has been asked for the location of the sought object during the search initiated by the seeking node for the sought object. A *wormhole* is an entry in a nodal registry for a nonresident sought object giving a (possibly out-of-date) location for the object. A *search perimeter* is conceptually the dynamic bounds of a search for a migratory object. *Expansion* is asking new nodes about the location of a sought object.

2.2 Interface to the Registry

To find an object, the object must be registered. Thereafter, until the registration is rescinded and the object destroyed, location requests are resolved by consulting a nodal registry. When asked for the location of an object, a nodal registry examines its database. If an entry exists for the object, and a location is given in the entry, the nodal registry will return that location. If no entry exists or no location is given in an existing entry, the registry will return the object's birthsite.

Having the registry's first answer, an attempt will be made to use it. If the attempt is successful, the registry's task is complete. If the attempt is unsuccessful either because no response is received or because the object is no longer resident at that location, there must be some standard policy for deciding what is next to be done. One choice is to request the registry to conduct a search for the object. In answer to such a search request, the registry may respond with a new location, answer that the object has been destroyed or is nonexistent, or say that it has failed to locate the object but has no sure knowledge that the object does not exist.

The external interface to a registry is composed of a set of procedures which send a message to the registry requesting information or an action and wait for a response. The eight registry interface procedures are **Register**, **IsResident**, **Destroy**, **GetLocation**, **Search**, **Moving**, **Moved** and **NotMoved**.

Register accepts the name of a resident object and returns a handle. It requests that the registry create a new entry in its database pairing the handle with the name. The entry is marked as *resident*. It will signal *Isolated* and refuse the registration request if the node has no neighbors. Note that registering an object is not idempotent: registering the same object twice will result in the object having two uids associated with it. Multiple registration may occur due to a fortuitous crash, resulting in the loss of the response containing the handle. Under this circumtance, multiple registration causes no problem other than wasting space. An old nonresident entry, unused for a long time, will be deleted by the registry as part of normal housecleaning. However, if an object is deliberately registered more than once, the registry database may become inconsistent if the object migrates. Since the registry has no control over this, it is up to the users of the registry to correctly handle the multiple notification of the registry required to prevent errors.

IsResident accepts a handle and requests that the registry check if the object's entry is marked as *resident*. *True* is returned if the object is resident. *No(location)* is signalled if the object is not resident and a location is given in the object's entry. *Unknown* is signalled if no location is given or no entry exists. *Destroyed* is signalled if the object's entry is marked as destroyed.

Destroy accepts the handle of an object and requests that the registry mark its entry in the registry database as *destroyed*. *Nonresident* is signalled if the object is not resident. If no error is signalled, the entry is marked as requested and will be removed during normal housecleaning at some indeterminate later time, when all local record of the object will be lost. Note that only resident objects may be destroyed.

GetLocation accepts a handle and requests that the registry return the object's location. If the object is resident, the value returned will be the registry's location. If the object is nonresident and an entry exists with a location given, the value returned will be the location given in the entry. If the object is nonresident and no entry exists or no location is given in an existing entry, the value returned will be the object's birthsite. No guarantee is given that a nonresident object still resides at the location returned. *Destroyed* is signalled if the object is found to have been destroyed.

Search accepts the handle of an object and requests that the registry conduct a search for the object. The object's registry entry is assumed to be incorrect unless it says that the object is resident. Otherwise a search is conducted. If the object is found, the value returned will be the object's location at the time the search concluded successfully. If a record of the object's destruction is found, the procedure will signal *destroyed*. If the object is not found but may be unreachable the procedure will signal *not found*. If the object is not found and all nodes have been reached without finding it, the procedure will signal *nonexistent*. The object's registry entry is updated according to the search results.

The previous procedures enabled an object to be registered, sought and destroyed. The three interface procedures **Moving**, **Moved** and **NotMoved** provide a way to coordinate an object's migration with the registry. When migration begins, the two nodal registries involved are notified through separate calls to **Moving**. Examination of the object's entries at the two nodes is blocked for the duration of the move. If the migration succeeds, the two nodal registries are notified that the object now resides on a new node through separate calls to **Moved**. If the migration fails, the two nodal registries are notified that object remains at the same node through separate calls to **NotMoved**.

Moving accepts a handle, an origin and a destination. It requests that the registry mark the object's entry as *migrating* to the destination. If no entry exists, *not registered* will be signalled. If the origin is the registry's location, and the object is not resident prior to the call, *origin error* will be signalled. If the destination is the registry's location and the object is resident prior to the call, *destination error* will be signalled. If neither the origin nor the destination is the registry's location, *third party migration* will be signalled. The object's entry is marked as *migrating* only if no error is detected.

Moved accepts a handle, an origin and a destination. It requests that the registry update the object's entry to reflect a successful migration. An entry must exist in the database and be marked as *migrating* from the origin to the destination prior to the call. If no entry exists, *not registered* will be signalled. If an entry exists but is not marked as *migrating*, then *not migrating* will be signalled. If the origin does not match the location given in the entry, *origin error* will be signalled. If the destination does not match that in the entry, *destination error* will be signalled. If the entry location is the same as the registry's location, then the object is now nonresident. The entry is marked as *nonresident* and the location is set to the destination. If the entry location is not the same as the registry's location, then the object is now resident. The entry is marked as *resident* and the location is set to the destination. The object's entry is changed only if no error is detected.

NotMoved accepts a handle, an origin and a destination. It requests that the registry update the object's entry to reflect unsuccessful migration. An entry must exist in the database and be marked as *migrating* to the destination from the origin prior to the call. If no entry exists, *not registered* is signalled. If it is not marked as migrating, *not migrating* is signalled. If the origin is not correct, *origin error* is signalled. If the destination is not correct, *destination error* is signalled. If the entry location is the same as the registry's location, then the object has failed to migrate from the registry's location. The entry is marked as *resident*. If the entry location is not the same as the registry's location, then the object has failed to migrate to the registry's location. The entry is marked as *nonresident*. The object's entry is updated only if no error is detected.

2.3 Search Strategies

This paper investigates two types of searches which differ in how expansion of the search perimeter is controlled. In a *centralized search*, the decision to expand is made by the registry of the seeking node. Quashing a search is simple: the registry of the seeking node ceases sending out queries. A centralized search works the seeking node hard, may be slower than a distributed search and comes to a halt if the seeking node crashes. In a *distributed search*, the decision to expand is made by the registry of each queried node after the registry has established that the sought object is nonresident. Quashing a search is no longer simple. A distributed search spreads the work of the search across the nodes of the network, may be faster than performing a centralized search, and can continue even while the seeking node is crashed.

Important more for defining the set of nodes on its inside and its outside than for itself, a search perimeter is the set of nodes which have been or soon will be queried and have not yet replied. A node is *within* or *inside* a search perimeter if it has replied to the search query. A node is *outside* a search perimeter if it is neither on the search perimeter nor within the search perimeter. The concept of a search perimeter becomes important when a sought object migrates from a node outside its search perimeter to a node inside its search perimeter. Since its new node has already been queried, the object could conceivably elude the search.

Each new search is assigned a uid when it is initiated to distinguish it from every other search.

Either of the types of search may be used, but it is here presumed that only one type is used at any time. If both search methods were to coexist in the same internet, differing ways of initiating searches would be necessary as well as differing types of communication among searching nodes. It was not considered as part of the thesis to indicate how any of this could be done.

2.4 Guardians and the Structure of a Nodal Registry

The design of a fully distributed registry of infrequently moving objects was instigated by the idea of allowing guardians to move from one node in an internet to another as external resources and policies changed or as a guardian's internal state required. The structure of a nodal registry is based on a model of guardians. This section explains that model by closely following the discussion in [Liskov 80][1] . The model is then related to the registry's design.

A guardian resides completely on a single physical node and provides controlled

[1]This is not the current model of guardians under research. See [Liskov 82] for the current model of guardians.

access to a resource. A guardian contains *processes* and *data objects.* A process is the execution of a sequential program. The processes do the actual work of the guardian, manipulating the data objects and communicating with one another through shared data objects.

