Distributed Match-Making

Sape J. Mullender

Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Paul M.B. Vitányi

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts
and
Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

ABSTRACT

This is a preliminary draft version of [Algorithmica. 3 (1988), 367-391. (Special 'Distributed Computing' issue)]. It contains some material not found in the published version.

In many distributed computing environments, processes are concurrently executed by nodes in a store-and-forward communication network. Distributed control issues as diverse as name server, mutual exclusion, and replicated data management, involve making matches between such processes. We propose a formal problem called 'distributed match-making' as the generic paradigm. Algorithms for distributed match-making are developed and the complexity is investigated in terms of messages and in terms of storage needed. Lower bounds on the complexity of distributed match-making are established. Optimal algorithms, or nearly optimal algorithms, are given for particular network topologies.

1980 Mathematics Subject Classification: 68C05, 68C25.
CR Categories: C.2.1, F.2.2, G.2.2.
Keywords & Phrases: locating objects, locating services, mutual exclusion, replicated data management, distributed algorithms, computational complexity, store-and-forward computer networks, network topology.

The work of the second author was supported in part by the Office of Naval Research under Contract N00014-85-K-0168, by the Office of Army Research under Contract DAAG29-84-K-0058, by the National Science Foundation under Grant DCR-83-02391, and by the Defence Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125. Current address of both authors: CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

1. Introduction

A distributed system consists of computers (*nodes*) connected by a communication network. Each node can communicate with each other node through the network. There is no other communication between nodes. Distributed computation entails the concurrent execution of more than one process, each process being identified with the execution of a program on a computing node. Communication networks come in two types: broadcast networks and store-and-forward networks. In a broadcast network a message by the sender is broadcasted and received by all nodes, including the addressee. In such networks the communication medium is usually suited for this, like ether for radio. An example is Ethernet. Here we are interested in the latter type, store-and-forward networks, where a message is routed from node to node to its destination. Such networks occur in the form of wide area networks like Arpa net, but also as the communication network of a single multicomputer. The necessary coordination of the separate processes in various ways constitutes distributed control. We focus on a common aspect of seemingly unrelated issues in this area, such as name server, mutual exclusion and replicated data management. This aspect is formalized below as the paradigm ''Distributed Match-Making.'' Roughly speaking, the problem consists in associating with each node v in the network two sets of network nodes, $P(v)$ and $Q(v)$, such that the intersection $P(v) \cap Q(v')$ for each ordered node pair (v,v') is nonempty. We want to minimize the average of $|P(v)| + |Q(v')|$, the average taken over all pairs (v,v'). This average is related to the amount of *communication* (number of messages) involved in implementations of the distributed control issues mentioned. We also associate with each node v in the network a set $S(v) = \{v' : v \in P(v')\}$. Then $|S(v)|$ represents the amount of *storage* needed in node v. We want to minimize the average storage, or worst case storage, over all nodes in the network as well.

As the most important contribution of this paper we regard the insight that there is a common core in many hitherto unconnected distributed control issues, and the subsequent isolation and formalization of this core in the form of the Distributed Match-Making paradigm. Previously, for instance in name servers in distributed operating systems, only ad hoc solutions were proposed. Lack of any theoretical foundation necessarily entailed that comparisons of the relative merit of different solutions could only be conducted on a haphazard basis. The second contribution we make is to analyse the formal problem, develop appropriate cost measures, and establish optimal lower bounds and trade-offs on the communication and storage complexity. The analysis leads to a natural quantification of the distributedness of a match-making algorithm. Our complexity results hold for the full range from centralized via hierarchical to totally distributed algorithms for match-making. For instance, if all nodes play a symmetric role in a match-making strategy, then, for any n-node network, the two nodes making a match need to use at least $2\sqrt{n}$ messages. The third contribution entails optimal, or nearly optimal, match-making algorithms for

many different network topologies, including Manhattan networks, binary n-cubes, hierarchical networks and the estimated logical topology of Usenet. For instance, we exhibit $2\sqrt{n}$ message algorithms for node-symmetric match-making in Manhattan networks, binary n-cubes and other networks. The fourth contribution consists in detailed suggestions for further research in this area, and how to relax the conditions to obtain better scalability of our algorithms. (For a million node network, 2000 messages to make a match between a pair of nodes is too much.) The paper is organized as follows. In Section 2 we give the formal statement of the problem and analyse its complexity. Section 3 contains short outlines of the practical notions of name server, mutual exclusion and replicated data management, and the relation with distributed match-making. It also contains references to related research. The reader who wants to know what match-making is good for, can proceed there first. This section also serves to further enhance Section 4 which gives simple algorithms for distributed match-making in networks with various topologies. Finally, in Section 5, we give some open-ended suggestions for methods which are unavoidably more messy than the deterministic ones analyzed, but which may better serve practical needs. These methods involve hashing and randomness. In this initial approach to the problem we assume that the systems are failure free. While this makes the problem more pure and easier to access, obviously extensions involving failures will enhance applicability.

