Distributed Match-Making for Processes in Computer Networks*
Preliminary Version

Sape J. Mullender
Paul M.B. Vitányi

CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

ABSTRACT

In the very large multiprocessor systems and, on a grander scale, computer networks now emerging, processes are not tied to fixed processors but run on processors taken from a pool of processors. Processors are released when a process dies, migrates or when the process crashes. In distributed operating systems using the service concept, processes can be clients asking for a service, servers giving a service or both. Establishing communication between a process asking for a service and a process giving that service, without centralized control in a distributed environment with mobile processes, constitutes the problem of distributed match-making. Logically, such a match-making phase precedes routing in store-and-forward computer networks of this type. Algorithms for distributed match-making are developed and their complexity is investigated in terms of message passes and in terms of storage needed. The theoretical limitations of distributed match-making are established, and the techniques are applied to several network topologies.

1. Introduction

We investigate the problem of setting up communication-when-needed between processes in a multiprocessor network where processes have *names* but no permanent *addresses*. A mechanism for this purpose is called a *name-server*, analogous to the telephone system's directory assistance server: given a *name* it returns an *address*. A single *centralized* name server in the network can be taken out through a single processor crash, thereby effectively killing all communication and crashing the entire network. A more robust solution is *distributing* the name server. A great variety of options and problems of both theoretical and practical interest are attached to this issue. Our motivation was provided by the design objectives of the *Amoeba* distributed operating system project [11].

1.1. The Catering Service Problem

Suppose you want to give a party in your Silicon Valley home, but do not care for the bother. You want a catering service. Now it so happens, that you do not know the address or telephone number of such a service. Anyway, even if you did, this would not do you much good. In Silicon Valley such small outfits come and go so fast that it is unlikely that this service, which you used two years ago, still exists at the old address. You can phone them, but the number gets you somebody who has never heard of your old catering service. There are several courses of action you can take.

- One way to solve your problem is to send mail to everybody in town asking whether they supply catering service. In computer networks this is called *broadcasting*.
- Another way is to wait until you get an advertisement leaflet of a catering service in your mailbox. Below we call this *sweeping*.

Most likely, you do one of the following:

- You look in the Yellow Pages under the appropriate heading. If everybody exclusively uses YP for all services then we may view the YP outfit as a centralized name server. Services reveal their whereabouts by advertising there and clients look them up there. If the YP company crashes then clients and services cannot be matched anymore, and society grinds to a halt.
- You buy a suitable newspaper and look up "catering" in the advertisement section. Now the name server is distributed. Catering services advertise in many newspapers. If one newspaper flounders, this will not create problems for you.
- You ask some of your friends whether *they* know where to find the desired service. Some of your friends crashing will not prevent you finding a caterer. The name server is distributed in this case as well, and, depending on how sociable you are, perhaps better.

* This work was supported by the *Stichting Mathematisch Centrum*.

Having found the address or telephone number of a catering service, you have to find a way to route your request to them. Thus, match-making between clients and services necessarily precedes routing in a mobile society. Note that the catering service, in order to execute the task you set them, may call on other services such as a car rental service. The catering service then is a client with respect to the car rental service. Clearly, everybody can be server, client or both.

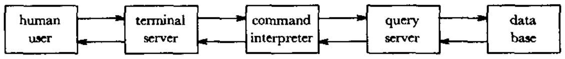
1.2. Multiprocessors & Computer Networks

New generation computers must be fast, reliable, and flexible. One way to achieve this is to build them from a small number of basic processor-memory modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance. The primary difference between machines should be the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. It should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users. Note that in this view a *computer network* is essentially such machine on a grand scale.

Software design for these new machines can advantageously be based on the *object model.* In this model, the system deals with abstract *objects*, each of which has some set of abstract *operations* that can be performed on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it. The object model is also known under the name of "abstract data type" [6]. A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

1.3. The Service Model

It is convenient to *implement* the object model in terms of clients (users) who send messages to services [10]. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies.

> As an example, consider a *file server.* The design must deal with how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. The internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. A server can itself be client to another service. The possible hierarchy of services is the strength of the model:
>
> 
>
>
> A crash of the database server, will be detected by the query server, which must then try to recover from it. The query server can retry the request, it might rephrase a query to get the answer from another database server, and as a last resort, it can report failure to its client, the command interpreter. In this way the human client at the top of the hierarchy gets to cope only with irrecoverable errors and crashes in the system.

More precisely, *Services* are offered by a number of *server* processes, distributed over the network. *Client* processes send *requests* to services; the services carry out these requests and return a *reply*. Essentially, every job in the system is executed by a dynamic network of servers executing each other's requests. So a process can be a client, a server, or both, and change its role dynamically. New services can be created by installing server processes for them. Services can be removed by destroying their server processes (or by making them stop behaving like a server, i.e., by telling them to stop receiving requests). Server processes can be migrated through the network, either by actually moving the process from one host to another, or only in effect, by destroying the server process in one host and creating another one in a different host at the same time. A specific service may be offered by one, or by more than one server process. In the latter case, we assume that all server processes that belong to one service are equivalent: a client sees the same result, regardless which server process carries out its request. A process resides in a network *node.* Each node has an *address* and we assume that, given an address, the network is capable of routing a message to the node at that address. A service is identified by its *port.* A port uniquely names a service. We shall therefore also refer to a service by its port. Ports give no clue about the physical location of a server process.

1.4. The Problem of Match-Making

Before a client can send a request to a server which provides the desired service, the client has to locate that server. The problem of efficient *routing* arises at a later stage; first the address of the destination has to be found in a *match-making* phase. We can view match-making as yet another service in the system, be it the *primus inter pares*. Thus, we need to implement a *name server* to serve a connection between client process and server process.

A *centralized* name server must reside at a so-called *well-known address* which does not change and is known to all processes. (Clearly, the name server cannot be used to locate itself.) When the host of the name server crashes, the entire network crashes. This solution also causes an overload of messages in the neighborhood of the host.

When clients *broadcast* for services with "where are you" messages, we have an example of a *distributed* name server. This solution is more robust than the centralized one. But in large store-and-forward networks, where messages are forwarded from node to node to their destination, broadcasting is considerably more costly than sending a message directly to its destination. Broadcast messages are sent to every host, while point-to-point messages need only pass through the hosts on the path between client and server. Conventional broadcast methods for locating services need a minimum of $\Omega(n)$ message passes to do the broadcast (*e.g.*, via a spanning tree [2]).