A guardian is an abstraction of a physical node of the underlying network. It supports one or more processes sharing private memory, and communicates with other guardians only by sending messages. Activity within a guardian is local and inexpensive because it takes place at a single physical node. Activity between guardians may be more costly. Each guardian acts as an autonomous unit, guarding its resource as it sees fit.

Each guardian definition can declare a set of permanent variables. The guardian's permanent state consists of these variables and all data reachable from them. This permanent state is guaranteed to survive crashes and is updated atomically. A guardian may also have volatile variables that comprise its volatile state. A guardian's volatile state does not survive crashes, nor do the guardian's processes.

A guardian definition has two code sections. The first is the **init** section, the purpose of which is to set the permanent state to a consistent initial value. This section runs whenever a new guardian is created. The second is the **start** section. This section runs when **init** is complete or when the guardian is restarted after a crash. It first initializes the guardian's volatile state and then performs the guardian's actual work.

Messages contain the *values* of data objects. To insure that the address space of a guardian remains local, it is impossible to place an address of an object in a message. Instead, a *token* for an object may be sent, which can be returned to the object's guardian to request manipulation of the object. Any object which is *transmissible* may have its value placed in a message [Herlihy 80]. Messages are sent to ports. Each port has a type that completely determines the set of messages it can receive and the responses to those messages. A guardian definition lists one or more port types to be provided for communication. Ports are globally available names with associated queues for receiving messages. Messages are not sent *through* ports; they are sent *to* ports. A port is owned by a unique guardian, which is the only guardian to receive messages on that port. Portnames are transmissible, but ports are not.

A nodal registry is a pair of cooperating permanent processes which maintain a database of object locations. Viewed as a guardian, a nodal registry's resource is its database, protected and manipulated by its processes. State information kept by these processes to control searches does not survive crashes. This information is reconstructed from a registry's database and from information supplied by other registries when a nodal registry is restarted after a crash. Each process executes in a large loop, reading a message from its message queue, processing it and returning to read another. Processes are distinguished with process uids included in the message so a response may be sent. The resident server process handles requests originating locally through calls to **Register**, **IsResident**, **Destroy**, **Moving**, **Moved**, **NotMoved**, and **GetLocation**. The search controller conducts searches on behalf of local requests made through calls to **Search**. It answers other nodal registries in a centralized search and conducts subsidiary searches for other nodal registries in a distributed search.

A registry database object entry contains an object handle, location, and disposition. A sample representation of an entry might be:

```
entry  =  record  [ obh:  handle,
                    loc:  location,
                    rft:  date,
                    dp:   disposition   ]
```

A **disposition** may be *sought*, *resident*, *nonresident*, *migrating*, or *destroyed*, reflecting the states that an object may assume from the viewpoint of a nodal registry.

Rft is used for database housekeeping: it is the date and time the entry was last examined or modified. Its initial value is the date and time when the entry was created and is updated whenever an entry is consulted. A nodal registry periodically removes old entries for nonresident and destroyed objects. Entries for sought objects can be deleted if the nodal registry knows that the search has terminated.

A copy of a nodal registry's database is kept on stable storage and a new copy is written during the execution of **Register**, **Destroy**, **Moving**, **Moved**, **NotMoved** and **Search**. This stable copy is used to restore searches when the registry restarts after a crash.

2.5 Lower Level Systems

The registry depends on the behavior of two lower level systems, a routing system and a neighbor system. The routing system constructs messages, transmits them across the internet and decomposes them for their recipients. The neighbor system constructs and maintains a logical neighboring relationship among the nodes of the internet.

2.5.1 Routing System

The interface to the routing system is composed of two procedures. Each constructs a low-level internet message from a high-level registry message and transmits it to an internet location where the high-level registry message is extracted and placed on the message queue of the appropriate registry process. The registry process can then read the high-level message from its message queue, determine its type and process it. **SendResidentMessage** accepts a location and a high-level message directed to a resident server. **SendSearchMessage** accepts a location and a high-level message directed to a search controller.

Exactly how a message is routed between a sending node and a receiving node is transparent to the nodal registry. If a path becomes unavailable, the routing system reroutes the message. A message directed to a location in a separate network partition is discarded. Messages may be lost, duplicated or interminably delayed, but never undetectably scrambled. A detectably scrambled message is discarded.

2.5.2 Neighbor System

The neighbor system supplies a nodal registry with the locations of a subset of other nodes which the registry can use as a basis for conducting searches for nonresident objects. A design for a neighbor system is an interesting problem which this thesis does not address. The three interface procedures to the neighbor system which are of interest to a nodal registry are **AreNeighbors**, **MyNeighbors** and **Neighbors**.

AreNeighbors accepts two locations and returns *true* if a neighboring relationship exists between them; *false* otherwise. The neighbor relation is symmetric.

Neighbors accepts a location and returns a list of the locations which have a neighboring relationship established with it. If the location is not in the neighbor system, *unrecognized location* will be signalled. If no neighboring relationships exist for the location, an empty list will be returned.

MyNeighbors returns a list of the locations which have a neighboring relationship with the location at which the procedure is executed. If the location is not in the neighbor system, *unrecognized location* will be signalled. If no neighboring relationships exist, an empty list will be returned. Calling **MyNeighbors** is equivalent to calling **Neighbors** with the location at which the procedure is executed.

The neighbor system caches information for some time T_{cache}, where it is assumed that searches never take longer than $T_{cache}-\Delta_{synch}$. T_{cache} is very large, for example a week or a month, longer than any search reasonably takes. Δ_{synch} is a small increment to make up for any lack of synchronization of clocks at different locations. The cached neighbor information is stable, but there is not a lot of it since changing neighbors is rare. Any information returned by the three interface procedures **AreNeighbors**, **MyNeighbors** and **Neighbors** incorporate this cached information.

The concept of neighboring nodes is above the level of the internet hardware. Neighboring nodes need not be on either end of a piece of wire. They need not be in the same subnetwork of the internet. A unique routing path need not exist between neighbors. As long as the underlying physical network is connected, the imposed network of neighbors should remain connected. Adding or removing a new node to the neighbor system, like adding or removing a new node to the internet, is a rare event. Changing neighbors is more common, and never isolates a node. The change is assumed to be atomic. The answer returned is correct at that moment but may be invalidated afterward.

Chapter Three

Controlling a Centralized Search

This chapter presents an implementation for a centralized search. A centralized search is conducted by a single nodal registry by sending messages to other nodal registries, receiving their replies and expanding the search perimeter. Simple search conditions begin the presentation. Conditions grow more complicated as they allow for changes in neighbors, an unreliable network, object migration, concurrent searches, and crash recovery.

3.1 Simple Search Conditions

This section presents how a centralized search is controlled under simple search conditions. Only one nodal registry is looking for the object. No nodes crash and the network transmits messages perfectly without loss, duplication or delay. The sought object does not migrate while the search is being conducted. The configuration of the neighbor system remains constant.

A search is initiated by calling the registry interface procedure Search which sends a **Search** request to the nodal registry's search controller. A **Search** request contains an object's handle and a uid of a process making a search request.

Information on current searches is kept by a search controller as a set, with an element of the set for each search. Each element contains a sought object's handle, a uid for the search, a uid for the process which requested the search by calling the registry interface procedure **Search**, the set of nodes on the search perimeter, the set of nodes inside the search perimeter, and the current state of the search.