2. The Problem

If U is a set, then $|U|$ denotes the number of elements, and 2^U denotes the set of all subsets of U. Given a set of elements $U = \{1, 2, \ldots, n\}$ and total functions

$$P, Q: U \to 2^U \quad ,$$

such that $|P(i) \cap Q(j)| = 1$ for all i, j, $1 \le i, j \le n$.

Question 1. Find a lower bound on the average of $|P(i)| + |Q(j)|$, the average taken over all ordered pairs $(i, j) \in U^2$. Investigate this lower bound when $|P(i)|$ and $|Q(j)|$ can be chosen freely, and when either one of $|P(i)|$ or $|Q(j)|$ has a prescribed value.

Question 2. If $S(i) = \{j : i \in P(j)\}$, then find trade-offs between the lower bound of Question 1, and the average number of elements (or worst case number of elements) in $S(i)$, the average taken over all i in U.

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 $P(i)$ (or $Q(j)$) is $|P(i)|/n$ (or $|Q(j)|/n$). 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 $|P(i)|\,|Q(j)|/n^2$. Since there are n elements in U, the expected size of $P(i) \cap Q(j)$ is given by

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

Therefore, to expect precisely one element in $P(i)\cap Q(j)$, we must have $|P(i)|+|Q(j)| \geq 2\sqrt{n}$. The above analysis holds for each ordered pair (i,j) of elements of U, since all nodes are interchangeable. Consequently, the minimal average value of $|P(i)|+|Q(j)|$ over all ordered pairs (i,j) in U^2 is $2\sqrt{n}$.

By deliberate choice of the sets $P(i)$ and $Q(j)$, as opposed to random choice, the result may improve in two ways.

(1) The intersection of $P(i)$ and $Q(j)$ with certainty contains one element, as opposed to one element expected, and

(2) $|P(i)|+|Q(j)| < 2\sqrt{n}$ suffices, for selected pairs, or even on the average.

Option (2) is suggested by the fact that the elements of U need not be treated as symmetric. For instance, with one distinguished element in U we can get by with $|P(i)|+|Q(j)| = 2$ on the average (see below and Example 3 in the Appendix).

2.1. Complexity

Denote the singleton set $P(i)\cap Q(j)$ by $r_{i,j}$, and call $r_{i,j}$ the *rendez-vous* element. (For convenience, identify a singleton set with the element it contains.)

Definition. The $n\times n$ matrix, R, with entries $r_{i,j}$ $(1\leq i,j\leq n)$ is the *rendez-vous* matrix (Figure 1). Note that:

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

By choice of $P(i)$'s and $Q(j)$'s we can always replace the inclusions in (M1) by equalities. We also say that R represents a *match-making strategy* between each ordered pair (i,j) of nodes in U. The interpretation is that i sends messages to all elements in $P(i)$, and j sends messages to all elements in $Q(j)$, to effect a match of the ordered pair (i,j) at $r_{i,j}$. In many applications we can assume that a node needs to send no messages to itself, which corresponds to empty elements $r_{i,i}$ on the main diagonal. This gives minor changes in the results below. For simplicity we do not make this assumption. Examples of *rendez-vous* matrices for different strategies, ranging from centralized via hierarchical to distributed, are given in the Appendix. The reader may find it useful to look at the examples before continuing.

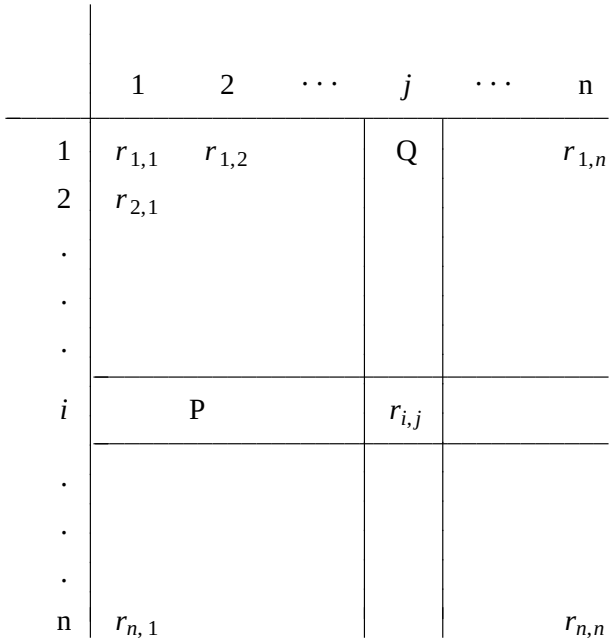

Figure 1. Rendez-vous matrix

2.1.1. Lower Bound

The *number of messages* $m(i,j)$ involved in the *match-making instance* associated with (i,j) is:

$$m(i,j) = |P(i)| + |Q(j)| \quad . \tag{M2}$$

In Example 7 in the Appendix we see that, for different pairs (i,j), the number of messages $m(i,j)$ for a match-making instance can, in a single *rendez-vous* matrix, range all the way from a minimum of 2 to a maximum of $2n$. We can determine the quality and complexity of a match-making strategy by the *minimum* of $m(i,j)$, or the *maximum* of $m(i,j)$. But the most significant measure appears to be the *average* of $m(i,j)$, for i,j ranging from 1 to n.

Definition. The *average* number of messages m of a match-making strategy (as determined by the *rendez-vous* matrix R) is:

$$m = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} m(i,j) \quad . \tag{M3}$$

We call m the *communication* complexity of R. We denote by $m(n)$ the *optimal* communication complexity, i.e., $m(n)$ equals the minimum value of m associated with R, where R ranges over all $n \times n$ *rendez-vous* matrices.

Generalizing the examples in the Appendix, we see that Example 1 and 2 have $m = n+1$, and

Example 3 has $m=2$. Thus, the method in Example 3 is more efficient. However, it is centralized. The method of Examples 1 and 2 is less efficient but is distributed. The symmetric method of Example 4 ($m=\sqrt{n}$) is more efficient than that of Example 1,2 and in some sense as distributed. Distributed methods are preferable since they can tolerate failures and distribute the message load better than centralized ones. A question is how to express the trade-off between communication efficiency and distributedness of these algorithms? It appears that communication efficiency is intimately tied up with the frequencies with which the respective nodes occur in the *rendez-vous* matrix.

Define the *frequency* k_i of i in R as the number of times element i occurs as an entry in R, i.e., how often i is used as *rendez-vous* for an ordered pair (j,k) of elements $(1\leq i,j,k\leq n)$. Clearly,

$$\sum_{i=1}^{n} k_i = n^2 \quad . \tag{M4}$$

We call the n-tuple $(k_1, \ldots, k_n)$ the *distribution vector* of R, and we consider it as a measure for the *distributedness* of strategy R. Looking at two extremes we see that this makes sense. If there is an i such that $k_i=n^2$ and $k_j=0$ for $j\neq i$, $1\leq i,j\leq n$, then the strategy is *centralized* (Example 3 with $m=2$). If $k_i=n$ for all i, $1\leq i\leq n$, then we call the strategy *distributed* (Examples 1 and 2 with $m=n+1$, and Examples 4 and 8 with $m=\sqrt{n}$). Intuitively, the statistical variation of the k_i's measures the distributedness of a strategy in a single figure. We derive a lower bound (Proposition 2) on $m(n)$ expressed in terms of the k_i's. We show that this lower bound optimal for distribution vectors $(n, \ldots, n)$ and $(0,...,0,n^2,0,...,0)$, by exhibiting strategies R which achieve it. We conjecture that the lower bound is optimal for all distribution vectors. To prove Proposition 2, it is useful to proceed by way of Proposition 1. Not only is Proposition 1 combinatorially more interesting than Proposition 2, which is an easy corollary, but it also quantifies the optimal trade-off between the sizes of the P-sets and the Q-sets. It has already been useful elsewhere in analysing many-dimensional and weighted versions of distributed match-making in [Kranakis colored

Proposition 1. *Consider the rendez-vous matrix R as defined above. Then,*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} |P(i)|\,|Q(j)| \geq \left(\sum_{i=1}^{n}\sqrt{k_i}\right)^2 \tag{M5}$$

Proof. The sum above is minimized if, for each element i of U, all entries of i in R occur in a $\sqrt{k_i}\times\sqrt{k_i}$ rectangle. We make this intuition precise as follows. First observe that while P and Q determine the number of distinct entries of $i\in U$ in each row and column, it is more convenient to consider the number of different rows and columns in which each element i of U occurs. Let r_i [c_i] be the number of different nodes in row i [column i] $(1\leq i\leq n)$. 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 ($1 \le i \le n$). 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 $\gamma_{i,j}=0$, ($1 \le i,j \le n$). Then,

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

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

The closest the occurrences of a particular element of U can be packed in the matrix is as a rectangle. This gives rise to the inequality:

$$R_i C_i \ge k_i \quad , \tag{3}$$

for all i ($1 \le i \le n$). We only need one other inequality. Using the fact that the square of a difference must be nonnegative:

$$\begin{aligned} k_j R_i^2 - 2\sqrt{k_i k_j}\, R_i R_j + k_i R_j^2 &= (\sqrt{k_j}\, R_i - \sqrt{k_i}\, R_j)^2 \\ &\ge 0 \quad , \end{aligned}$$

