Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender, P.M.B. Vitányi

Distributed match-making for processes in computer networks
(preliminary version)

Department of Computer Science Report CS-R8424 December

Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Distributed Match-Making for Processes in Computer Networks (preliminary version)

Sape J. Mullender

Paul M.B. Vitányi

Centre for Mathematics & Computer Science (CWI)
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

Locating services in a computer network is usually done by broadcasting "where are you" messages. In many networks this is an efficient method, because the network medium is itself a broadcast medium. In other networks, such as large store-and-forward networks, broadcasting is considerably more costly than sending a message directly to its destination. Here we examine methods for locating services that are less expensive than broadcasting in terms of message passes or "hops." For these methods we investigate the complexity in terms of needed storage, in terms of message passes and in terms of processing needed. The general problem consists of distributed match-making between processes, such as server processes and client processes, in computer networks. The processes are assumed to be mobile and not have fixed addresses. When the servers assist the clients in getting themselves found, it appears that, in many mesh networks, match-making can be done in $O(\sqrt{N})$ message passes, where N is the number of nodes in the network. Conventional broadcast methods for locating services need a minimum of $O(N)$ message passes to do the broadcast. The theoretical limitations of distributed match-making are established, and the techniques are applied to several network topologies.

1980 Mathematics Subject Classification: 68C05, 68C25. 69C21, 69F21, 69Q12
CR Categories: C.2.1, F.2.2, G.2.2.
Keywords & Phrases: locating objects, locating services, computer networks, network topology.

1. Introduction

We investigate problems of establishing communication between processes without permanent addresses in a distributed environment. Such problems of *distributed match-making of mobile processes* are now emerging in the design of distributed operating systems for computer networks and other multiprocessor systems.

Report CS-R8424
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The service model. The service model is widely accepted as a convenient vehicle for designing distributed computer systems; see, e.g., [TMa]. In this model *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*. 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 on 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.

The network nodes are assumed to maintain a *cache* to store (port, address) pairs of ports and their addresses. In the algorithms presented below this cache plays an important role. The cache can be large or small. We define a cache to be *large* when it is large enough to hold so many (port, address) pairs that it never has to discard one for a server that is still active. A cache is *small* when this is not the case. Entries are made or updated whenever a message is received from a server process with its address, or when the reply from a locate operation is received.

The Problem. Locating services in a computer network is usually done by broadcasting "where are you" messages. In many networks this is an efficient method, because the network medium is itself a broadcast medium. In other networks, such as 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, usually a small subset of the network.

Here we examine methods for locating services that are less expensive than broadcasting in terms of message passes or "hops." For these methods we investigate the complexity in terms of needed storage, in terms of message passes and in terms of processing needed. When the servers give the clients some assistance in getting themselves found it appears that, in many mesh networks, locate can be done in $O(\sqrt{N})$ message passes, where N is the number of nodes in the network. Conventional broadcast methods for locating services need a minimum of $O(N)$ message passes to do the broadcast (*e.g.*, via the minimum spanning tree [Da]). At the other end of the spectrum are what may be called *sweep* methods, where the servers periodically broadcast their whereabouts to all other nodes in the network and the clients wait until an offer of the desired service is made. There is a formal similarity between broadcasting and sweeping by the analogy between a client in need for a service broadcasting for it and a server being free performing a sweep to inform clients waiting

for its service. Here we consider the entire range between broadcasting and sweeping.

The Algorithm. 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, 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 $P_s \cap Q_c \neq \emptyset$, 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 we are broadcasting; if $P_s = U$ and $Q_c = \{c\}$ then we are sweeping. We have called this class of algorithms *Shotgun Locate* algorithms: put so many pheasants in the bushes that the hunter has a good probability of hitting one for the amount of shot he is willing to spend. For large caches we use this shotgun locate variant, for small caches the *lighthouse* locate variant, as explained below.

Outline of the paper. We introduce a new class of distributed algorithms for match-making between client processes and server processes in computer networks. First, a theory for the general problem is developed. We investigate the expected performance of the algorithms in absence of known network topology. 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. This yields a combinatorial lemma which may be interesting in its own right, and results in a lower bound on the trade-off between the number of nodes a server advertises at and the number of nodes a client inquires at. Second, we apply the method to particular networks, both designed networks and spontaneously emerged networks.