On reading a **Search** request, a search controller first checks the database to see if the object is resident; the object may have migrated here since the registry was last asked for its location. If that is the case, the current location is returned as the value of the **Search** request. Otherwise, the search controller initializes a new element and adds it to its set of current searches. The sought object's handle and the uid of the requesting process are taken from the **Search** request. A new uid that incorporates the start time of the search is created for the search uid. The initial set of nodes on the search perimeter is the node's neighbors, acquired through a call to the neighbor system interface procedure **MyNeighbors**, plus the sought object's birthsite, acquired from the object's handle, plus an outdated residence, acquired if it exists from the sought object's registry entry. The initial set of nodes inside the search perimeter contains only the seeking node. The initial state of the search is *notfound*. Figure 3-1 illustrates the information kept by a centralized search controller under simple search conditions.

```
search      =  record   [ sought_object:          handle,
                          suid:                   search_uid,
                          requesting_process:     process_uid,
                          outqueries:             locset,
                          inqueries:              locset,
                          state:                  search_state]

locset         =  set[location]

search_state     =  variant   [ found:          location,
                                destroyed,
                                not_found:      null   ]
```

Figure 3-1: Information Kept by a Centralized Search Controller Under Simple Search Conditions

Each location in the initial set of nodes on the search perimeter is sent a **GetLocation** request. A search controller receiving a **GetLocation** request examines its registry database for information on the sought object.

If a sought object is resident, a queried node's search controller returns a **Found** response. On receipt of a **Found** response, the search state is set to *found.* The requesting process is notified of the sought object's location and the registry's database is modified to show the object as *nonresident* with the queried node's location.

If a sought object's entry is marked *destroyed,* a queried node's search controller returns a **Destroyed** response. On receipt of a **Destroyed** response, the search state is set to **destroyed**. The requesting process is notified of the sought object's destruction and the registry's database is modified to show the object as *destroyed.*

If a sought object is not resident or destroyed, a queried node's search controller returns a **NotFound** response. A **NotFound** response contains the responding node's location, the search's uid, and a set of locations. This set is created by calling the neighbor interface procedure **MyNeighbors**. If a wormhole location exists for the sought object in the queried node's registry database, it is added to this set. (Remember that a wormhole is an entry in a nodal registry for a nonresident sought object giving a (possibly out-of-date) location for the object.)

On receipt of a **NotFound** response, a seeking node's search controller finds the search's record in its set of current searches. The queried node's location is removed from the set of nodes on the search perimeter. If the search state is *found* or *destroyed* and the set of nodes on the search perimeter is now empty, there will be no more responses to this search. The search's record is removed from the set of current searches. If the search state is *not found,* the set of locations returned in the

NotFound response is screened against the sets of locations on and within the search perimeter. Any location from the **NotFound** response that is found in either set is discarded. If both the set of locations on the search perimeter and the set of screened locations are empty, every node has been asked without finding the sought object. The requesting process is notified that the object is nonexistent. The search's record is removed from the set of current searches. Any entry for the object in the registry database is removed. If either set is nonempty, the queried node's location is added to the set of locations inside the search perimeter, the screened locations are added to the set of locations on the search perimeter and a **GetLocation** request is sent to each of the screened locations.

3.2 Changing Neighbors

In this section, the neighboring relations between nodes are allowed to change. Such a change causes no problem if it occurs while no search is in progress. The change in configuration is picked up naturally when a seeking or queried node uses the neighbor interface procedure **MyNeighbors**.

If a neighboring relation is dropped but each node retains other neighbors, the change does not effect a search in progress. At worst it might take longer for one of the nodes to receive a search query. If dropped neighbor relations result in a disconnected neighbor network, searches will be confined to the separate networks. This is assumed never to happen unless the change is purposeful and permanent.

If a neighboring relation is added but each node was not previously isolated, the change does not effect a search in progress. At worst it will give the searching node one more node to screen against its sets of already-queried nodes.

If a node becomes isolated in preparation to leaving the neighbor system, it may be overlooked by a search in progress. This will cause a search for an object resident on the isolated node to report the object as nonexistent. This is acceptable because the node is removing itself from the internet and all objects resident on it will become unreachable.

If a new node is added to the neighbor system while a search is in progress, it may also be overlooked. This is acceptable because objects resident on the new node cannot be the object of a search started before the new node joined the neighbor system.

A node might still be overlooked if it changes neighbors on or outside the search perimeter for neighbors inside the search perimeter. Being overlooked under these circumstances is prevented because the neighbor system caches old neighbor information. When a former neighbor asks for its neighbors to return this information to the seeking node, the might-have-been-overlooked node is included.

3.3 An Unreliable Network

In this section, a network is admitted to be unreliable. Messages can be lost, duplicated or interminably delayed, although not scrambled. The routing system detects imperfect messages and discards them. Partitions can occur. To handle an unreliable network, a timeout mechanism is introduced for retransmission of queries. Retransmission of queries introduces the possibility of duplicate queries, forcing each request and response to be idempotent. A message may still be interminably delayed and could arrive after its search has terminated. Recognition and correct handling of such delayed messages is required.

Two new pieces of information are kept for each search: the last time a response was received; and the number of times requests have been resent since the last

```
search    = record  [ sought_object:          handle,
                      suid:                   search_uid,
                      requesting_process:     process_uid,
                      outqueries:             locset,
                      inqueries:              locset,
                      last_response:          date,
                      timeout_count:          int  ]

locset       = set[location]
```

Figure 3-2: Information Kept by a Centralized Search Controller Assuming an Unreliable Network

response.

Last_response is the date and time when the last response for a search was received. When the **search** record is initialized or a response is received, last_response is set to the current date and time. Periodically, the search controller checks last_response for each **search**. If **last_response** is beyond a tolerable limit, new requests are sent to each location in **outqueries**.

Timeout_count is the number of times requests have been renewed since the last response. When a **search** record is initialized or a response received, **timeout_count** is set to zero. The timeout count is incremented when requests are renewed. If the timeout count surpasses a tolerable limit, the search controller assumes that the remaining outstanding queries will not be answered. The search is terminated by removing the **search** record and notifying the requesting process that the object was not found.

Each **GetLocation** request received is now checked for age. The search's start time, incorporated in the search uid, is compared against T_{cache}-Δ_{synch}, the length of time the neighbor system has ached information. If a greater period has passed since the search started, the request is discarded as too old.

It is not surprising to find no record of a search when a response comes in. This means that the search has terminated. The response can be discarded. This makes it unnecessary to keep the search record after receiving a **Found** or **Destroyed** response, which in turn makes keeping search state information superfluous.

Special handling of duplicate **Found** and **Destroyed** responses is unnecessary because the first **Found** or **Destroyed** response removes the search's record. Special handling of duplicate **NotFound** responses is unnecessary because the set of locations is screened against the set of nodes on the search perimeter as well as the set of nodes inside the search perimeter.

A network partition is an aggravated case of an unreliable network. Depending on when the partition occurs and its duration, the residence of a sought object may never be queried or may never be able to respond to a query. No special handling beyond the timeout mechanism discussed above is proposed.

3.4 Migration

In this section, objects are allowed to migrate. This introduces no new problem unless an object migrates across a search perimeter, from the outside from a node which has not been queried to the inside to a node which has already been queried. To solve this problem, evidence of the search must be stored at queried nodes.

Note that if an object migrates in the other direction, from the inside from a node which has been queried to the outside to a node which has not been queried, the answer returned by the search will be out-of-date. Since migration is rare, it is always possible to catch up with a migrating object.

As part of the processing of a **GetLocation** request by a queried node's search controller, an entry is created in the registry database for the sought object. The entry is marked as *sought* and the seeking node and search uid stored. When an object migrates to a queried node, the resident server sends a **Eureka** response to the seeking node. In reaction, a search controller sends a **EurekaAck**. A **Eureka** response contains a sought object's handle, a search uid and a location. The location is the sought object's new residence. A **EurekaAck** consists of a search uid and a location. The location is the responding seeking node.