We investigate realizations of name servers in the entire range between centralized and distributed forms. The efficiency of solutions is measured in terms of message passes and local storage. It appears that, in many n-node networks, very efficient distributed match-making between processes can be done in $O(\sqrt{n})$ message passes, by using limited numbers of point-to-point messages.

1.5. Locate Algorithms

In all cases, the method used to locate a port is the following: A server process s located at address A_s and offering a service identified by a port π, selects a collection P_s of network nodes and *posts* at these nodes that server s receives requests on port π at the address A_s. Each of the nodes in P_s stores this information in a cache for future reference. When a client process c located at address A_c has a request to send to π, it selects a collection of network nodes Q_c and *queries* each node in Q_c for the address of π. When $P_s \cap Q_c \neq \varnothing$, the node(s) in the intersection will return a message to c stating that π is available at A_s. If $P_s = \{s\}$ and $Q_c = U$ then the technique is called *broadcasting*; if $P_s = U$ and $Q_c = \{c\}$ then the technique is called *sweeping*.

1.6. Outline of the paper.

We develop a class of distributed algorithms for match-making between client processes and server processes in computer networks. We investigate the expected performance of such an algorithm under random choices. Subsequently, we determine the optimal lower bound on the performance in number of message passes or "hops" for any such algorithm, in any network, under any strategy, distributed or not. This yields a combinatorial lemma which may be interesting in its own right, and results in a lower bound on the trade-off product between the number of nodes a server advertises at and the number of nodes a client inquires at. We consider criteria for robustness. Second, we apply the method to particular networks, both designed networks and spontaneously emerged networks. Finally, a probabilistic and a hashing algorithm for match-making are investigated.

1.7. Related work.

Distributed match-making between *clients* and *servers* will be used in the *Amoeba* distributed operating system [11]. Essentially the Manhattan topology method below has been used before in the torus-shaped Stony Brook Microcomputer Network [5]. Some current multiprocessor systems avoid the communication overload due to mobile processes, which use broadcasting to do the match-making, by opting for the processes to run on fixed processors [8]. Other system designers have chosen for mobile processes, but use the crash-vulnerable solution of a centralized name server [7]. The present paper introduces, and systematically explores for the first time, the general concept of distributed match-making.

2. A Theory of Distributed Match-Making

Below we obtain lower bounds on the message pass complexity of a class of Locate algorithms (called Shotgun Locate), for the entire range from centralized to distributed methods, and for any network topology. In the next section we give methods which achieve these lower bounds, or nearly achieve these lower bounds, for many network topologies.

2.1. Framework for Shotgun Locate

The networks we consider are point-to-point (store-and-forward) communications networks described by an undirected communications graph $G = (U,E)$, with a set of nodes U representing the processors of the network, and a set of edges E representing bidirectional noninterfering communication channels between them. No common memory is shared by the node-processors. Each node processes messages it receives from its neighbors, performs local computations on messages and sends messages to neighbors. All these actions take finite time. A *message pass* or *hop* consists of the sending of a message from one node to one of its direct neighbors.

1. The number of message passes needed for match-making depends on the topology of a network. We want to obtain topology independent lower bounds. Therefore, assume that all messages can be routed in one message pass to their destinations. Equivalently, assume that the network is a *complete* graph. Lower bounds on the needed number of message passes in complete networks *a fortiori* hold for all networks.

2. For each network $G=(U,E)$ and associated match-making algorithm, there are total functions P, Q such that:

$$P,Q: U \rightarrow 2^U \quad .$$

(Here 2^U is the set of all subsets of U.) Any server residing at node i starts its stay there by *posting* its (port, address) pair at each node in $P(i)$. Any client residing at node j *queries* each node in $Q(j)$ for each service (port) it requires.

3. We assume that all nodes j have a *cache* which is large enough to store all (port, address) pairs associated with addresses i such that $j \in P(i)$. That is, the nodes at which the *rendez-vous*' are made can hold all posted material. The caches are large enough to hold so many (port, address) pairs that they never have to discard one for a server that is still active. Entries are made or updated whenever a message is received from a server process with its address (or when a reply from a locate operation is received). We can timestamp the messages to determine which addresses are out of date in case of a conflict.

We have dubbed this class of algorithms *Shotgun Locate* algorithms. (Put so many pheasants in the bushes that the hunter can expect success for the amount of shot he is willing to spend.) Later we consider alternative locate methods: *Hash Locate* where the functions P, Q depend on the service ports as well, and *Lighthouse Locate* which is a probabilistic version of Shotgun Locate where too-small caches can discard (port, address) pairs.

2.2. Probabilistic Analysis

Let the number of elements in a given set U (universe) of nodes be n. Let a given server s reside at node i. Let p be the cardinality of $P(i) \subseteq U$, the set of nodes where s posts its whereabouts. Let a given client c reside at node j. Let q be the number of elements in $Q(j) \subseteq U$, the set of nodes queried by c. If the elements of $P(i)$ and $Q(j)$ are randomly chosen then the probability for any one element of U to be an element of $P(i)$ $[Q(j)]$ is p/n $[q/n]$. If $P(i)$ and $Q(j)$ are chosen independently then the probability for any one element of U to be an element in both $P(i)$ and $Q(j)$ is pq/n^2. Since there are n elements in U, the expected size of $P(i) \cap Q(j)$ is given by

$$E(\#(P(i) \cap Q(j))) = \frac{pq}{n} \quad .$$

Therefore, to expect one full node in $P(i) \cap Q(j)$, we must have $p+q \geqslant 2\sqrt{n}$. This is the situation for a particular pair of nodes. For the performance of the whole network we have to consider the combined performance of the n^2 pairs of nodes. The above analysis holds for each pair i,j of elements of U, since they are all interchangeable. Consequently, the minimal *average* value of $p+q$ over all pairs in U^2 must be $2\sqrt{n}$, in order to expect a successful match-making for *each* pair.

By choice of the sets $P(i)$ and $Q(j)$, we may improve the situation in two ways:

- The method deterministically yields success.
- We get by with $p+q < 2\sqrt{n}$.