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

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

from which it follows that:

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

This gives us the required result, since

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{n} |P(i)|\,|Q(j)| &\ge \sum_{i=1}^{n}\sum_{j=1}^{n} r_i c_j \quad \text{(by (M1) \& (1))} \\ &= \sum_{i=1}^{n} r_i \times \sum_{j=1}^{n} c_j \\ &= \sum_{i=1}^{n} R_i \times \sum_{j=1}^{n} C_j \quad \text{(by (2))} \\ &\ge \sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_j R_j^{-1} \quad \text{(by (3))} \end{aligned}$$

$$\geq \left(\sum_{i=1}^{n} \sqrt{k_i} \right)^2 \quad \text{(by (4))},$$

which yields the Proposition. •

Proposition 2.

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

Proof. Define everything as in the proof of Proposition 1. Use the fact that the square of a difference is nonnegative to derive:

$$\sum_{i=1}^{n} r_i^2 + \sum_{i=1}^{n} c_i^2 \geq 2 \sum_{i,j=1}^{n} r_i c_j \tag{5}$$

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

$$n^2 m(n) \leq \sum_{i=1}^{n} \sum_{j=1}^{n} (r_i + c_j) = n \sum_{i=1}^{n} (r_i + c_i)$$

$$= n \left(\sum_{i=1}^{n} r_i + \sum_{i=1}^{n} c_i \right) < 2n \sum_{i=1}^{n} \sqrt{k_i} \ . \tag{6}$$

Divide both sides of inequality (6) by *n*, and square the results. Substitute for the sum of squares in the resulting lefthand side, using (5), and divide by 4. Then we are left with

$$\sum_{i=1}^{n} r_i \sum_{i=1}^{n} c_i < \left(\sum_{i=1}^{n} \sqrt{k_i} \right)^2 \ ,$$

which contradicts Proposition 1. •

It is not difficult to see that Propositions 1 and 2 hold *mutatis mutandis* for nonsquare matrices *R*. For totally *distributed* strategies they specialize to:

Corollary. *Let R be a rendez-vous matrix such that* $k_1 = k_2 = \cdots = k_n = n$. *Then:*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} |P(i)| \, |Q(j)| \geq n^3 \ , \text{ and } \ m \geq 2\sqrt{n} \ .$$

The second inequality is the same lower bound we saw in the probabilistic analysis. Note that in the latter case the elements were also symmetric in the sense of interchangeable. Singling out one element gives *centralized* match-making as follows:

Corollary. *Let R be a rendez-vous matrix such that* $k_2 = k_3 = \cdots = k_n = 0$ *and* $k_1 = n^2$, *that is, 1 is the central element. Then:*

$$\sum_{i=1}^{n}\sum_{j=1}^{n} |P(i)|\,|Q(j)| \geq n^2 \quad \text{, and } \; m \geq 2 \; .$$

Remark. The constraints (M1)-(M5) and Proposition 1 give a trade-off between the $P(i)$'s and $Q(j)$'s, which is much stronger than the one implied by Proposition 2. We can illustrate this by a simple example. If $P(i)=p$ and $Q(i)=q$ for $1\leq i\leq n$, then by Proposition 1 we have $pq\geq n$. If we set $p=n^{1/4}$, then it follows that $q\geq n^{3/4}$, which gives $p+q\geq n^{3/4}+n^{1/4}$. Proposition 2 gives, for $p=n^{1/4}$ only $q\geq 2n^{1/2}-n^{1/4}$, while $p+q>2n^{1/2}$ does not change. As suggested by this example, we can use the trade-off in Proposition 1 to adjust distributed match-making strategies so as to minimize the *weighted* overall number of messages. For instance, in many applications as in Section 3, we are actually interested in minimizing m with (M2) replaced by (M2′):

$$m(i,j) = |P(i)| + \alpha_{i,j}\,|Q(j)| \quad . \qquad \text{(M2′)}$$

This question is treated in [Kranakis colored

2.1.2. Upper Bound

The lower bounds can be matched by upper bounds, modulo integer round-off. E.g.,

Proposition 3. *Let $U=\{1,\ldots,n\}$ be as above. (i) If n is an square integer, then there exist functions P, Q as required, with distribution vector $(n,\ldots,n)$, such that, for all $1\leq i,j\leq n$, $|P(i)|\,|Q(j)|=n$, and $|P(i)|+|Q(j)|=2\sqrt{n}$. (ii) There exist functions P, Q as required, with distribution vector $(0,...,0,n^2,0,...,0)$, such that, for all $1\leq i,j\leq n$, $|P(i)|\,|Q(j)|=1$, and $|P(i)|+|Q(j)|=2$.*

Proof. (i) Arrange the *rendez-vous* matrix R as a checker board consisting of $\sqrt{n}\times\sqrt{n}$ squares, of n entries each. Each square contains n copies of a single element of U, a different one for each square; *cf.* Example 4 in the Appendix.