Because the sought object's new residence is unlikely to be queried again about its new resident, notification is sent repeatedly until an acknowledgement is received or until it times out. To support this, a resident server keeps a set of records for each newly migrated resident that it knows to be the object of a search. Each record contains the object's handle, the uid of the search, the location of the seeking node, the time when the last notification was sent, and the number of times that a notification has been sent. (See Figure 3-3.)

```
eureka  =  record  [ sought_object:    handle,
                     seeking_node:     location,
                     suid:             search_uid,
                     when_sent:        date,
                     timeout_count:    int  ]
```

Figure 3-3: Information Kept by a Centralized Resident Server Assuming Migrating Objects

Periodically, a resident server checks the time when a **Eureka** response was sent for each record in its set. If the time passed is beyond a tolerable limit, a new **Eureka** response is sent and the **timeout_count** is incremented. If the **timeout_count** surpasses a tolerable limit, the **eureka** record is removed and the corresponding search information in the object's registry entry is deleted.

On receipt of a **Eureka** response, a search controller always sends a **EurekaAck** to the location in the **Eureka** response. If the search has terminated, that is all the search controller must do. If the search has not terminated, the search controller removes the search's record from its set of current searches and notifies the requesting process of the object's location. It marks the object's registry entry as *nonresident* with the location in the **Eureka** response.

On receipt of a **EurekaAck**, a resident server finds and removes its **eureka** record. The search information stored in the object's registry entry is also removed. If a **eureka** record is not found, the **EurekaAck** is discarded.

Some synchronization between the operations that move an object and send eureka messages is required to ensure that an object is not missed by a search. This synchronization is discussed below.

3.4.1 A closer look at migration

When an object attempts to migrate, both nodal registries are notified through separate calls to the interface procedure **Moving**. Examination of the object's entry at these two nodes is blocked. If migration is unsuccessful, separate calls to the interface procedure **NotMoved** notify both nodal registries and unblock the object's entry.

If migration is successful, both nodal registries are notified through separate calls to the interface procedure **Moved**. There are three major cases of successful migration to consider: migration to nodes outside the search perimeter; migration to nodes on the search perimeter; and migration to nodes inside the search perimeter. Each major case has three subcases, one each for the origin of the migration: inside, on or outside the search perimeter. The following subsections will treat each of the subcases.

3.4.2 Migration from inside the search perimeter to outside the search perimeter

This is a null category: if a node of residence is inside the search perimeter, then the search has terminated. This kind of migration has no effect on the terminated search; no extra messages are generated.

3.4.3 Migration from the search perimeter to outside the search perimeter

This occurs when a **GetLocation** request arrives at the destination while the object's entry is blocked. Once the object's entry is unblocked, the **GetLocation** request is processed normally. A **NotFound** response is sent which includes a list of neighbors with the node's new residence added. The node's new residence will be naturally included in the next expansion of the search perimeter.

3.4.4 Migration completely outside the search perimeter

If migration completes before the search perimeter overtakes either the origin or destination, this case is the same from the viewpoint of the seeking node as that of no migration taking place.

3.4.5 Migration to the search perimeter from inside the search perimeter

This is a null category: if a node of residence is inside the search perimeter, then the search has terminated. However, a delayed **GetLocation** request may arrive while examination of the entry is blocked at the destination. Once the block is removed through a call to the interface procedure **Moved**, the **GetLocation** request will be processed normally. A **Found** response will be sent to the seeking node. This **Found** response will be treated as a duplicate response and will be discarded.

3.4.6 Migration from the search perimeter to the search perimeter

This occurs when a **GetLocation** request arrives at both the origin and destination while the object's entry is blocked at each. Once the object's entry at the origin and destination is unblocked through a call to **Moved**, each processes the **GetLocation** request normally. The origin sends a **NotFound** response to the seeking node. The destination sends a **Found** response to the seeking node, which terminates the search.

3.4.7 Migration to the search perimeter from outside the search perimeter

This occurs when a **GetLocation** request arrives at the origin while examination of the object's entry is still blocked. Once the block on the object's entry at the destination is removed through a call to **Moved**, the **GetLocation** request will be processed normally, causing a **Found** response to be sent to the seeking node. This **Found** response will be treated as a normal **Found** response, terminating the search.

3.4.8 Migration completely inside the search perimeter

This ia a null category: if a node of residence is inside the search perimeter, then the search has terminated. However, the destination node would not realize this because success in searching is not anounced. If the destination still has the object in its database as *sought*, it will send a delayed **Eureka** response to the seeking node. The seeking node will acknowledge and discard it.

It should be pointed out that an object might migrate to the seeking node, by definition inside the search perimeter. The resident server sends a **Eureka** response to the search controller without having to notice that the search controller is at the same node. The object's migration does not interfere with the search because the search information is separate from the database entry.

3.4.9 Migration from the search perimeter to inside the search perimeter

This occurs when the destination has already answered a **GetLocation** request before the object's entry is blocked at the destination, and the origin receives a **GetLocation** request while the object's entry is blocked at the origin. Care must be taken about the order in which the object's entry is unblocked at the origin and destination.

If the origin is unblocked first, the pending **GetLocation** request will be processed normally. A **NotFound** response will be sent that includes the destination in its set of locations. Because the destination is inside the search perimeter, its location will be discarded during the screening process and the search may terminate incorrectly.

To prevent this, the destination must be unblocked first. At this point, during the execution of the interface procedure **Moved** when the resident server is marking the object's entry as *resident*, the resident server discovers that the object has migrated across the search perimeter. The resident server creates a **eureka** record, enters it in its set of **eureka** records and sends a **Eureka** response to the seeking node. The **Moved** procedure call does not return to allow a subsequent **Moved** call for the origin until the **Eureka** is acknowledged or times out.

3.4.10 Migration from outside the search perimeter to inside the search perimeter

This case has a timing problem similar to the preceding case. If the origin is unblocked before proper notification of the seeking node by the destination, then it is possible that the search perimeter may overtake the origin and cause the search to terminate incorrectly. Unblocking the destination, followed by the origin, should be handled as explained in the previous case.

3.5 Concurrent Searches

In this section, more than one nodal registry may conduct a search for the same object at the same time. Separate searches overlap when their search perimeters cross. To support concurrent searches, the search information stored in the registry entry is expanded from a single (seeking node, search uid) pair to a set of (seeking node, search uid) pairs. To support concurrent searches by a single node for a single object, the **requesting_process** field is changed from a single process uid to a set of process uids.

When a search controller initiates a search, it marks the object's registry entry as *sought* by itself, storing the search uid and its own location as seeking node. A subsequent **GetLocation** request to the resident server will report that the object is being sought. A subsequent **Search** request to the search controller will add the requesting process to the set of requesting processes

When a search controller receives a **GetLocation** request from another node, it examines the registry database as before. If the object is nonresident, a **NotFound** response is sent to the seeking node in the **GetLocation** request and the new seeking node and search uid are stored. A **GetLocation** request is still idempotent because the (seeking node, search uid) pairs are stored in a set, which filters duplicates.

Concurrent searches require a minor modification in the behavior of a node after a successful migration. If the node is the destination of a successful migration and is inside the search perimeter, a **Eureka** response is sent to each seeking node. To ensure retransmission of the **Eureka** until an acknowledgement is received or a timeout occurs, a **eureka** record is created for each (seeking node, search uid) pair.

3.6 Crash Recovery

In this section, it is admitted that physical nodes crash. To enable recovery of searches, the registry database must be copied to stable storage whenever the disposition of an object changes. To restart a nodal registry, a single process is started which in turns starts the resident server and search controller.

The resident server scans the database to houseclean and to recover information on newly migrated residents. It looks for old entries for nonresident objects which may be thrown out and for entries marked both *sought* and *resident*. The latter will be for recently migrated objects. For each such entry, a new **eureka** record is created, initialized and inserted in the resident server's **eurekaset**. The **sought_object**, **suid** and **seeking_node** fields are set from information in the registry entry. A **Eureka** response is sent to the seeking node.

The search controller scans the database to recover information on objects sought by itself and by other nodes. It looks for entries marked *nonresident* and *sought* by some other node. For each such entry and each (search uid, seeking node) pair, a **StillSearching?** request is sent to the seeking node. A **StillSearching?** request contains a sought object's handle, a search uid, and the location to be answered. When a search controller receives a **StillSearching?** request, it checks its set of current searches and responds with a **SearchReport** response. A **SearchReport** response contains a sought object's handle, a search uid and a boolean value. If the boolean value is *true*, the search is still being conducted and its (search uid, seeking node) should be kept. If the boolean value is *false*, the search is no longer being conducted and its (search uid, seeking node) pair may be discarded.