2.3. Number of Messages for Match-Making

To match a server at node i to a client at node j the following actions have to take place. The server at i tells a set $P(i)$ of nodes about its location. Client j queries a set $Q(j)$ of nodes for the desired service. Call the set of nodes $r_{i,j} = P(i) \cap Q(j)$ the set of *rendez-vous* nodes, that is, the nodes at which a *rendez-vous* between a client at j looking for a service and a server at i offering that service can be made.

Definition. The $n \times n$ matrix, R, with entries $r_{i,j}$ $(1 \leqslant i,j \leqslant n)$ is the *rendez-vous* matrix. Each entry $r_{i,j}$, in the ith row and jth column of R, represents the set of *rendez-vous* nodes where the client at node j can find the location i and port of the server at node i. Note that:

$$\bigcup_{j=1}^{n} r_{i,j} \subseteq P(i) \quad \& \quad \bigcup_{i=1}^{n} r_{i,j} \subseteq Q(j) \quad . \qquad \text{(M1)}$$

To prevent waste in message passes, we can take care that the inclusions in (M1) are replaced by equalities. (But then the surviving subnetwork after a node crash may lack this property again.) An optimal shotgun method has exactly *one* element in each $r_{i,j}$. Below, we represent such singleton sets by their single element. (If faults occur in the network then we may opt for more redundancy by using larger $r_{i,j}$, *cf.* § 2.4.)

2.3.1. Examples of rendez-vous matrices associated with both well-known and lesser known strategies.

1. *Broadcasting.* The server stays put and client looks everywhere:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	1	1	1	1	1	1	1	1
S	2	2	2	2	2	2	2	2	2	2
e	3	3	3	3	3	3	3	3	3	3
r	4	4	4	4	4	4	4	4	4	4
v	5	5	5	5	5	5	5	5	5	5
e	6	6	6	6	6	6	6	6	6	6
r	7	7	7	7	7	7	7	7	7	7
s	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9

2. *Sweeping.* The client stays put and the server looks for work:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	2	3	4	5	6	7	8	9
S	2	1	2	3	4	5	6	7	8	9
e	3	1	2	3	4	5	6	7	8	9
r	4	1	2	3	4	5	6	7	8	9
v	5	1	2	3	4	5	6	7	8	9
e	6	1	2	3	4	5	6	7	8	9
r	7	1	2	3	4	5	6	7	8	9
s	8	1	2	3	4	5	6	7	8	9
	9	1	2	3	4	5	6	7	8	9

3. *Centralized name server.* All services post at node 3 and all clients query for services at node 3:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	3	3	3	3	3	3	3	3	3
S	2	3	3	3	3	3	3	3	3	3
e	3	3	3	3	3	3	3	3	3	3
r	4	3	3	3	3	3	3	3	3	3
v	5	3	3	3	3	3	3	3	3	3
e	6	3	3	3	3	3	3	3	3	3
r	7	3	3	3	3	3	3	3	3	3
s	8	3	3	3	3	3	3	3	3	3
	9	3	3	3	3	3	3	3	3	3

4. *Truly distributed name server.* All nodes are used equally often as *rendez-vous* node:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	1	1	2	2	2	3	3	3
S	2	1	1	1	2	2	2	3	3	3
e	3	1	1	1	2	2	2	3	3	3
r	4	4	4	4	5	5	5	6	6	6
v	5	4	4	4	5	5	5	6	6	6
e	6	4	4	4	5	5	5	6	6	6
r	7	7	7	7	8	8	8	9	9	9
s	8	7	7	7	8	8	8	9	9	9
	9	7	7	7	8	8	8	9	9	9

5. *Hierarchically distributed name server.* Links for nodes lower in the hierarchy are served by *rendez-vous* nodes higher in the hierarchy. The nodes are hierarchically ordered by 1,2,3<7; 4,5,6<8; 7,8<9:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	7	7	7	9	9	9	9	9	9
S	2	7	7	7	9	9	9	9	9	9
e	3	7	7	7	9	9	9	9	9	9
r	4	9	9	9	8	8	8	9	9	9
v	5	9	9	9	8	8	8	9	9	9
e	6	9	9	9	8	8	8	9	9	9
r	7	9	9	9	9	9	9	9	9	9
s	8	9	9	9	9	9	9	9	9	9
	9	9	9	9	9	9	9	9	9	9

6. *Distributed name server* for the binary 3-cube topology. The node addresses are the 3-bit addresses of the corners of the cube. For all $a,b,c \in \{0,1\}$, $P(abc) = \{axy \mid x,y \in \{0,1\}\}$ and $Q(abc) = \{xbc \mid x \in \{0,1\}\}$:

		Clients							
		000	001	010	011	100	101	110	111
	000	000	001	010	011	000	001	010	011
S	001	000	001	010	011	000	001	010	011
e	010	000	001	010	011	000	001	010	011
r	011	000	001	010	011	000	001	010	011
v	100	100	101	110	111	100	101	110	111
e	101	100	101	110	111	100	101	110	111
r	110	100	101	110	111	100	101	110	111
s	111	100	101	110	111	100	101	110	111

2.3.2. Lower Bound

There are n possible *rendez-vous* nodes and n^2 elements in R. By choice of P, Q the algorithm distributes the load of being a *rendez-vous* node over the nodes in the network. It is sometimes preferable to distribute the load unevenly. For instance, in the very large networks with millions of processors which are now envisioned, $\sqrt{n}$ message passes is just too much because n is so large. In hierarchical networks (Example 5) the number of message passes for a match-making instance can be as low as $\log n$. This means that some nodes are used very often as *rendez-vous* node, and others very seldom or not at all. A combination of hierarchical and local posting may also be useful.

Let the *rendez-vous* matrix R have n^2 node entries, constituted by $k_i \geq 0$ copies of each node i, $1 \leq i \leq n$. Clearly,

$$\sum_{i=1}^{n} k_i = n^2 \ , \tag{M2}$$

To match a server at node i with a client at node j, the server sends messages to all nodes in $P(i)$ and the client sends messages to all nodes in $Q(j)$. So, all in all, the *number of message passes* $m(i,j)$ involved in this *match-making instance* is given, in a complete network, by

$$m(i,j) = \#P(i) + \#Q(j) \quad . \tag{M3}$$

In the examples above we have seen that, for different pairs i,j, the number of message passes $m(i,j)$ for a match-making instance can, in a single *match-making strategy*, range all the way from a minimum of 2 to n, and beyond. We determine the quality and complexity of a match-making strategy by the *minimum* of $m(i,j)$, the *maximum* of $m(i,j)$ and, above all, the *average* of $m(i,j)$, for $1 \leqslant i,j \leqslant n$.