(ii) By Example 3 in the Appendix. •

There is a way to scale up any solution so that it becomes asymptotically distributed and optimal.

Proposition 4. *Let R be the rendez-vous matrix for an n element set U. Let k_i $(1\leq i\leq n)$ be the multiplicity of element i in R, and let m be associated with R. We can lift this match-making strategy to a 4n-element set by constructing a $4n\times 4n$ rendez-vous matrix R′ with $k_j'=4k_{j \bmod n}$ is the multiplicity of element j in R′ $(1\leq j\leq 4n)$ and the associated communication complexity $m'=2m$. (I.e., iterating this process, the associated communication complexity as a function of the number n of nodes asymptotically approaches $2\sqrt{n}$, while the distribution vector approaches $(n,\ldots,n)$.)*

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':

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

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

2.2. Storage-Communication Trade-off

The Examples suggest a trade-off between storage and average number of messages. Let R be a *rendez-vous* matrix over a set $U = \{1, \ldots, n\}$. Define the *storage set* associated with each $i \in U$ as $S(i) = \{j : i \in P(j), 1 \le i, j \le n\}$, and $|S(i)|$ is the *storage* complexity of i. Let $s = \max\{|S(i)| : i \in U\}$ denote the *worst-case storage* needed in strategy R. Let m be the communication complexity of R.

Proposition 5. *For a rendez-vous matrix R over an n-element set we have* $s(m-1) \ge n$.

Proof. Let the number v of *rendez-vous* elements in R be $v = |\{k : k \in P(i) \cap Q(j), 1 \le i, j, k \le n\}|$. Form a graph, with $2n+v$ nodes called $a(1), \ldots, a(n)$, $b(1), \ldots, b(n)$, $r(1), \ldots, r(v)$. Think of the $a(i)$ as the arguments of P, the $b(i)$ as the arguments of Q, and the $r(i)$ as the rendez-vous elements. Whenever j is the rendez-vous element $P(i) \cap Q(k)$, put an edge from $a(i)$ to $r(j)$ and an edge from $r(j)$ to $b(k)$. Let $d(j)$ be the number of edges from a nodes to $r(j)$, and let $e(j)$ be the number of edges from b nodes to $r(j)$. Clearly m equals $(d(1)+\cdots+d(v)+e(1)+\cdots+e(v))/n$, since $d(1)+\cdots+d(v)+e(1)+\cdots+e(v)$ is the sum of the $|P(i)|$'s and the $|Q(i)|$'s. By definition $s = \max\{d(j) : 1 \le j \le v\}$. Thus each $r(j)$ has at most s edges to a nodes. We know that for any i and k there is some j such that $r(j)$ is adjacent to both $a(i)$ and $b(k)$. Call such a triple (i,k,j) a good triple. Now there are at least n^2 good triples. On the other hand it is obvious that for any j there are at most $d(j)e(j)$ good triples with j as the last entry. Thus,

$$m = \frac{d(1)+\cdots+d(v)+e(1)+\cdots+e(v)}{n} ,$$

$$s(e(1)+\ldots+e(v)) \ge e(1)d(1)+\cdots+e(v)d(v) \ge n^2 \text{ , and}$$

$$d(1)+\cdots+d(v)\geq n$$

yield

$$sm \geq n+\frac{s(d(1)+\cdots+d(v))}{n}$$

$$\geq n+s\ .$$ •

Obviously, the graph we constructed is not necessary for the proof but it helps in visualizing what is going on. In centralized match-making as in Example 1 we have $m=2$ and therefore $s\geq n$. For broadcasting in Examples 1,2 we have $m=n+1$, and therefore $s\geq 1$. For a distributed method like Example 4, we have $m=2\sqrt{n}$, and therefore $s>\sqrt{n}/2$.