To rebuild its set of current searches, a search controller also scans the registry database for entries marked as *sought* by itself. For each such entry, a new search record is created, initialized and inserted in the search controller's srchset. The **sought_object** and **suid** fields are set from the registry entry. The **last_response** field is set to the current date and time. The **timeout_count** is zeroed.

The **requesting_processes** field does not survive crashes. Processes are revived from breakpoints and reissue registry requests, so it is reasonable to make the **requesting_processes** field empty and attempt to restart the searches in anticipation of reissued requests.

The **outqueries** field does not survive crashes and is initialized as for a new search. Responses from nodes on the old search perimeter will be processed normally as they come in, resulting in a new search perimeter growing in three directions. One, it will grow outward from the seeking node toward the old search perimeter, querying nodes inside the old search perimeter. Two, it will grow inward from the old search perimeter toward the seeking node, querying nodes inside the old search perimeter. These two will eventually meet and stifle each other. Three, it will grow outward from the old search perimeter, querying nodes outside the old search perimeter.

The **inqueries** field does not survive crashes and is reinitialized with the seeking node as its only element. A possibly large number of duplicate queries will be sent because the screening mechanism will be shortcircuited.

3.7 Housecleaning

Some provision needs to be made to cleanse the registry database of out-of-date search information and unused entries. This can be handled by a process which sleeps most of the time, awaking periodically to make a cleaning sweep through the database. For each entry marked as *sought* by another node, the housekeeper process can send a **StillSearching?** request. The **SearchReport** will be handled normally by the search controller. For each entry marked as *nonresident*, the housekeeper process examines the date and time the entry was last used. If the date and time are beyond a reasonable limit, the entry is removed.

Alternatively, the resident server and search controller can periodically scan the database as they do for crash recovery.

3.8 Discussion

A centralized search is reliable under simple search conditions. It will never say that an object has been destroyed when it still exists or give a location for an object that was destroyed before it started searching. Slightly outdated information may be returned when an object migrates just after it was found. A centralized search may report that it couldn't find an object if the network is unreliable or nodes crash.

A centralized search functions robustly in the face of lost, duplicated or delayed messages, node failures and network partition. It will return a correct answer if the object's location is not in a separate partition or on a crashed node which remains down for the duration of the search. If the seeking node becomes isolated, reliable information is still available on resident objects and cached information is available on nonresident objects.

A centralized search always terminates. If no positive answer is received, it will terminate when all nodes have been asked. A search will time out if responses are persistently lacking. Crashes of the seeking node reset the timeout mechanism but are rare and will not cause a search to be restarted indefinitely.

Chapter Four

Controlling a distributed search

This chapter presents an implementation for a distributed search. As in the previous chapter, search conditions are at first simple then grow more complicated to allow for changes in neighbors, an unreliable network, migration of a sought object, concurrent searches and crash recovery.

A distributed search, like a centralized search, is initiated at a single nodal registry. Unlike a centralized search, it is conducted as an expanding ripple of subsidiary searches. The seeking node asks each of its neighbors for the location of the sought object. Before responding, each neighbor asks *its* neighbors about the sought object. An already-queried node responds immediately that it has already been queried to any subsequent querying node. A newly-queried node responds immediately to both the seeking node and its querying node only if the sought object is resident or known to be destroyed. Once all a node's neighbors have answered, a node answers its querying node, propagating information back toward the seeking node. A distributed search terminates when the seeking node receives notification of the sought object's location or destruction, or when all its neighbors respond negatively to its original request.

4.1 Simple Search Conditions

This section presents how a distributed search is controlled under the same simple search conditions as in the previous chapter. Only one nodal registry initiates a search. No nodes crash and the network transmits messages perfectly without loss, duplication or delay. The sought object does not migrate while the search is being conducted. The configuration of the neighbor system remains constant.

A distributed search is initiated in the same manner as a centralized search, by calling the registry interface procedure **Search**, which sends a **Search** request to the nodal registry's search controller. A **Search** request contains the same information as for a centralized search.

Information on current searches is kept by a search controller as a set, with an element of the set for each search, just as for a centralized search. Additionally the location of the querying node and the seeking node are stored to enable a node to propagate search information back towards the seeking node.

On reading a **Search** request, a search controller first checks the database to see if the object is resident; the object may have migrated here since the registry was last asked for its location. If that is the case, the current location is returned as the value of the **Search** request. Otherwise, the search controller initializes a new element and adds it to the set of current searches. The seeking node and the querying node are set to this location. Outqueries is the list of nodes which have been sent queries but have not answered. It is initialized to contain this node's neighbors. The initial search state is *notfound*. Figure 4-1 illustrates the information kept by a distributed search controller under simple search conditions.

A search perimeter is created by sending a **GetLocation** request to each location in **outqueries**. A **GetLocation** request now contains two locations: the querying and seeking nodes. A search controller receiving a **GetLocation** request examines its set of current searches for a search with the same search uid.

If a search record exists with the same search uid, it means that this node has

```
search          = record   [sought_object:            handle,
                             suid:                     search_uid,
                             requesting_process:       process_uid,
                             seeking_node:             location,
                             querying_node:            location,
                             outqueries:               locset,
                             state:                    search_state]

search_state      =  variant   [ found:         location,
                                 destroyed,
                                 notfound:      null]

locset      =  set[location].
```

Figure 4-1: Information Kept by a Distributed Search Controller Under Simple Search Conditions

already been queried and a subsidiary search is already underway. The search controller immediately sends a **NotFound** response to the querying node. On receipt of a **NotFound** response, a search controller finds the search record and removes the responding node's location from **outqueries**. If this leaves the set empty, the search record is removed. If the search state is *notfound* and all nodes have responded, it means that the subsidiary search has concluded unsuccessfully. The search controller sends a **NotFound** response to its querying node.

Only if a subsidiary search is not underway does the search controller examine the registry database for information on the sought object.

If a sought object is resident, a queried node's search controller sends a **Found** response to both its querying node and the seeking node. On receipt of a **Found** response, a search controller finds the search record and removes the responding node from **outqueries**. The search state is changed to *found*. If this leaves the set empty, the search record is removed. The search controller propagates the search information back towards the seeking node by sending a **Found** response to its querying node.

If a sought object's entry is marked *destroyed*, a search controller sends a **Destroyed** response to both is querying node and the seeking node. On receipt of a **Destroyed** response, a search controller finds the search record and removes the responding node from **outqueries**. The search state is changed to *destroyed*. If this leaves the set empty, the search record is removed. The search controller sends a **Destroyed** response to its querying node.

Otherwise a queried node's search controller initiates a subsidiary search. A new element is initialized for its set of current searches. The sought object's handle, the search uid, the querying node's location and the seeking node's location are taken from the **GetLocation** request. The requesting process field is set null. The initial set of locations is the set of the node's neighbors plus an outdated residence, acquired if it exists from the registry database, minus the locations of the querying and seeking nodes. The initial search state is *notfound*. A search perimeter is expanded by a queried node sending a **GetLocation** request to the search controller at each location in the set of locations for its subsidiary search. The queried node then awaits responses.

The seeking node collects responses just like any querying node, but reports the result of its search to the requesting process rather than another node. It discards the search record when the search terminates.

4.2 Changing Neighbors

The discussion in the previous chapter on changing neighbor relations during a centralized search pertains to changing neighbors during a distributed search and will not be repeated here.

4.3 Unreliable Network

In this section, a network is admitted to be unreliable. Messages can be lost, duplicated or interminably delayed, although not scrambled. Partitions can occur. To handle an unreliable network, a timeout mechanism is introduced for retransmission of requests. This mechanism is the same as that introduced in the previous chapter to handle an unreliable network for a centralized search. Figure 4-2 shows the two new pieces of information kept for each search: the last time a response was received; and the number of times requests have been renewed since the last response. The discussion of **last_response** and **timeout_count** in the previous chapter pertains as well to their use in a distributed search and will not be repeated here. Also previously discussed is the determination of a **GetLocation** request as old enough to be discarded.