Definition. The *average* number of message passes $m(n)$ of the given match-making strategy (which is determined by the *rendez-vous* matrix R) is:

$$m(n) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} m(i,j) \quad . \tag{M4}$$

We now proceed to derive an exact lower bound on $m(n)$ expressed in terms of the number k_i of times node i occurs in R, i.e., is used as *rendez-vous* for a pair of nodes ($1 \leqslant i \leqslant n$).

Proposition 1. *Consider the rendez-vous matrix R as defined. Then the average value $\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j)$ is bounded below by:*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) \geqslant \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 \tag{M5}$$

Proof. Let r_i $[c_i]$ be the number of different nodes in row i [column i] ($1 \leqslant i \leqslant n$). Then

$$r_i = \# \bigcup_{j=1}^{n} r_{i,j} \quad \& \quad c_j = \# \bigcup_{i=1}^{n} r_{i,j} \quad . \tag{1}$$

Let R_i be the number of different rows containing node i, and let C_i be the number of different columns containing node i ($1 \leqslant i \leqslant n$). Let $\rho_{i,j} = 1$ if node i occurs in row j and else $\rho_{i,j} = 0$, and let $\gamma_{i,j} = 1$ if node i occurs in column j and else $\gamma_{i,j} = 0$, ($1 \leqslant i,j \leqslant n$). Then,

$$\sum_{j=1}^{n} r_j = \sum_{j=1}^{n} \sum_{i=1}^{n} \rho_{i,j} = \sum_{i=1}^{n} R_i \tag{2}$$

$$\sum_{j=1}^{n} c_j = \sum_{j=1}^{n} \sum_{i=1}^{n} \gamma_{i,j} = \sum_{i=1}^{n} C_i \quad .$$

Clearly, for all i ($1 \leqslant i \leqslant n$) we have

$$R_i C_i \geqslant k_i \quad . \tag{3}$$

Furthermore, since

$$k_j R_i^2 - 2\sqrt{k_i k_j} R_i R_j + k_i R_j^2 = (\sqrt{k_j} R_i - \sqrt{k_i} R_j)^2 \geqslant 0 \quad ,$$

for all i,j ($1 \leqslant i,j \leqslant n$), we obtain immediately:

$$\frac{k_j R_i}{R_j} + \frac{k_i R_j}{R_i} \geqslant 2\sqrt{k_i k_j} \quad ,$$

from which it follows that:

$$\sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_j R_j^{-1} \geqslant \sum_{i=1}^{n} \sum_{j=1}^{n} \sqrt{k_i k_j} \quad . \tag{4}$$

Hence,

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) &\geqslant \sum_{i=1}^{n} \sum_{j=1}^{n} r_i c_j \quad \text{(by (M1) \& (1))} \\
&= \sum_{i=1}^{n} r_i \times \sum_{j=1}^{n} c_j \\
&= \sum_{i=1}^{n} R_i \times \sum_{j=1}^{n} C_j \quad \text{(by (2))} \\
&\geqslant \sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_j R_j^{-1} \quad \text{(by (3))} \\
&\geqslant \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 \quad \text{(by (4)),}
\end{aligned}$$

which yields the Proposition. □

The constraints (M1)-(M5) imply a lower bound trade-off between the number of message passes (and nodes) for posting a server's (port, address) and the number of message passes due to a client querying nodes for the whereabouts of services.

> We can adjust the distributed match-making strategy to the relative frequency of these happenings, so as to minimize the weighted overall number of messages. For instance, if the average call for a service at i by a client at j occurs $\alpha_{i,j}$ times more often than the average posting of a service available at i, then we may want to minimize $m(n)$ replacing (M3) by (M3′):
>
> $$m(i,j) = \#P(i) + \alpha_{i,j} \#Q(j) \quad . \tag{M3′}$$

Proposition 1 immediately gives us a lower bound on the average number of messages involved with a *rendez-vous*:

Proposition 2. *For a complete n-node network and any Shotgun Locate strategy, with the k_i's as defined above, the average number $m(n)$ of message passes (c.q., distinct nodes accessed) to make a match is*

$$m(n) \geqslant \frac{2}{n} \sum_{i=1}^{n} \sqrt{k_i} \quad .$$

Proof. Assume, by way of contradiction, that the Proposition is false, that is,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (r_i + c_j) = n \sum_{i=1}^{n} (r_i + c_i) < 2n \sum_{i=1}^{n} \sqrt{k_i} \quad .$$

Then,

$$\sum_{i=1}^{n} r_i \sum_{i=1}^{n} c_i < \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 \quad ,$$

which contradicts Proposition 1. □

It is not difficult to see that Propositions 1 and 2 hold *mutatis mutandis* for nonsquare matrices R, that is, for networks where some nodes can host only servers and other nodes perhaps only clients.

2.3.3. Truly Distributed Match-Making, Centralized Link-Server

Propositions 1 and 2 specialize to the Corollary below for $k_1 = k_2 = \cdots = k_n = n$, the *truly distributed case.* Here, each node occurs equally often as *rendez-vous* node in matrix R, and hence carries an equal load of the work.

Corollary. *Consider the rendez-vous matrix* R *as defined, for* $k_1 = k_2 = \cdots = k_n = n$. *Then:*

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) \geq n \quad ,$$

$$m(n) \geq 2\sqrt{n} \quad .$$

This lower bound we saw before in the probabilistic approach. Another choice of the k_i's gives:

Corollary. *For* $k_2 = k_3 = \cdots = k_n = 0$ *and* $k_1 = n^2$, *that is, there is a centralized name server, we obtain:*

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) \geq 1 \quad ,$$

$$m(n) \geq 2 \quad .$$

2.3.4. Upper Bound for Complete Networks

For complete networks the above lower bounds on the number of message passes for match-making are about sharp. For instance:

Proposition 3. *For the truly distributed case arrangements can be constructed such that the lower bounds are (nearly) matched by upper bounds. Viz., for each complete network there exists functions* P, Q *such that, for all* $1 \leq i,j \leq n$, $\#P(i)\#Q(j) \approx n$, $\#P(i)+\#Q(j) \approx 2\sqrt{n}$, *and* $k_i \approx n$.