Related work. Distributed match-making between *clients* and *servers* will be used in the Amoeba network [TMb]. A similar idea has been used in the Stony Brook Microcomputer Network [GB]. The present paper is the first systematic exploration of distributed match-making.

2. Theory of Distributed Match-Making

Let the number of elements in a given set U (universe) of nodes be n. Let p be the cardinality of $P \subseteq U$, the set of posted-to nodes. Let q be the amount of elements in $Q \subseteq U$, the set of queried nodes. If P and Q are randomly chosen then the probability for any one element of U to be an element of P (Q) is p/n (q/n). If P and Q are chosen independently then the probability for any one element of U to be an element in both P and Q is pq/n^2. Since there are n elements in U, the expected size of $P \cap Q$ is given by

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

If we want to find an element by repeated searches, say by independent choice of Q_1 through Q_k of cardinalities q_1 through q_k, we obtain

$$E(\#(P \cap (Q_1 \cup Q_2 \cdots \cup Q_k)) \leqslant \sum_{i=1}^{k} E(\#(P \cap Q_i))$$

$$= \frac{p}{n} \sum_{i=1}^{k} q_i \ .$$

Consequently, for a single trial to expect success the setup number p and the trial number q must multiply to n. Repeated sampling does not improve matters under the above model. Assuming that $p \approx q$ both p and q need to be of size $\sqrt{n}$ for success. Therefore, without knowing anything of the topology or other characteristics of the network, aiming at a minimal sum of $p+q$, gives as a condition for expected success $p = q = \sqrt{n}$. 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. Knowledge of the structure of the network may improve the situation in two ways:

1. We do not only expect success but the method deterministically yields success.

2. We get by with $p+q < 2\sqrt{n}$ for expected or certain success.

An obvious way to improve this result in both ways is the following: All servers use one special node x to tell about their whereabouts, and all clients ask x for the location of a service. In fact, x acts as a *name server* here. This method is optimal in message passes and is deterministic as well.

Still, this method is not acceptable: All the work in locating nodes is *centralised* in x, effectively limiting the maximum size of the network by the capacity of x. Moreover, when x crashes, the whole system stops working.

In this paper we examine truly *distributed* versions of the locate problem. By this we mean that, if we assume that servers and clients are distributed homogeneously over the network, each node is used equally often as a *rendez-vous* node. This distributes the burden of processing locate operations over the network, and guarantees that a crash in a node can only have a limited effect on the 'locatability' of services. It appears that, under these conditions, expected success can often be turned into certain success, but that $pq \geqslant n$ is a lower bound for certain success. Curiously, deterministic distributed match-making requires at least as many message-passes, on the average, as expected success in probabilistic match-making in the absence of any global network information.

The gain in *message complexity* and *processing complexity* (both $O(p+q)$) over conventional locate algorithms is partially compensated by a loss in *storage complexity*: in conventional broadcast locate algorithms only the server itself needs to know its own address, while in the shotgun algorithms, all nodes that the server *tells* about its whereabouts must remember this also. The storage complexity for each node is thus $O(p)$.

2.1. On the Necessary Average Number of Messages for Certain Success.

In order to prove that server and client must select at least $2\sqrt{n}$ nodes for certain success we assume that every node has an equal probability to host a server or a client, or to act as a *rendez-vous* node. When a server i is matched to a client j the following happens: Server i tells a set P_i of nodes about its location. Client j asks a set Q_j of nodes where server i is. Define the $n \times n$ *rendez-vous* matrix, M, such that the entries $m_{i,j}$ represents the set of *rendez-vous* nodes for a client at node j to find the location of a server at node i. That is,

$$m_{i,j} = P_i \cap Q_j \quad .$$

An optimal shotgun method would have exactly one element in each $m_{i,j}$. There are n possible *rendez-vous* nodes and n^2 elements in M; if every node occurs equally often as a

rendez-vous node then each node must occur n times in the matrix. The *rendez-vous* matrix M therefore has n^2 entries consisting of n copies of node i for $i = 1,2,\ldots,n$. Obviously, $P_i = \bigcup_{j=1}^{n} m_{i,j}$, and $Q_j = \bigcup_{i=1}^{n} m_{i,j}$. We prove that for each algorithm, which chooses sets P and Q such that every server/client combination produces a *rendez-vous* node and the *rendez-vous* nodes are distributed uniformly over the network, the sum of the sizes of P and Q is at least $2\sqrt{n}$, *on the average*. Below, the *average* is the arithmetic mean taken over all n^2 client-server pairs.