Another indication of storage requirements is the average storage. The *average storage* s_{ave} for a particular strategy is

$$s_{ave}=\frac{1}{n}\sum_{i=1}^{n}|S(i)|=\frac{1}{n}\sum_{i=1}^{n}|P(i)|\ .$$

Therefore, it follows straightaway from Proposition 1 that

Proposition 6. *Let everything be as above. Then:*

$$s_{ave}\geq\frac{(\sum_{i=1}^{n}\sqrt{k_i})^2}{n\sum_{i=1}^{n}|Q(i)|}=\frac{(\sum_{i=1}^{n}\sqrt{k_i})^2}{n\sum_{i=1}^{n}(m-|P(i)|)}\ .$$

3. Three Distributed Control Issues

Below we give three distributed control issues exhibiting match-making features. These are name server, mutual exclusion and replicated data management. Since some form of distributed match-making is required in all of them, algorithms for these problems are subject to the limitations analysed in the previous section. We assume throughout that we are dealing with a set of computers connected by a network. The network is a communication graph $G=(U,E)$, where U is the set of nodes (computers), and E is the set of edges. Each edge represents two-way noninterfering communication channels between the nodes it connects. Nodes communicate only by messages and do not share memory. The underlying communications network G is error-free, and supports the message transfers in which the delivery time may vary but messages between two nodes are delivered in the order sent. We will identify the idealized distributed match-making subproblems below, by exhibiting P and Q functions. In each case it will turn out that there is a

requirement $P(i) \cap Q(j) \neq \emptyset$, for each pair (i,j) of nodes in U^2. To obtain the lower bounds in in the previous section, w.l.o.g. we used a minimal requirement $|P(i) \cap Q(j)| = 1$.

3.1. Name Server

In the object model of software design, 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 [abstract data types For example, a typical object is the file, with operations to read and write portions of it. A major advantage of the object or abstract data type model is that the semantics are inherently location independent, and therefore convenient for multicomputer systems. 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. It is convenient to *implement* the object model in terms of *clients* (users) who send messages to *services* [Tanenbaum Mullender overview amoeba Each service is handled by one or more *server* processes that accept messages from clients, carry out the required work, and send back replies. A process can be a client, a server, or both. 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. Before a client can send a request to a server which provides the desired service, the client has to locate that server. Each service is identified by a *name*. A client asks the system for a particular service by its name. The mechanism that translates the *name* of a service into a location or *address* in the network is called a *name server*. Thus, the name server offers yet another service in the system, be it the *primus inter pares*. A *centralized* name server must reside at a well-known address which does not change and is known to all processes. (Clearly, the name server cannot be used to locate itself. You can't call the telephone directory assistance server to obtain the number of telephone directory assistance, if you don't know it.) When the host of a centralized name server crashes, the entire network crashes. A centralized name server also requires excessive amount of storage at the host site, and causes message overload (hot spot) in the host's neighborhood. These disadvantages can be overcome by distributing the name server. One way to distribute the name server, is to have clients broadcast for services with ''where are you'' messages. 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 costs at least as many message

as there are nodes in the network.

If processes never move, than the name for a service can encode the address where an appropriate server resides. We assume that processes are *mobile*, but we make the simplifying assumption that during a locate of a server by a client, the process/processor allocation does not change. Let $h(p)$ be the current *address* of process p's host. Since processes may migrate, die or be created, $h(p)$ can change, become empty or nonempty. Locate of services by the processes is achieved by the following procedure. Each server s selects a set $P(s)$ of nodes and posts at these nodes the availability of the service it offers and the address $h(s)$ where it resides. (Each node in $P(s)$ stores this information in its individual *cache*.) When a client c wants to request a service it selects a set $Q(c)$ of nodes and queries each node in $Q(c)$ for the required service. When $P(s) \cap Q(c)$ is not empty the node (or any node) in $P(s) \cap Q(c)$ will be able to return a message to c stating the address $h(s)$ at which the service is available (recall that this information is already stored in the caches of all the nodes in $P(s)$). For example, a *centralized* name-server corresponds to

$$P(s) = \{x\},\ Q(c) = \{x\},$$

for all servers s and clients c with $h(s), h(c) \in U$, and a fixed $x \in U$ (Example 3 in the Appendix). As another example, *broadcasting* corresponds to

$$P(s) = \{h(s)\},\ Q(c) = U,$$

for all servers s and clients c with $h(s), h(c) \in U$ (Example 1 in the Appendix). In the formal set-up, we restricted ourselves to methods where the sets $P(s)$ and $Q(c)$ depend on the respective hosts $h(s)$ and $h(c)$ only. It therefore makes more sense to talk about $P(h(s))$ and $Q(h(c))$ instead of $P(s)$ and $Q(c)$. The relation with match-making is now established.