Proofsketch. Arrange the *rendez-vous* matrix R as a checker board consisting of (as near as possible) $\sqrt{n} \times \sqrt{n}$ squares, or nearly squares, of about n entries each. Each square is filled with about n copies of one unique node out of the n nodes, a different one for each square; *cf.* Example 4. □

Proposition 4. *Let* R *be the rendez-vous matrix for an* n*-node network. Let* k_i $(1 \leq i \leq n)$ *be the multiplicity of node* i *in* R, *and let* $m(n)$ *be the average match-making cost associated with* R. *We can lift this strategy to a* $4n$*-node network by constructing a* $4n \times 4n$ *rendez-vous matrix* R' *with* $k_i' = 4k_{i \bmod n}$ *the multiplicity of node* i *in* R' $(1 \leq i \leq 4n)$ *and* $m'(4n) = 2m(n)$ *the associated average match-making cost.*

Proof. Replace each entry $r_{i,j}$ of R by a 2×2 submatrix consisting of 4 copies of $r_{i,j}$. The resulting $2n \times 2n$ matrix is M. Let R_i $(i = 1,2,3,4)$ be four, pairwise element disjoint, isomorphic copies of M. Consider the $4n \times 4n$ matrix R':

$$R' = \begin{bmatrix} R_1 & R_2 \\ R_3 & R_4 \end{bmatrix} \quad .$$

The number of distinct nodes in R' is 16 times that in R and $k_i' = 4k_{i \bmod n}$ $(1 \leq i \leq 4n)$. It is easy to see that the $(2i \bmod 2n)$th column [row] of R' contains twice as many distinct nodes as the $(i \bmod n)$th column [row] of R $(1 \leq i \leq 2n)$. Therefore, the average match-making cost associated with R' is $m'(4n) = 2m(n)$. □

The most *inefficient* match-making strategy is $P(i) = Q(j) = U$ $(1 \leq i,j \leq n)$, yielding $m(n) = 2n$.

2.3.5. Upper Bound for Non-Complete Networks

The topology of a network $G = (U,E)$ determines the overhead in message passes needed for routing a message to its destination. For the complete networks we have considered, the number of message passes $m(i,j)$ for a match-making between a service at node i and a client at node j equals $\#P(i) + \#Q(j)$. If the subgraph induced by the sets $P(i)$, $Q(j)$ $(1 \leq i,j \leq n)$ is connected, and $i \in P(i)$ and $j \in Q(j)$, and we broadcast the messages over spanning trees in these subgraphs, then the number of message passes $m(i,j)$ equals the number of addressed nodes $\#P(i) + \#Q(j)$. Otherwise, there is an overhead $m(i,j) - \#P(i) - \#Q(j) > 0$ of message passes for routing messages from i,j to $P(i)$, $Q(j)$. In designing distributed name servers for non-complete networks, the achievable message pass efficiency of match-making very much depends on how far we can reduce this overhead. For this reason, in a *ring network*, *no* match-making algorithms can do significantly better than broadcasting (i.e., $m(n) \in \Omega(n)$).

2.4. Robustness, Fault-Tolerance, and Efficiency

In computer networks, and also in multiprocessor systems, the communication algorithms must be able to cope with faulty processors, crashed processors, broken communication links, reconfigured network topology and similar issues. A centralized name server (Example 3) is very *efficient*, but if its host crashes the whole network fails. It is one of the advantages of truly distributed algorithms that they may continue in the presence of faults. With respect to implementing the name server, we can distinguish two distinct criteria for robustness.

- The name server should be *distributed* in the sense that no number of node crashes, which leaves a surviving network, can prevent surviving clients from locating surviving servers offering a desired service (for instance, by first moving to another address). This rules out a centralized name server, but the distributed Examples 1, 2, 4, 5, 6 are fine. It is lack of robustness according to this criterion that makes the efficient Hash Locate (last section) so fragile.
- The name server should be *redundant* in the sense that no number of node crashes can prevent a client at a surviving node from locating a service offered at a surviving node. For example, the Shotgun algorithm expounded above, may be locally incapacitated by a *rendez-vous* node crashing. We can remedy this situation by choosing P and Q such that, for all $1 \leq i,j \leq n$,

  $$\#(P(i) \cap Q(j)) \geq f + 1 \quad ,$$

  where f is the maximal number of faults at any time in the network. (There remains of course the problem of how, or whether it is still possible, to route the match-making messages to their destinations in the surviving subnetwork.) The safest solution is obviously $P(i) \cap Q(j) = U$

($1 \leq i,j \leq n$). This criterion holds equally for Shotgun Locate and Hash Locate.

Robustness is *inefficient* and has a price tag in number of message passes per match-making instance. That question is not addressed in this paper.

3. Implementations in Particular Networks

We assume that each node has a table containing the names of all other nodes together with the minimum cost to reach them and the neighbor at which the minimum cost path starts. In [4] a construction is given to divide every connected graph in $O(\sqrt{n})$ disjoint *connected* subgraphs of $\leq \sqrt{n}$ nodes each. Number the nodes in each subgraph 1 through $\sqrt{n}$ (if necessary, divide the excess numbers over the nodes). Each node i has a table containing the route to the next (adjacent) node i. In the worst case such a path consists of $2\sqrt{n}$ message passes. (Each of the connected subgraphs contains at most $\sqrt{n}$ nodes. The shortest path, between the two nodes labelled i in two adjacent connected subgraphs, is therefore not longer than $2\sqrt{n}$.)

Server's Algorithm. A server at the node labelled i in one of the subgraphs communicates its (port, address) to all nodes i in the remaining $O(\sqrt{n})$ subgraphs . It follows from above that this takes $O(n)$ message passes. Size $O(\sqrt{n})$ suffices for the cache of each node.

Client's Algorithm. A client broadcasts for a service (along a spanning tree) in the subgraph where it resides. This takes at most $\sqrt{n}$ message passes.