Proposition 1. *Consider the rendez-vous matrix M as defined. The average value of the product of the number of different nodes in a column and the number of different nodes in a row is at least n.*

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

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

$$\sum_{j=1}^{n} r_j = \sum_{j=1}^{n}\sum_{i=1}^{n} \rho_{i,j} = \sum_{i=1}^{n} R_i \quad .$$

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

$$R_i C_i \geqslant n \quad . \tag{2}$$

Since $R_i^2 - 2R_iR_j + R_j^2 \geqslant 0$ for all i,j $(1 \leqslant i,j \leqslant n)$:

$$\frac{R_i}{R_j} + \frac{R_j}{R_i} \geqslant 2$$

and therefore,

$$\sum_{i=1}^{n} R_i^{-1} \sum_{j=1}^{n} R_j \geqslant n^2 \quad . \tag{3}$$

Then,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} c_i r_j &= \sum_{i=1}^{n} c_i \times \sum_{i=1}^{n} r_i \\
&= \sum_{i=1}^{n} C_i \times \sum_{i=1}^{n} R_i \quad \text{(by (1))} \\
&\geqslant n \sum_{i=1}^{n} R_i^{-1} \sum_{i=1}^{n} R_i \quad \text{(by (2))} \\
&\geqslant n^3 \quad \text{(by (3))},
\end{aligned}$$

which yields the Proposition. □

The trade-off embodied by this Proposition can be used advantageously in cases where we assume that the average call for services by clients exceeds the average advertising

necessary for the services. That is, for any such algorithm, for any network, under any strategy, we have that *on the average* $\#P\#Q \geq n$. New queries for services by clients are engendered by needs for service, new postings of services are necessary when services move from location. Proposition 1 immediately gives us a lower bound on the average number of messages involved with a *rendez-vous*. This suggests how to adjust the distributed match-making strategy to the relative frequency of these happenings, so as to minimise the overall number of messages.

Proposition 2. *For each n-node network and any distributed match-making strategy which guaranties success, the average number of distinct nodes accessed by client or server to make a match is at least* $2\sqrt{n}$.

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

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

Then, obviously,

$$\sum_{i=1}^{n} r_i \sum_{i=1}^{n} c_i < n^3 \quad ,$$

which contradicts Proposition 1. □

Proposition 3. *The average value of the product of the number of different nodes in a column and a row can not exceed* n^2. *There are arrangements such that this value is reached.*

Proof. The square upper bound is trivial. It is the least upper bound on the average product because in the matrix below each row and each column contain n different nodes. Let the first row of a matrix M consist of $(1, 2, \ldots, n)$, the second row consists of the cyclic permutation $(n, 1, \ldots, n-1)$ of the previous row, and so on. The last row is therefore $(2, 3, \ldots, 1)$. □

2.2. *On the Sufficient Number of Messages for Certain Success*

Suppose $N = pq$, p and q positive integers. We can construct a *rendez-vous* matrix R as a checker board consisting of squares of $p \times q$ items each. Each one of the $p \times q$ squares is filled with N copies of one unique node out of N nodes, a different one for each square. Therefore:

Proposition 4. *For each connected network there exists a strategy such that the number of potential rendez-vous nodes posted at by the service added to [multiplied by] the number of potential rendez-vous nodes queried by the client involved in each match-making is* $2\sqrt{n}$ *[n].*

In the analysis of the necessary number of messages for certain success we have ignored the number of hops necessary to get a message to its destination. Under this idealised cost criterion we can upgrade each strategy for N nodes to a similar cost strategy for $4N$ nodes. For let R be a *rendez-vous* matrix for N nodes. 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':

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

It is obvious that the ith column [row] of R' contains twice as many distinct nodes as the $i \bmod N$th column [row] of R. The number of distinct nodes in R' is $4N$.

The topology of the network determines the overhead in hops needed for routing a message to its destination. This overhead can be minimised as follows. First, it is advantageous if the subgraph induced by the set of target nodes is connected. In that case we broadcast the messages over a minimum spanning tree. If, in addition, the sender is also a target node (or adjacent to a target node) the number of hops can be made equal to the number of posted-to nodes.