As before, a delayed response arriving after a search's record is removed would not be rare. These delayed messages which arrive after their search has terminated are simply discarded. This makes it unnecessary to keep the **search** record after receiving a **Found** or **Destroyed** response. This in turn makes the search state superfluous.

Special handling of duplicate **Found** and **Destroyed** responses is unnecessary because the first **Found** or **Destroyed** response removes the search's record. Special handling of duplicate **NotFound** responses is unnecessary because removing the

```
search   = record  [sought_object:        handle,
                     suid:                 search_uid,
                     requesting_process:   process_uid,
                     seeking_node:         location,
                     querying_node:        location,
                     outqueries:           locset,
                     last_response:        date,
                     timeout_count:        int]


locset   = set[location]
```

Figure 4-2: Information Kept by a Distributed Search Controller Assuming an Unreliable Network

responding node's location from **outqueries** more than once doesn't cause an error.

As mentioned in the previous chapter, a network partition is an aggravated case of an unreliable network and no special handling beyond the timeout mechanism is proposed.

4.4 Migration

In this section, objects are allowed to migrate. This introduces no new problem unless an object migrates across a search perimeter, from the outside from a node which has not been queried to the inside to a node which has already been queried.. As in the previous chapter, to solve this problem evidence of the search must be stored at queried nodes.

As part of the processing of a **GetLocation** request by a queried node's search controller, an entry is created in the registry database for the sought object. The object's entry is marked as *sought* and the seeking node, querying node and search uid stored . As before, when an object migrates to a queried node, a **Eureka** response is sent to the seeking node and a **EurekaAck** is sent in reaction. Also as before, notification is sent repeatedly until an acknowledgement is received or until it times out.

On receipt of a **Eureka** response, a search controller removes the search's record from its set of current searches and notifies the requesting processes of the object's location. It marks the object's registry entry as *nonresident* with the location in the **Eureka** response.

On receipt of a **EurekaAck**, a resident server removes its **eureka** record. The search information stored in the object's registry entry is also removed. If a **eureka** record is not found, the **EurekaAck** is discarded.

The search controller of the same node may be conducting a subsidiary search for the newly migrated resident. It need not be notified because the search will eventually terminate when all its queried nodes respond. The search controller can then check the registry database to see if the object migrated to its location during the search.

4.5 Concurrent Searches

In this section, more than one nodal registry may conduct a search for the same object at the same time. Separate searches overlap when their search perimeters cross. To support concurrent searches, the search information stored in the registry entry is expanded from a single (seeking node, querying node, search uid) triple to a set of (seeking node, querying node, search uid) triples and the **requesting_process** field is changed from a single process uid to a set of process uids.

When a search controller initiates a search, it marks the object's registry entry as *sought* by itself, storing the search uid and its own location as seeking node. A subsequent **GetLocation** request to the resident server will report that the object is being sought. A subsequent **Search** request to the search controller will add the requesting process to the set of requesting processes. If an object is already sought by another node when a search controller tries to initiate a search, a new search is started separately which will be independent of the one being conducted by the other node.

When a search controller receives a **GetLocation** request from another node, it examines the registry database as before. If the object is nonresident, a **NotFound** response is sent to the seeking node in the **GetLocation** request and the new seeking node and search uid are stored. A **GetLocation** request is still idempotent because the (seeking node, querying node, search uid) triples are stored in a set, which filters duplicates.

When a sought object successfully migrates from outside the search perimeter to inside the search perimeter, the resident server discovers its newly migrated resident is *sought*. A **Eureka** response is sent and a **eureka** record is created for each (seeking node, querying node, search uid) pair.

4.6 Crash Recovery

In this section, it is admitted that physical nodes crash. To enable recovery of searches, the registry database must be copied to stable storage whenever the disposition of an object changes. To restart a nodal registry, a single process is started which in turns starts the resident server and search controller. The resident server scans the database to houseclean and to recover information on newly migrated residents. The search controller scans the database to recover information on objects sought by itself and by other nodes. A **StillSearching?** request is sent to the seeking node for each object sought by another node. A **SearchReport** is sent in reaction.

A resident server scans the registry database looking for old entries for nonresident objects which may be thrown out and for entries marked both *sought* and *resident*. The latter will be for recently migrated objects. For each such entry, a new **eureka** record is created, initialized and inserted in the resident server's **eurekaset**. The **sought_object**, **suid** and **seeking_node** fields are set from information in the registry entry. A **Eureka** response is sent to the seeking node.

A search controller scans the registry database for entries marked *nonresident* and *sought* by some other node. For each such entry and each (search uid, querying node, seeking node) triple, a **StillSearching?** request is sent to the seeking node. A **StillSearching?** request contains a sought object's handle, a search uid, and the location to be answered. When a search controller receives a **StillSearching?** request, it checks its set of current searches and responds with a **SearchReport**. A **SearchReport** contains a sought object's handle, a search uid and a boolean value. If the boolean value is *false*, the search is no longer being conducted and its (search uid, querying node, seeking node) triple may be discarded.

If the boolean value is *true*, the search is still being conducted and its (search uid, querying node, seeking node) triple should be kept in the queried node's registry database. A subsidiary search record is created, initialized and inserted in the search controller's **srchset**. The **sought_object**, **suid**, **querying_node** and **seeking_node** fields are set from the registry entry. The **last_response** field is set to the current date and time. The **timeout_count** is zeroed. The **requesting_processes** field is set null. The **outqueries** field is set as for a new subsidiary search.

A search controller also scans the registry database for entries marked as *sought* by itself. For each such entry, a new **search** record is created, initialized and inserted in the search controller's **srchset**. The **sought_object** and **suid** fields are set from the registry entry. The **last_response** field is set to the current date and time. The **timeout_count** is zeroed.

The **requesting_processes** field does not survive crashes. Processes are revived from breakpoints and reissue registry requests, so it is reasonable to make the **requesting_processes** field empty and attempt to restart the searches in anticipation of reissued requests. The **outqueries** field is initialized as for a new search.

4.7 Discussion

The distributed-search scheme must be analyzed in the same terms as the centralized-search scheme: for reliability, robustness and guaranteed termination. Because control is not centralized, a distributed search must also handle the problem of a **GetLocation** request delayed past the duration of its search, causing an 'orphaned' subsidiary search.

4.7.1 Reliability

A distributed search, like a centralized search, is reliable under simple search conditions. It will never report that an object exists if the object was destroyed before the search started. It will never report that an object is destroyed if the object still exists. Outdated information may be reported if an object migrates soon after it is found. If a partition is persistent or the residence of the sought object remains crashed for the duration of a search, the search will report that it could not find the object.

4.7.2 Robustness

A distributed search functions robustly in the face of lost, duplicated or delayed messages, nonpersistent node failure and transient network partitions. If the seeking node becomes isolated, reliable information is available on resident objects and cached information on nonresident objects.

4.7.3 Termination

Unless a **GetLocation** request is delayed beyond the termination of its search, a distributed search always terminates. The case of a **GetLocation** request delayed beyond the termination of its search is discussed below. If the sought object is not found, it will terminate after all subsidiary searches have terminated and reported their failure. It will time out if responses are persistently lacking. Crashes of the seeking node reset the timeout mechanism, but are rare and will not cause a search to be restarted indefinitely. There is, however, no way to stop a search once an object has been found other than letting the search run its course.

4.7.4 The problem of delayed queries

One of the operating assumptions presented in Chapter One was that messages may be interminably delayed. If a **GetLocation** request is 'interminably delayed', it could finally arrive at its destination after all record of its search has been removed through normal housecleaning. This could cause the completed and forgotten search to be resurrected.