Under the practical assumption that clients need to locate services usually far more frequently than servers need to post (port, address), this scheme is fairly optimal. Additionally, the caches are kept to a moderate size. Moreover, in practice, many store-and-forward networks will require but $O(\sqrt{n})$ message passes on the average to broadcast over the required subsets of $\sqrt{n}$ nodes of the server's algorithm. All this suggests that in most networks using this method the average number of message passes per match-making instance can be substantially less than the order n figure. In the remainder of this section we look at match-making in some networks with specific topologies.

3.1. Manhattan Networks

The network is laid out as a $p \times q$ rectangular grid of nodes. Post availability of a service along its row and request a service along the column the client is on. Caches are of size $O(q)$ and number of message passes for each match-making instance is $O(p+q)$. For $p=q$ we have $m(n)=2\sqrt{n}$ and caches of size $\sqrt{n}$. For the 9-node network below,

```
1 - 2 - 3
|   |   |
4 - 5 - 6
|   |   |
7 - 8 - 9
```

the *rendez-vous* matrix looks as follows:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	2	3	1	2	3	1	2	3
S	2	1	2	3	1	2	3	1	2	3
e	3	1	2	3	1	2	3	1	2	3
r	4	4	5	6	4	5	6	4	5	6
v	5	4	5	6	4	5	6	4	5	6
e	6	4	5	6	4	5	6	4	5	6
r	7	7	8	9	7	8	9	7	8	9
s	8	7	8	9	7	8	9	7	8	9
	9	7	8	9	7	8	9	7	8	9

Wrap-around versions of the method can also be used in cylindrical networks, or torus-shaped networks. It is, in fact, the method used in the torus-shaped Stony Brook Microcomputer Network [5]. In the obvious generalization to d-dimensional meshes the method takes $m(n)=2n^{(d-1)/d}$ message passes.

3.2. Multidimensional Cubes

The network $G=(U,E)$ is a d-dimensional cube with U the set of nodes of the cube with addresses of d bits and E the set of edges which connect nodes of which the addresses differ in a single bit. $n=\#U=2^d$ and $\#E=d2^{d-1}$. Assume that d is even.

Server's Algorithm. A server at an address $s=s_1s_2 \cdots s_d$ broadcasts its (port, address) along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$P(s) = \{a_1a_2...a_{\frac{d}{2}}s_{\frac{d}{2}+1}...s_d \mid a_1,...,a_{\frac{d}{2}} \in \{0,1\}\} .$$

Client's Algorithm. A client at an address $c=c_1c_2 \cdots c_d$ broadcasts its query along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$Q(c) = \{c_1c_2...c_{\frac{d}{2}}a_{\frac{d}{2}+1}...a_d \mid a_{\frac{d}{2}+1},...,a_d \in \{0,1\}\} .$$

For each pair $s,c \in \{1, \ldots, n\}$ the *rendez-vous* node is given by

$$P(s) \cap Q(c) = \{c_1c_2...c_{\frac{d}{2}}s_{\frac{d}{2}+1}...s_d\} .$$

The number of message passes is the same for each server-client pair, and therefore

$$m(n) = \#P(s)+\#Q(c) = 2\sqrt{n} .$$

The nodes need $\sqrt{n}$-size caches. Variants of the algorithm are obtained by splitting the corner address used in the algorithm not in the middle but in pieces of ϵd and $(1-\epsilon)d$ bits. *Cf.* Example 6. For instance, to adapt the method to take advantage of relative immobility of servers, to get lower average. Excessive clogging at intermediate nodes may be prevented by sending messages to a random address first, to be forwarded to their true destination second [12].

3.3. Fast Permutation Networks

For various reasons [1] fast permutation networks like the *Cube-Connected Cycles* network are important interconnection patterns. An algorithm similar to that of the d-dimensional cube yields, appropriately tuned, for an n-node CCC network caches of size $\sqrt{n / \log n}$ and $m(n) \in O(\sqrt{n \log n})$.

3.4. Projective Plane Topology.

The projective plane $PG(2,k)$ has $n = k^2 + k + 1$ points and equally many lines. Each line consists of $k + 1$ points and $k + 1$ lines pass through each point. Each pair of lines has exactly one point in common. A server s posts its (port, address) to all nodes on an arbitrary line incident on its host node. A client c queries all nodes on an arbitrary line incident on its own host node. The common node of the two lines is the *rendez-vous* node. A ` n size cache for each node suffices. Since the nodes are symmetric, it is easy to see that

$$m(n) = \#P(s) + \#Q(c) = 2(k+1) \approx 2\sqrt{n} \quad .$$

This combination of topology and algorithm is resistant to *failures* of lines, provided no point has all lines passing through it removed.

3.5. Hierarchical Networks

Local-area networks are often connected, by *gateway* nodes, to wide-area networks, which, in turn, may also be interconnected. Locating services and objects in such network hierarchies is bound to become an acute problem.

Service naming preferably should be resolved in a way which is machine-independent and network-address-independent. Consequently, ways will have to be found to locate services in very large networks of hierarchical structure. There, the truly distributed $\sqrt{n}$ solutions to the locate problem are not acceptable any more. Fortunately, in network hierarchies, it can be expected that local traffic is most frequent: most message passing between communicating entities is intra-host communication; of the remaining inter-host communication, most will be confined to a local-area network, and so on, up the network hierarchy. For locate algorithms these statistics for the locality of communication can be used to advantage. When a client initiates a locate operation, the system first does a local locate at the lowest level of the network hierarchy (e.g., inside the client host). If this fails, a locate is carried out at the next level of the hierarchy, and this goes on until the top level is reached.

Assume that a level i network connects n_i level $i-1$ networks through n_i gateways, for each $1 < i \leq k$ (or basic nodes, at the lowest level 0 for $i=1$). Assume also that the n_i gateway hosts compose a level i network with a topology which allows thrifty truly distributed match-making with $2\sqrt{n}$ message passes per match, for all $i \geq 1$.

Server's Algorithm. A server posts its (port, address) by selecting $\sqrt{n_i}$ gateways, connecting level $i-1$ level networks in a level i network, at each level i of the hierarchy, on a path from its host node to the highest level network, to advertise their location.

Client's Algorithm. Similarly, at each level i on a path from its host node to the highest level network, a client's locate in a network of that level can be done in $O(\sqrt{n_i})$ message passes.