3. Applications to Particular Networks

For arbitrary 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 neighbour at which the minimum cost path starts. In [EGM] it is shown that every connected graph can be divided in $O(\sqrt{n})$ connected subgraphs of $\leqslant \sqrt{n}$ nodes each. This suggests the following algorithm. Divide the connected graph in this way. Number the nodes in each subgraph 1 through $\sqrt{n}$. (If necessary, assign two numbers to some 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 $2\sqrt{n}$ hops. (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, along the associated minimum spanning trees is therefore not longer than $2\sqrt{n}$.)

Server. A server at the node labelled i in one of the subgraphs broadcasts its presence to all nodes i in the remaining $O(\sqrt{n})$ subgraphs. This takes $O(n)$ hops in the worst case. The cache of each node needs to be size $\sqrt{n}$.

Client. A client broadcasts for a service (e.g., along a spanning tree) in the subgraph where it resides. This takes $O(\sqrt{n})$ message passes.

Under the practical assumption that clients asking for services are usually far more frequent than servers offering services, this scheme is fairly optimal. Additionally, the caches are kept to a moderate size. In practice, most store-and-forward networks will require about $O(\sqrt{n})$ message passes on the average to broadcast over a set of $\sqrt{n}$ nodes. This suggests that $\#P\#Q$ can be substantially less than $n\sqrt{n}$ for this method in practice.

If the network has a particular topology, then distributed match-making can make use of that to increase its efficiency:

3.1. Manhattan Networks.

The network is laid out as a rectangular grid of $p \times q = n$ nodes. An obvious shotgun algorithm is obtained by letting each server tell the p nodes in its row, while having each client ask the q nodes in its column. It is clear that this algorithm leads to certain success with message complexity and processing complexity $O(p + q)$ and storage complexity $O(p)$. This same algorithm can also be used in cylindrical networks, or torus-shaped networks. It is, in fact, used in the torus-shaped Stony Brook Microcomputer Network [GB].

3.2. Fast Permutation Networks

For various reasons fast permutation networks like the *Shuffle-Exchange* network, the *Fast Fourier Transform* network, and the *Cube-Connected Cycles* network are important interconnection patterns. For instance, the diameter of the resulting graph is $O(\log n)$ and the minimum bisection width (number of edges one needs to cut to divide the network into two approximately equal pieces) is $\Omega(n / \log n)$. Below we investigate one representative: the *Cube-Connected Cycles* network. Imagine the k-dimensional cube with corners named by the obvious k-bit words. Replace each corner node by a k-length cycle such that each resulting node has degree 3. The k-dimensional CCC has $n = k2^k$ nodes and $3k2^{k-1}$ edges. Each node has an address consisting of the binary location of the corner it resides on followed by its address in the k-length cycle. (If k is not a power of 2 then choose the next higher power for the addresses.) Below, the *cycle name* of a node is the address of the cycle it resides on. The *shotgun* locate algorithm proposed here works as follows.

Server. The server sends messages to all cycles designated by the first $k / 2$ bits of its own cycle name and all combinations for the remaining $k / 2$ bits. Thus, each server sends out $2^{k/2}$ messages. Each cycle receiving a server message stores it in the cache of one of its k nodes, say the node of entry in the cycle. Since there are at most n servers, and each sends out at most $2^{k/2}$ messages, while all nodes are basically symmetric to one another, the cache of a node on a cycle needs to contain no more than $2^{k/2} = \sqrt{n / \log n}$ messages.

The total number of message passes per server is estimated as follows. The server wants to deposit an advertisement of its location on every cycle on a corner of a $k / 2$-dimensional hypercube, one of which corner cycles is its own habitat. There are two strategies: (1) sending a separate message to each corner cycle; and (2) sending a single message around along all cycles.

(1) According to strategy (1) the limited broadcast takes a number of hops per message of at most the diameter $O(\log^2 n)$ of the CCC network. So each server sends out $\sqrt{n / \log n}$ messages which altogether take $O(\sqrt{n \log^3 n})$ hops. An advantage of this scheme is that excessive clogging at intermediate nodes may be prevented by sending messages to a random address first to be forwarded to their true destination second [Va].

(2) According to strategy (2) we send the message along an euler path through the $k / 2$-dimensional k-length cycles hypercube. This necessitates the traversing of at most $k \cdot k / 2 \cdot 2^{k/2}$ edges. That is, again $O(\sqrt{n \log^3 n})$ hops. If, however, we have ordered the hypercube such that the eulerpath needs to traverse only a constant number of nodes in a cycle before moving on to another cycle then the number of hops is only $O(\sqrt{n \log n})$. Alternatively, we can broadcast the message over a minimum length spanning tree of all cycles in the hypercube.