The research reported in this paper started from design considerations of the name server in the Amoeba operating system [Tanenbaum Mullender design capability-based distributed operating system In an early version of this paper [Mullender Match-Making the focus was only on algorithms for a distributed name server in computer systems with mobile processes. Essentially the Manhattan topology method (cf. later) has been used before in the torus-shaped Stony Brook Microcomputer Network [Gelernter In [Farber The Distributed System Communication the name server is implemented by broadcasting. In the Cosmic Cube processes run on fixed processors [cosmic cube Other system designers have chosen for mobile processes, but use the crash-vulnerable solution of a centralized name server [Needham Cambridge A detailed proposal for a hierarchical name server is contained in [Lampson global Methods which maintain a tree of forwarding addresses in the network, for each mobile process, have been used in [Powell Miller and analysed [Fowler

3.2. Mutual Exclusion

Another application of the match-making paradigm is distributed *mutual exclusion*. Suppose processes can compete for a single resource in the system, while this resource can be granted to only one of them at a time. An example is a printer which can be accessed by several processes from different hosts. The problem consists in designing a protocol which ensures that only one process is granted access to the resource at a time, while satisfying certain ‘‘niceness’’ conditions such as absence of deadlock. This problem was originally formulated by Dijkstra [Dijkstra Solution Problem Concurrent Programming Control The assumption of the availability of mutual exclusion underlies much of the work in concurrency. For a thorough treatment see [Lamport Mutual Exclusion Problem Assume that each network node can issue a mutual exclusion request at an arbitrary time. In order to arbitrate the requests, any pair of two requests must be known to one of the arbitrators. Since these arbitrators must reside in network nodes, any pair of two requests originating from different nodes must reach a common node. Assume that each node i must obtain a permission from each member of a subset $S(i)$ of U before it can proceed to enter its critical section. Then for each pair $(i,j)\in U^2$ we must have $S(i)\cap S(j)\neq\emptyset$ so that the node in the intersection can serve as arbitrator. The complexity of a distributed mutual exclusion strategy is the average number of messages involved in a mutual exclusion request from a node i, with the average taken over all nodes. In [Maekawa the situation is analysed where each node in the network serves as arbitrator equally often, that is, $|U|$ times. The actual algorithm presented uses at most $5\cdot|S(i)|$ messages, where for some K, $|S(i)|=K$ for all i, $i\in U$. It is clear that at least $2K$ messages are required: K messages to query a set $S(i)$, and K answers from every member of $S(i)$ to i. The overhead of $3K$ messages arises from additional locking and unlocking protocols to guarantee mutual exclusion, absence of deadlock, and so on. Here, we may view a *strategy* for distributed mutual exclusion as a mapping

$$S: U\rightarrow 2^U$$

and view it as a restricted case of match-making for which the *symmetry condition* $P(i)=Q(i)$ $(=S(i))$ holds for all $i\in U$. One way to achieve this symmetry is to let the functions P, Q be as in the original definition, and set $S(i)=P(i)\cup Q(i)$ for all i, $i\in U$. In [Maekawa the particular match-making algorithm for the projective plane topology is investigated, cf. also Section 4.

3.3. Replicated Data Management

We describe a variant originating from [Awerbuch Hardware] [Awerbuch Kirousis This is related to replication methods as in [Gifford 1979] [Gifford 1983 Contrary to the latter references, we assume that the system is failure free. A *replicated object* is implemented by a collection of *versions* of the object. Here, let the replicated object be a *variable* which is *shared* by

several users. The operations are *read*s and *write*s. A read returns the variable's current value, and a write assigns a new value to the variable. Reads and writes by different users are allowed to occur concurrently, but we do not allow concurrent operations by different users to wait for one another. Our objective is to manage the shared variable such that it appears atomic. *Atomicity* means that each operation execution can be *thought* to happen in an indivisible instant of time, a different instant for each operation, such that the reads and writes are totally ordered. This ordering should be internal consistent in the sense that each read returns the value assigned by the last write that precedes it in the order. To have external consistency, the time instant the operation appears to take place must be within the actual duration of the operation execution. We implement an atomic shared variable by maintaining several *versions* of it. Each version can be written by each user out of an associated set of *writers*, and read by each user out of an associated set of *readers*. Call the operations on the versions *subreads* and *subwrites*. Each read or write by a user comprises many subreads and subwrites. Each version resides at a user. We assume that the subreads and subwrites to a version are executed atomically, i.e., that a version is itself an atomic shared variable. Our goal is to implement an atomic variable, which is shared by a set of users, using atomic variables which are shared by *subsets* of users only. Below we give a family of algorithms which reduces the problem to match-making.

A *quorum* $T(u)$ for an operation by user u on the shared variable, is a set of versions whose cooperation is sufficient to execute that operation. It is convenient to divide the quorum in an *initial* quorum $Q(u)$ and a *final* quorum $P(u)$. Each version has an attached *version number*, to identify the order in which the versions were created. A *version number* is a pair (t,u), where t is a nonnegative integer and u a user identifier. Let the user identifiers be 1 through n if there are n users.