This gives an average message pass complexity $m(n) \in O(\sum_{i=1}^{k} \sqrt{n_i})$ for a hierarchical network with a total of $n \leq \prod_{i=1}^{k} n_i$ nodes. Assuming that all n_i's equal a fixed α, the number of levels in the hierarchy is k, and the total number of nodes in the network is $n = \alpha^k$ then the message pass complexity of the locate is $m(n) \in O(k\sqrt{\alpha})$. Therefore,

$$m(n) \in O(kn^{\frac{1}{2k}}) \quad .$$

Having the number k of levels in the hierarchy depend on n, the minimum value

$$m(n) \in O(\log n)$$

is reached for $k = ½\log n$. This message pass complexity is much better than $\Omega(\sqrt{n})$, but the cache size towards the top of the hierarchy increases rapidly. Essentially, the cache of a node may need to hold as many (port, address)'s as there are nodes in the subtree it dominates. In some cases this can be avoided. For in a network hierarchy, as we have sketched, services are often exclusively accessed by local clients.

In the *Amoeba* distributed operating system, for instance, even the operating system itself is accessed just like any other service [11]. "Operating System Service" is thus a local service, useful only to local clients. Clients on other hosts must use similar services, local to their host. The *Amoeba* system provides a way for services to restrict the availability of the service they offer to some local group of processes, the processes within the host where the service resides, the processes within the local-area network of the service, within the campus network, etc. This last model seems the most likely model for the interaction between clients and services. Nearly every service will be a local service in some sense, with only few services being truly global. Under these assumptions, the burden of the processing of locate postings and requests can be distributed more or less evenly over the hosts at each level of the network hierarchy. This is essentially the generalization presented later in the section on Hash Locate.

3.6. Existing Networks

Many wide-area computer networks are not completely designed at the outset but grow and change dynamically. Yet one can identify common characteristics.

- The network resembles an undirected tree with a core in which we can imagine the root, and with some additional edges thrown in. It appears that UUCPnet (the anarchistic network connecting most UNIX* systems) has this form in the sense that the number of extra edges thrown in are not more than the the number of nodes in a spanning tree. The extra edges would typically occur between geographically near nodes.

* UNIX is a trademark of Bell Laboratories.

- The degree of the nodes should not be to large. Ideally bounded by a constant. Yet nodes nearer to the core of the tree tend to be of higher degree. Compare *backbone* sites, *feeder* sites and *terminal* sites in UUCPnet. The hierarchy of the nodes towards the core is very pronounced as can be seen in the table. The degree of super-backbone sites like *ihnp4* is over 600, of backbone sites like *decvax* 40 and *mcvax* 45, and a feeder site like *sdcsvax* is 17. Terminal sites like *ace* have degree 1.
- The network is planar to a large extent. This reflects the geographical cost factor but also the tree aspect mentioned above. Thus, the ARPAnet, to a large extent predesigned, is approximately planar and even the chaotic UUCPnet is not too unplanar.

In the table below we have collected some statistics about the state of the known sites of UUCPnet at August 15, 1984. The total number of sites of UUCPnet is 1916 and of EUnet (European part) 153. The total number of edges in UUCPnet is 3848 and in EUnet 211. The degree of the nodes varies between the unlikely number 0 (one such node is appropriately named *loyalist*) and 641 (which is *ihnp4*, in real life AT&T in Naperville). In the table below we list the number of nodes having a given degree.

#sites	degree	#sites	degree
25	0	3	25
840	1	1	27
384	2	2	28
207	3	2	30
115	4	2	32
83	5	1	33
71	6	2	34
32	7	1	35
29	8	2	36
11	9	1	37
17	10	1	38
5	11	1	39
7	12	1	40
14	13	1	42
10	14	1	43
6	15	1	44
2	16	3	45
2	17	1	46
3	18	1	47
3	19	1	52
3	20	2	63
3	21	1	70
4	22	1	471
3	23	1	641
3	24		

Table

Let us consider trees as described above. The number of nodes in the balanced tree is n, the number of levels is l with the root at level l and the leaves at level 0, and the degree of nodes at the i-th level is $d(i)$. Then a 'factorial' relation holds:

$$d(l)d(l-1) \cdots d(1) = n \ .$$

Setting $d(l) = cl^{1+\epsilon}$, for constants $c, \epsilon > 0$, yields $c^l (l!)^{1+\epsilon} = n$. By Stirling's approximation, we get after some calculation:

$$l \sim \frac{\log n}{(1+\epsilon)\log\log n} \ .$$

If the exponent $1+\epsilon$ in the expression for $d(m)$ is doubled then the depth of the tree is halved for the same number of nodes.

Setting $d(l) = c2^{\epsilon l}$, for constants $c, \epsilon > 0$ yields:

$$n = c^l 2^{\sum_{i=1}^{l} \epsilon i} = c^l 2^{\frac{\epsilon}{2} l^2} \ .$$

Therefore,

$$l = \frac{\sqrt{\log^2 c + 2\epsilon \log n} - \log c}{\epsilon}$$

(The logarithms have base 2.) If ϵ is quadrupled then the depth of the tree is halved for the same number of nodes.

The strategy in such trees can be simple: all services advertise at the path leading to the root of the tree, and similarly the clients request services on the path to the root of the tree. Then the average number of message passes used for each match-making instance, is $m(n) \in O(l)$. The cache at each node needs to be of the order of the number of elements in the subtree of which it is the root. For smaller caches the older and less used entries can be discarded in favour of new ones, leading to a Lighthouse Locate like algorithm (see below). It may seem that such large caches are unrealistic and that, anyway, in distributed networks all nodes should be symmetric. However, even in a genuinely distributed and anarchistically growing network as UUCPnet a hierarchy of nodes develops according to the node degree (number of links with other nodes in the network). This points to the fact that nodes higher in the hierarchy must dedicate more computing power and memory to running the network. Hence it is not unrealistic to have the cache size increase for nodes higher in the hierarchy.

4. Lighthouse Locate

We imagine the processors as discrete coordinate points in the 2-dimensional Euclidean plane grid spanned by $(\epsilon,0)$ and $(0,\epsilon)$. The number of servers satisfying a particular port in an n-element region of the grid has expected value sn for some fixed constant $s > 0$.