Client. The client sends messages to all cycles designated by the suffix $k / 2$ bits of its own cycle name and all combinations for the remaining bits. Once a message has reached its destination cycle it circles the entire cycle to locate a server message. The client can follow the same two strategies as the server and accounts for the same number of hops.

Analysis. The intersection of the set of cycles covered by the server and the set of cycles covered by the client must contain 1 element. Consequently, the algorithm is deterministic.

The same algorithm can be used for networks consisting of the k-dimensional cube using $\sqrt{n}$ size caches. So the CCC network has the advantage of smaller caches, for storing posted server locations, in the local nodes. In both cases the product of the setup number p and the trial number q multiply to the number of nodes in the network, or somewhat more. The excess is due to the fact that a message cannot be at its destination in a single hop, but needs to traverse intermediate nodes. (Only in a complete graph topology can a message be transmitted from each source to each destination in a single hop.) Apart from this, when a message reaches its destination cycle it needs to go to a cache on that cycle which has vacancies. Under an optimal euler path routing in strategy (2) this excess gets reduced. The advantage taken from the network topology is contained in *certainty* and *flexibility* of the method. Variants of the algorithm are obtained by splitting the cycle address used in the algorithm not in the middle but in pieces of ϵk and $(1-\epsilon)k$. Yet another variation is obtained by smaller caches and random selection of hypercubes by both client and server.

3.3. Broadcast Networks

Among the most popular local-area networks are CSMA/CD networks (commonly called *Ethernets*) and token rings. These networks are broadcast networks; that is, every message, even if intended as a point-to-point message is broadcast on the network medium, where the destination node or destination nodes can pick it off. The network interface at every node is thus always on the look-out for incoming messages. The message complexity of broadcast messages and point-to-point messages is the same and between O(1) and O(n), because adding nodes to the network will increase the time needed for broadcast. The processing complexity for point-to-point messages is O(1), while the processing complexity for broadcast messages is O(n).

The best locate algorithm for broadcast networks is usually the conventional broadcast algorithm, optionally combined with a client cache where recently used server locations can be found to decrease the processing complexity.

3.4. Projective Plane Topology

There are many other ways to construct a network topology idealy suited for shotgun locate. 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. An obvious shotgun algorithm is obtained by choosing a network topology according to $PG(2,k)$ and letting each server tell all nodes on an arbitrarily chosen line through its host node about its location and similarly letting each client ask each node on an arbitrarily chosen line through its host node. Since each pair of lines has an intersection, the method yields certain success in O($\sqrt{n}$) message passes. Note that the algorithm works unimpaired if a selection of lines is removed, provided no point has all lines passing through it removed.

3.5. *Hashing*

A special technique for locating services (or objects) is worth mentioning here: Construct a hash function that maps service names or object names onto network addresses. Services and clients use this function to produce a *rendez-vous* node.

This technique is very efficient (clients and servers need only ask or tell one network node) and, 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. This is not a frequent event, however, and it need not even be done every time a node is added. If a node breaks down, it can no longer serve as a *rendez-vous* node.

Two approaches can make the algorithm robust for such events: First, the hash function can be changed to map a service name into two or three different network addresses for added reliability. Second, when a node is used as a *rendez-vous* node and it is down, rehashing will come up with another network address to act as a backup *rendez-vous* node. To this end, services must regularly poll their *rendez-vous* nodes to see if they still work. Note that the hashing algorithm breaks the bound $pq \geqslant n$. This is possible, because the sets chosen by client and server are no longer independent; the client 'knows' the node chosen by the server.

3.6. *Hierarchical Networks*

Local-area networks are often connected to wide-area networks, which, in turn, may also be interconnected. Locating services and objects in such network hierarchies is therefore bound to become an acute problem. The reason that it has not yet become an issue for investigation is that currently each network (and each hierarchical level) has its own specific methods for naming, addressing, locating and routing messages. Most networks use host-specific naming, thus eliminating the locate problem at the cost of portability and flexibility.

However, one may hope that some day service naming and object naming will become machine-independent and network-address-independent. In fact, a research project to encourage just this has been proposed under the European Community's COST-11 programme [Mu]. When this happens, ways will have to found to locate objects and services in very large networks. If these networks are to be manageable, they must have a hierarchical structure. This is not only a way to distribute and localise the burden of network management and network maintenance; it is also a way to combine the possibilities of high-speed local-area network techniques with reliable (but lower-speed) long-haul packet switching network techniques.