To prevent this, a timestamp is required in all **GetLocation** requests. It could be incorporated as part of the search uid or stand on its own as a piece of the query. The nodal registry, receiving a **GetLocation** request and finding no record of the search in its database, checks the timestamp. If the timestamp shows that the query is 'old', the query may be discarded. Alternatively, a **StillSearching?** request may be sent to the seeking node, and the 'old' query held pending a **SearchReport** response.

This changes the problem to deciding what 'old' is. A nodal registry housecleans at regular intervals and whenever the node crashes. If the timestamp shows that the search started more than a regular housecleaning internal ago, it is 'old'.

This definition requires that the interval between housecleanings be long in relation to the time it takes to conduct a search. The desire to keep the database clean works to shorten the interval. Less than an hour would risk falsely identifying a query as old, and involve significant overhead in frequent housecleaning. More than a several hours would risk a search resurrecting itself unnecessarily. The interval would remain the discretion of policy-makers at each network location.

This mechanism can use the timestamp already incorporated in the search uid for enforcing the timing restraint of cached neighbor information. The two definitions of 'old' do not coincide: $T_{cache}-\Delta_{synch}$ should be much longer than the interval between housecleanings.

Chapter Five

Final Words

This final chapter discusses a simulation of the search methods, compares the two search methods, summarizes the thesis work and presents problems unsolved by the work.

5.1 Simulation of the Search Methods

A separate simulation of the two search methods was programmed in CLU on a TOPS-20 machine. The simulations accounted for crashing and recovery of nodes, lost or delayed messages, network partitions, multiple searches for the same or different objects, and migration of sought objects. They did not take into account dynamic reconfiguration of the neighbor system or garbage collection of the registry database.

Simulated time was based on a sweep through all the nodes of the internet, with every node touched during each sweep. During a sweep, nodes were picked at random to process messages already arrived, so that anomalies due to a consistent processing order were eliminated. (Aberrant message propagation is an example of a possible anomaly due to processing order. If during every sweep nodes were picked in the same serial order to process their messages, propagation of search queries and replies would proceed faster in the direction of the serial order.) After a node was picked, the simulation randomly decided whether it should crash and for how long. The downtimes were multiples of one sweep.

Messages were arbitrarily picked to be lost or duplicated or delayed by the simulated message transport system, with the delays again a multiple of one sweep. Network partitions were simulated by consistently discarding messages intended for nodes in separate partitions.

Objects were picked to migrate before, during and after searches for them.

Under the conditions of the simulation, searches for existing objects mostly succeeded in correctly reporting the object's current location at the time of the search, regardless of the object's migration history. When a search failed (returning 'not found' for an existing object), it was due to expiration of the timeout mechanism in the face of a persistent network partition or opportune crashes of individual nodes (i.e., the seeking node or the node of residence).

5.2 Comparison: Centralized vs. Distributed

A centralized search requires a seeking node to retain information about the nodes that have not yet responded to its queries, but is easily ended. There is small fear of it being erroneously resurrected. It may take longer than a distributed search under similar conditions because all queries must be send and all responses must be precessed by a single seeking node.

A distributed search sends more messages since no screening is done for already-queried nodes by querying nodes. Time is saved because the messages are sent in parallel by all the querying nodes rather than by a single seeking node. Ensuring that a distributed search ends is not straightforward.

Both search strategies require the queried node to remember that it has been queried in case the sought object migrates there during the search. They also both require positive acknowledgement from the current node of residence.

5.3 Searching vs. Multiple Updating

In working out the details of making the search methods work, it became clear that the original thought of trading execution time for storage space when choosing a search method rather than multiple updating would not stand up, given a network of arbitrary topology. Queried nodes need to save information on sought objects, though perhaps not as much information as for a distributed update scheme. The latter would cache information at each node. This cached information may or may not be stable. At least two complete copies of the entire database must be kept (stable) so that the database could be highly available. These copies would likely be partitioned among many locations. Some of the stable information kept for the searches is not needed. If it is decided that a node should not continue searches begun before a crash, search information need not be stable. If the number of seaches in progress at any time is small, remembering search information requires less storage than duplicate location information.

Although some space is saved in the searching methods, this is done at a prohibitive cost in searching under the assumptions in the thesis. The search strategies could each be more efficient if more was known about the network topology and if that topology had a good structure. The Xerox Clearinghouse [Oppen 81] has a three-level hierarchical structure reflected in the names of Clearinghouse objects which speeds its searches. A spine topology being discussed at MIT would make searches practical. In the spine topology, gateways form a long thin communications line through a longhaul network. Off these gateways hang local networks such as ring nets. Such a topology could be searched in a distributed manner in three hops, using broadcast at each point.

5.4 Evaluation of Thesis Work

The thesis problem as proposed was to devise a method of tracking user-defined objects that are allowed to move from one machine to another in an internet. Tracking these objects is a support mechanism facilitating the use of the objects without requiring a prohibitive penalty from the object's users in the form of possible program reconstructuring or tracking of the objects by the users themselves.

Search methods were explored to investigate whether execution time could be traded for the storage used in distributed updating in previous research. In particular, the two search methods, centralized and distributed, explained in detail in the foregoing chapters, were promoted as alternatives and tested under conditions simulating some conditions of a real internet.

The two search methods seem impractical for use under current conditions for three reasons. First, the searches are expensive, as discussed in the previous section. Second, termination is a problem for the distributed search, especially terminating a search when the answer has already been discovered. Third, because of the nature of the searches, an object could never be considered found unless a positive acknowledgement was received from the current node of residence. This condition could result in a "miss"; that is, the object could be reported to the seeking process as "not found" whereas the actual residence was known but not directly verifiable because temporarily unreachable. A distributed update scheme would give the location without requiring the residence be currently available.

Future research should investigate alternate strategies. Until a distributed update scheme is worked out in detail, a clear evaluation of its performance and cost is not possible. Working on new network topologies might be very interesting. Further work on termination of distributed searches would be useful.

References

[Abraham 80]
S.M. Abraham and Y.K. Dalal.
Techniques for Decentralized Management of Distributed Systems.
In *Twentieth IEEE Computer Society International Conference (CompCon)*, pages 430-437. IEEE, February, 1980.

[Abramson 70]
N. Abramson.
The Aloha System.
In *AFIPS Conference Proceedings, FJCC*, pages 281-285. 1970.

[Accetta 80]
M. Accetta, G. Robertson, M. Satyanarayanan, and M. Thompson.
The Design of a Network-Based Central File System.
Technical Report CMU-CS-80-134, Carnegie-Mellon University, August, 1980.

[Astrahan 76]
M.M. Astrahan et al.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1(2):97-137, June, 1976.

[BBN 76]
Bolt, Beranek and Newman, Inc.
MSG: The Interprocess Communication Facility for the National Software Works.
Report 3237, Bolt, Beranek and Newman, Inc., January, 1976.

[Binder 75]
R. Binder, N. Abramson, F. Kuo, A. Okinak, and D. Wax.
Aloha packet broadcasting - a retrospect.
In *AFIPS Conference Proceedings, NCC*, pages 203-215. 1975.

[Birrell 80]
A. Birrell, R. Levin and M. Schroeder.
Grapevine: A Distributed Electronic Message System.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIPGLAN, Pala Mesa Resort, December, 1980.

[Clark 80]
D.D. Clark and L. Svobodova.
Design of Distributed Systems Supporting Local Autonomy.
In *Twentieth IEEE Computer Society International Conference (CompCon)*, pages 438-444. IEEE, February, 1980.

[Cook 80]
R.P. Cook.
Abstractions for Distributed Programming.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Cullum 76]
P.G. Cullum.
The transmission subsystem in Systems Network Architecture.
IBM Systems Journal 15(1):24-38, January, 1976.

[Cypser 78]
R.J. Cypser.
Communications Architecture for Distributed Systems.
Addison-Wesley Publishing Company, Inc., 1978.
Uses IBM's SNA as major example throughout text.