Server's Algorithm. Each server sends out a random direction beam of length l every δ time units. Each trail left by such a beam disappears after d time units. That is, a node discards a (port, address) posting after d time units. Assume that the time for a message to run through a path of length l is so small in relation to d that the trail appears and disappears instantaneously.

Client's Algorithm. To locate a server, the client beams a request in a random direction at regular intervals. Originally, the length of the beam is l and the intervals are δ. After e unsuccessful trials, the client increases its effort by doubling the length of the inquiry beam and the intervals between them ($l \leftarrow 2l$ & $\delta \leftarrow 2\delta$). And so on.

Another possibility is to govern the length of the locate beam (and its duration) by the sequence

$$121312141213121512131214121312161213 12 \cdots$$

Here the length of the locate beam is il once in each interval of 2^i trials. (This sequence is sequence 51 in Sloane's catalogue [9].) The schedule can conveniently be maintained by a binary counter: the position i of the most significant bit changed by the current unit increment indicates the current beam length il. This schedule has the additional profit that the servers which drift nearer to the client are located with less time-loss. Note that in a sequence of 2^k trials there are 2^{k-i} length il trials ($1 \leqslant i \leqslant k$).

Before the locate method for the euclidean plane can be converted into a practical algorithm for locating services it is necessary to find ways of mapping point-to-point networks onto the euclidean plane in such a way that the euclidean plane algorithm can be converted into an algorithm for a point-to-point network. Fortunately, such a mapping can often be found. Most point-to-point networks have routing tables that tell each node which outgoing arc to use to get a message to its destination. In [3] these tables are used back-to-front to broadcast messages over the network in near optimal fashion. We can use these tables back-to-front to simulate sending messages along "a straight line" of certain length. The technique is as follows.

A client (or server) wishing to send a beam of length k (using message passes as the unit of length) chooses a random outgoing arc and sends the message along it to its neighbor. This neighbor, upon reception of such a message decreases the hop count (in the message) by 1, and sends the message on any one outgoing arc that is used to send messages *from* the node at the *other end* of the arc *to* the *original* client (or server) where the beam started from. And so on, until the hop count reaches 0.

5. Hash Locate and Beyond

Let in a given network $G=(U,E)$ the set of ports (i.e., types of services available) be Π. We can define the functions P and Q like in the Shotgun Locate but using the port identities as well:

$$P,Q\colon U \times \Pi \rightarrow 2^U$$

If we are dealing with a very large network, where it is advantageous to have servers and clients look for nearby matches, we can hash a service onto nodes in neighborhoods. A neighborhood can be a local network, but also the network connecting the local networks, and so on. Therefore, such functions can be used to implement the idea of certain services being local and others being more global (*cf.* the section on hierarchically structured networks) thus balancing the processing load more evenly over the hosts at each level of the network hierarchy. Like Shotgun Locate, the Hash Locate below is a specialization of this more general method.

In *Hash Locate* we construct hash functions that map service names onto network addresses. That is,

$$P,Q\colon \Pi \rightarrow 2^U \quad \& \quad P=Q.$$

This technique 'is very efficient. Each server s posts its (port, address) at the node(s) $P(\pi)$, if π is the port of s, and each client in need for a service at port π queries the node(s) in $P(\pi)$. Apart from redundancy for fault-tolerance, clients and servers need only use one network node each in every match-making. (Clearly, the *rendez-vous* matrix must be interpreted differently in this setting.) Provided the hash function is well-chosen, it distributes the burden of the locate work over the network. It suffers from the drawback that, if nodes are added to the network, the hash function must be changed to incorporate these nodes in the set of potential *rendez-vous* nodes. Moreover, if all *rendez-vous* nodes for a particular service crash then this *takes out completely* that particular service from the entire network. If the service is indispensable, the entire network crashes. In this sense Hash Locate is far more vulnerable to node crashes than the more distributed versions of Shotgun Locate. Examples 1, 2 and 3 may also be viewed as borderline examples of Hash Locate. Examples 4, 5 and 6 are not Hash Locate methods, since Hash Locate cannot be distributed in this genuine sense.

Two obvious approaches can make Hash Locate more robust for node crashes. First, the hash function can map a service name onto many different network addresses for added reliability. Second, when the *rendez-vous* node for a particular service is down, rehashing can come up with another network address to act as a backup *rendez-vous* node. It then becomes necessary that services regularly poll their *rendez-vous* nodes to see if they are still alive.

References

[1] Broomel, G. and J.R. Heath, "Classification categories and historical development of circuit switching topologies," *ACM Computing Surveys*, vol. 15, pp.95-133, 1983.

[2] Dalal, Y.K., "Broadcast Protocols in Packet-Switched Computer Networks", Ph.D. Thesis, Stanford University, April 1977.

[3] Dalal, Y.K. and R. Metcalfe, "Reverse path forwarding of broadcast packets," *Communications of the ACM,*, vol. 21, pp.1040-1048, 1978.

[4] Erdös, P., L. Gerëncser, and A. Maté, "Problems of graph theory concerning optimal design," pp. 317-325 in Colloquium Math. Soc. Janos Bolyai 4, ed. P. Erdös, V.T. Sós, North-Holland Publishing Company, Amsterdam (1970).

[5] Gelernter, D. and A.J. Bernstein, "Distributed communication via a global buffer," pp. 10-18 in Proceedings 1th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1982).

[6] Liskov, B. and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, pp.50-59, 1974.

[7] Needham, R. M. and A. J. Herbert, *The Cambridge Distributed Computer System.* Addison-Wesley, 1982.

[8] Seitz, Ch.L., "The cosmic cube," *Communications of the Ass. Comp. Mach.*, vol. 28, pp.22-33, 1985.

[9] Sloane, N.J.A., *A Handbook of Integer Sequences.* New York:Academic Press, 1973.

[10] Tanenbaum, A. S. and S.J. Mullender, "An overview of the Amoeba distributed operating system," *Operating System Review*, vol. 15, pp.51-64, 1981.

[11] Tanenbaum, A. S. and S.J. Mullender, "The design of a capability-based distributed operating system," *Computer Journal*, to appear.

[12] Valiant, L.G., "A scheme for fast parallel communication," *SIAM J. on Computing*, vol. 11, pp.350-361, 1982.