Network hierarchies are necessary when distributed systems grow very large. In such networks, even $\sqrt{n}$ solutions to the locate problem may not be acceptable. However, as has been shown, $\sqrt{n}$ is a lower bound on the required number of message passes for distributed locate. Fortunately, in network hierarchies, it can be expected that local traffic occurs most frequently: 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.

In locate algorithms 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. At each level, locate can be done in, say, $O(\sqrt{n_i})$ message passes, where n_i is the number of networks (or nodes, at the lowest level) connected by level i in the hierarchy. The total cost for clients will be

$$c_1 + (1-p_1)(c_2 + (1-p_2)(c_3 + \cdots (1-p_{k-1})c_k)) \cdots),$$

where c_i is the cost of locate at level i of the hierarchy, p_i is the probability of a successful locate at level i and k is the number of levels in the hierarchy

In principle, services are required to ensure their locatability at each level of the hierarchy. They must select, say, $\sqrt{n_i}$ at each level of the hierarchy to advertise their location. This gives a complexity of $\sum_{i=1}^{k} \sqrt{n_i}$ for a network with a total of $\prod_{i=1}^{k} n_i$ nodes. Assuming that all n_i's equal a fixed α and the number of levels in the hierarchy is k, the total number of nodes in the network is $n = \alpha^k$ and the message pass complexity of the locate is $m(n) = k\sqrt{\alpha}$. This yields

$$m(n) = \frac{n^{\frac{1}{2k}} \log n}{\log \alpha} = k n^{\frac{1}{2k}} .$$

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

$$m(n) = \frac{n^{\frac{1}{\log n}} \log n}{2}$$

is reached when $k = \frac{1}{2}\log n$.

This message pass complexity is much better than our previous $\Omega(\sqrt{n})$ algorithms, but the algorithm is no longer truly distributed as defined earlier: the cache at the top of the hierarchy must be enormous. However, in some cases this can be avoided also. In a network hierarchy, as we have sketched, services are often exclusively accessed by local clients. In the *Amoeba* distributed operating system, for instance, which has been designed by one of the authors, even the operating system itself is accessed just like any other service. "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 locale service in some sense, with only few services being truly global. The burden of the processing of locate requests can thus be distributed more or less evenly over the hosts at each level of the network hierarchy.

3.7. 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.

1. The network resembles an directed tree with a core in which we can imagine the root, and with some additional edges thrown in. It appears that USENET 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.
2. The degree of the nodes is not 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 USENET. 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 have degree 1.
3. 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 USENET is not too unplanar.

In the Table we have collected some statistics about the state of the known sites of USENET at August 15, 1984. The total number of sites of USENET is 1916 and of EUNET (European part) 153. The total number of edges in USENET is 3848 and in EUNET 211. The degree of the nodes varies between 0 and 641, the latter is ihnp4 or in real life AT&T in Naperville. In the Table below we list the number of nodes having a given degree.

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

$$d(m)\,d(m-1)\ldots d(1) = n \quad .$$

Setting $d(m) = cm^{1+\epsilon}$, for constants $c, \epsilon > 0$, yields $c^m (m!)^{1+\epsilon} = n$. Therefore, by Stirling's approximation,

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

If the exponent in d is doubled then the depth of the tree is halved for the same number of nodes. Considering for the moment n as a function of m, broadcasting from the root to level i reaches $n(m)/n(i)$ nodes.

Setting $d(m) = c\,2^{\epsilon m}$, for constants $c, \epsilon > 0$ yields

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

Therefore,

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

#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

So 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. This works out as a number of message passes for each such action of the order of the depth of the tree (see above). 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. It may seem that such large caches are unrealistic and that, anyway, in distributed networks all nodes should be symmetric. This, however, is not necessarily the case. In a genuine distributed and unorganised growing network as USENET a hierarchy of nodes develops as indicated by the 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.