[Davies 80]
C.T. Davies Jr.
Position Paper on Fundamental Issues in Distributed Computing.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Dijkstra 79]
E.W. Dijkstra.
Termination Detection for Diffusing Computations.
EWD-687a

[Finkel 81]
Finkel.
Distributed Breadth-first Search Algorithm.
Submitted to IEEE Transactions on Software Engineering

[Francez 80]
N. Francez.
Distributed Termination.
ACM Transactions on Programming Languages and Systems 2(1):42-55, January, 1980.
Spanning tree used to control termination of disjoint processes.

[Friedman 78]
D.U. Friedman.
Communication Complexity of Distributed Shortest Path Algorithms.
Master's thesis, Massachusetts Institute of Technology, December, 1978.

[Gray 78]
J. Gray.
Notes on Data Base Operating Systems.
Research Report RJ2188(30001), IBM San Jose Research Laboratory, February, 1978.

[Gray 80]
J. Gray.
Minimizing the Number of Messages in Commit Protocols.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Halstead 79]
R.H. Halstead.
Reference Tree Networks: Virtual Machine and Implementation.
Technical Report MIT/LCS/TR-222, Massachusetts Institute of Technology, July, 1979.

[Hammer 80]
M. Hammer and D. Shipman.
Reliability Mechanisms for SDD-1: A System for Distributed Databases.
ACM Transactions on Database Systems 5(4):431-466, December, 1980.
Description of reliable network substrate for SDD-1.

[Held 75]
G. Held, M. Stonebraker, and E. Wong.
INGRES: A Relational Data Base System.
In *AFIPS National Computer Conference.* AFIPS, 1975.

[Herlihy 80]
M.P. Herlihy.
Transmitting Abstract Values in Messages.
Master's thesis, Massachusetts Institute of Technology, April, 1980.

[Johansson 80]
C. Johansson and B. Liskov.
Catalogs in a Distributed System.
DSG Note 74, Massachusetts Institute of Technology, Cambridge, MA

[Kahn 78]
R.E. Kahn, S.A Gronemeyer, J. Burchfiel, R.C. Kunzelman.
Advances in Packet Radio Technology.
Proceedings of the IEEE 66(11):28:1-6, November, 1978.

[Lampson 79]
Butler W. Lampson and Howard E. Sturgis.
Crash Recovery in a Distributed Data Storage System.
Xerox Palo Alto.

[Lampson 80]
B.W. Lampson.
Replicated Commit.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Lauer 80]
H.C. Lauer.
Decentralised Assignment of Names in Networks.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Levin 76]
R. Levin and M. Schroeder.
Teleinformatics 79.
North-Holland Publishing Company, 1976, pages 29-33.
In the chapter entitled 'Transport of Electronic Messages Through a Network'.

[Lindsay 80]
B. Lindsay.
Authorization and Site Autonomy in Distributed Database Management Systems.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Lindsay 81a]
B. Lindsay.
Object Naming and Catalog Management for a Distributed Database Manager.
Technical Report RJ2914, International Business Machines, 1981.

[Lindsay 81b]
B. Lindsay and P.G. Selinger.
Site Autonomy Issues in R: A Distributed Database Management System.*
Technical Report RJ2927, International Business Machines, 1981.

[Liskov 79]
B. Liskov et al.
CLU Reference Manual.
Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, October, 1979.

[Liskov 80]
B. Liskov.
Linguistic Support for Distributed Programs: A Status Report.
Computer Structures Group Memo 201, Massachusetts Institute of Technology, October, 1980.

[Liskov 82]
B. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Systems.
In *Proceedings of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pages 7-19. January, 1982.

[Lomet 80]
D.B. Lomet.
The Ordering of Activities in Distributed Systems.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Lunn 81]
K. Lunn and K.H. Bennett.
An Algorithm for Resource Location in a Loosely Linked Distributed Computer System.
ACM Operating Systems Review 15(2), April, 1981.
Superficial explanation of centralized directory with local resource directory in a superimposed ring structure.

[Martin 81]
A.J. Martin.
A Distributed Path Algorithm and its Correctness Proof.
To appear in TOPLAS.

[McFadyen 76]
J.H. McFadyen.
Systems Network Architecture: An Overview.
IBM Systems Journal 15(1):4-23, January, 1976.

[McQuillan 77]
J.M. McQuillan.
Routing Algorithms for Computer Networks -- A Survey.
In *Proceedings of the National Telecommunications Conference*, pages 28:1-6. 1977.

[McQuillan 80]
J.M. McQuillan, I. Richer, and E.C. Rosen.
The New Routing Algorithm for the ARPANET.
IEEE Transactions on Communications 28(5):711-719, May, 1980.
Description of new routing algorithm and comparison with former routing algorithm.

[Metcalfe 76]
R.M. Metcalfe and D.R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM 19(7):395-404, July, 1976.
General discussion of design, topology, mechanism, implementation and performance of the Ethernet.

[Needham 80]
R.M. Needham.
Current Research in Distributed Computing.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Nelson 80]
B.J. Nelson.
Remote Procedure Call.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Oppen 81]
Derek C. Oppen and Yogen K. Dalal.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Destributed Environment.
Technical Report OPD-T8103, Xerox Office Products Division, October, 1981.

[Rashid 80a]
R.F. Rashid.
An Inter-Process Communication Facility for Unix.
Technical Report CMU-CS-80-124, Carnegie-Mellon University, March, 1980.

[Rashid 80b]
R.F. Rashid.
Research into Loosely-Coupled Distributed Systems at CMU.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Roberts 72]
L. Roberts and B. Wessler.
Computer Communication Networks.
Prentice-Hall, Englewood Cliffs, N.J., 1972, .
In the chapter entitled 'The ARPA Computer Network'.

[Rothnie 79]
J.B. Rothnie et al.
SDD-1: A System for Distributed Databases.
Technical Report CCA-02-79 (Revised), Computer Corporation of America, August, 1979.

[Rowe 79]
L. Rowe and K. Birman.
Network Support for a Distributed Data Base System.
In *Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks*, pages 337-352. August, 1979.

[Rudin 76]
H. Rudin.
On Routing and Delta Routing: A Taxonomy and Performance Comparison of techniques for Packet-Switched Networks.
IEEE Transactions on Communications 24(1):43-59, January, 1976.

[Schwartz 77]
Mischa Schwartz.
Computer-Communication Network Design and Analysis.
Prentice-Hall, Inc., 1977.

[Shoch 78a]
J.F. Shoch.
A Note on Inter-Network Naming, Addressing and Routing.
Internet Experiment Note 19, Xerox Palo Alto Research Center, January, 1978.

[Shoch 78b]
J.F. Shoch.
Internetwork Naming, Addressing, and Routing.
In *Seventeenth IEEE Computer Society International Conference (CompCon)*, pages 72-79. IEEE, September, 1978.

[Shoch 80]
J.F. Shoch and J.A. Hupp.
Notes on the 'Worm' programs -- some early experience with a distributed computation.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Stonebraker 79]
M. Stonebraker.
Concurrency Control and Consistency of Multiple Copies of Date in Distributed INGRES.
IEEE Transactions on Software Engineering SE-5(3):188-194, May, 1979.

[Sturgis 80]
H.E. Sturgis.
Notes on Current Research.
Submitted to the Workshop on Fundamental Issues in Distributed Computing, ACM/SIGOPS and ACM/SIGPLAN, Pala Mesa Resort, December, 1980.

[Svobodova 79]
L. Svobodova, B. Liskov, and D. Clark.
Distributed Computer Systems: Structure and Semantics.
Technical Report LCS TR-215, Massachusetts Institute of Technology, March, 1979.

[Tanenbaum 81]
Andrew S. Tanenbaum.
Computer Networks.
Prentice-Hall, Inc., 1981.

[Unknown 81]
Unknown.
Grapevine.

[Walden 72]
D.C. Walden.
A System for Interprocess Communication in a Resource Sharing Computer Network.
CACM Volume=15 (4), April, 1972.

Table of Contents

Table of Figures