3.8. Distributed Match-Making in the Presence of Faults

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. It is one of the advantages of truly distributed algorithms that they may continue in the presence of faults. Due to its probabilistic nature, the Lighthouse algorithm below will not suffer very much from the presence of faults. The Shotgun algorithm expounded above, however, may be locally incapacitated. We can remedy this situation by setting the expected (certain) intersection between P and Q to at least $f+1$, where f is the number of faults at any time in the network. This can be worked out for all of the above algorithms, especially easy for the CCC network Shotgun method. For the Manhattan network we can devise some variations. Add diagonal connections to the axis parallel connections. If one of the messages cannot continue in an axis parallel connection, because of a failure, it chooses a diagonal and continues parallel thereafter. If entering a cul-de-sac, a message may back up and try another direction. (This resembles restructurable wafer-scale gate arrays design as discussed in [GG].)

An interesting fault-tolerant broadcast network consists of two loops with the nodes on the connecting wires between those loops. This network is a three-connected ring network, and two faults cannot prevent a ring broadcast.

4. Variation: Lighthouse Locate in the Euclidian Plane

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 $s>0$.

Servers. 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. Since the time for a message to run through a path of length l is small in relation to d we can assume that the trail appears and disappears instantaneously.

Client. 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.

Analysis. Assume that each point in the plane has as good a chance too host a server as any other. Let the probability of a successful trial per time interval δ be p originally. After e trials or $e\delta$ time the radius of the covered disc is doubled and the area quadrupled, so the expected number of services within reach quadruple too. Nonetheless, the number of trials needed to cover all discs around servers has doubled. This seems to imply that the independent probability to miss stays the same at each trial, viz., $1-p$. Therefore, for time $t = e\sum_{i=1}^{r} i\delta$ there have been er trials and the probability of success is $1-(1-p)^{er}$. So, the probability of success as a function of time turns out to be

$$1-(1-p)^{\sqrt{\frac{2te}{\delta}}}.$$

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

$$12131214121312151213121412131216121312\cdots$$

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 [Sl]; see also [NN].) 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 has the additional profit that the servers which drift nearer to the client are located with less time-loss. To analyse the cost, note that in a sequence of 2^k trials there are 50% length l trials, 25% length $2l$ trials and so on. Thus, $t = \sum_{i=1}^{k} 2^{k-i} i\delta$ for 2^k trials. Consequently, the probability of success is (with c de limit of the series $\sum_{i=1}^{k} 2^{-i} i$)

$$1-(1-p)^{\frac{t}{\delta c}} .$$

Basically, the probability of success in per interval δ stays the same under both regimes. This probability p per interval should be that of [MW], that is, about one-quarter.

Before the locate method for the euclidian 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 euclidian plane in such a way that the euclidian 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 [DM] 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 simple: A client (or server) wishing to send a beam of length k (using hops as the unit of length) chooses a random outgoing arc and sends the message along it to its neighbour. This neighbour, upon reception of such a message decreases the hop count (in the message) to $k-1$, and sends the message on any outgoing arc that is used to send messages in the opposite direction to the client (or server).

If the network can be laid out as a planar graph, client beams and server beams will intersect with a probability that is related to the euclidian plane analysis. In a later version, we shall look at some existing networks and see that most wide-area networks are, in fact, nearly planar, and we shall show some simulation results of the lighthouse locate method.

References

[Da] Dalal, Y.K., "Broadcast protocols in packet switched computer networks,", Stanford Electronics Lab. Tech. Rept. 128, Stanford University, April 1977.

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

[EGM]
Erdös, P., Geréncser, L., and Maté, A., "Problems of graph theory concerning optimal design," pp. 317-325 in Combinatorial Theory and its Applications, Vol. 1,, ed. P. Erdös, V.T. Sós, North-Holland Publishing Company, Amsterdam (1970).

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

[GG] Greene, J.W. and Gamal, A. El, "Area and delay penalties in restructurable wafer-scale arrays," pp. 165 - 184 in Proceedings 3rd Caltech Conference on VLSI, ed. R. Bryant, Springer Verlag, Berlin (1983).

[Mu] Mullender, S.J., "Distributed systems management in wide-area networks", Computer Science Dept. Tech. Rept. CS-R8419, CWI, Amsterdam, November, 1984.

[MW]
Mullender, S.J. and Wattel, E., "Locating services in store-and-forward packet switched computer networks," *In preparation*, 1985.

[NN] NN, "Problem Section," *Mathematics Magazine*, vol. 40, p.164, 1967.

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

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

[TMb]
Tanenbaum, A. S. and Mullender, S.J., "The design of a capability-based distributed operating system", Computer Science Dept. Tech. Rept. CS-R8418, CWI, Amsterdam, October, 1984.

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