To *read* the variable, a user, say v, reads the versions from its initial quorum $Q(v)$, and determines the version with the lexicographic greatest version number, say (t,u). Let this version contain value M. Then v writes value M together with version number (t,u) to the versions in its final quorum $P(v)$. (Note that u may be unequal v.) The value returned by v is M.

To *write* value N to the shared variable, a user, say u, first reads the version numbers of its initial quorum $Q(u)$. It determines the greatest first coordinate of the version numbers, say t. Then u writes value N together with new version number $(t+1,u)$, to the versions in its final quorum $P(u)$.

For the method to implement an atomic shared variable, quorums are subject to the following constraint: each final quorum must intersect each initial quorum. I.e., $P(u) \cap Q(v) \neq \emptyset$, for each ordered pair (u,v) of users in U^2. The proof of correctness of this algorithm is by no means simple, but outside the scope of this paper. (It is essentially given in [Awerbuch Hardware] [Awerbuch Proving) The important issue here is that we have established yet another case of

match-making, as follows. W.l.o.g., assume $P(i) \cap Q(j) = \{v_{i,j}\}$, for each pair (i,j) of users in U. Let the entry $r_{i,j}$ in the ith row and jth column of the *rendez-vous* matrix be the user node where version $v_{i,j}$ resides.

Example. Let there be n users which can read and write an atomic shared variable, as above. If each user occurs with the same frequency in the *rendez-vous* matrix, then the average number of messages between users to read or write the shared variable is $\geq 2\sqrt{n}$, by Proposition 2. A strategy achieving an average of $2\sqrt{n}$ messages to read or write the shared variable, is implemented by Example 4. That is, each node hosts a version which is itself an atomic shared variable, and which can be written by $\sqrt{n}$ writers and read by $\sqrt{n}$ readers.

4. Match-Making Algorithms

The topology of a network $G=(U,E)$ in Section 3 determines the overhead in message passes needed for routing a message to its destination. For complete networks, the number of message passes for match-making between node i and node j equals the number of messages $m(i,j)$. 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 equals the number of messages $m(i,j)$. Otherwise, there is an overhead of message passes for routing messages from node i to $P(i)$, and from node j to $Q(j)$. For this reason, in a *ring network, no* match-making algorithm can do significantly better than broadcasting (i.e., the average number of message passes involved in matching a pair of nodes is $\Omega(n)$).

4.1. General Networks

Let $G=(U,E)$ be an arbitrary network. 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. It is not difficult to give a construction to cover every connected graph with $O(\sqrt{n})$ *connected* subgraphs of $\leq \sqrt{n}$ nodes each. The subgraphs can not always be chosen pairwise disjoint as is shown by the counterexample of an n-node *star* graph with $n-1$ nodes of degree 1 and 1 node of degree $n-1$. If the original graph has node degree bounded by a constant then the covering subgraphs can be constructed pairwise disjoint (Proposition 7). Either way, label the nodes in each subgraph 1 through $\sqrt{n}$. If the subgraph has less than $\sqrt{n}$ nodes, then use up the excess numbers by labelling some nodes more than once. Then, the shortest path, between the each pair of nodes labelled i in two adjacent connected subgraphs, is not longer than $2\sqrt{n}$. Let there be $k \in O(\sqrt{n})$ subgraphs G_j, $1 \leq j \leq k$. Denote a node labelled i in a subgraph G_j by (i,j), $1 \leq i \leq \sqrt{n}$ and $1 \leq j \leq k$. Let $P((i,j))$ consist of all nodes labelled i in G_l, $1 \leq l \leq k$. To access all nodes in $P((i,j))$ from the original node takes $O(\sqrt{n})$ messages, but $O(n)$ message passes. Size $O(\sqrt{n})$ suffices for the cache of each node. Let $Q((i,j))$ equal the set of all nodes in G_j. To

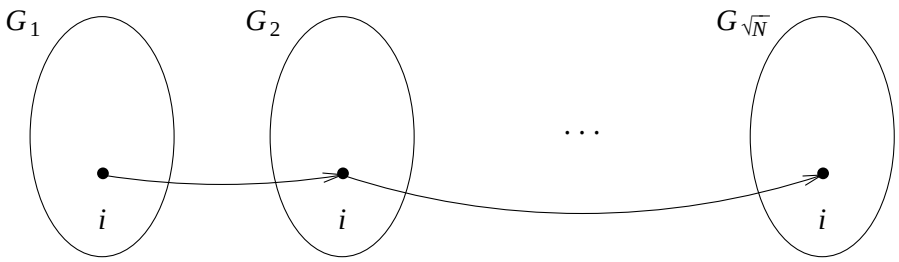

Figure 2.

access all these nodes in $Q((i,j))$ from (i,j), takes at most $\sqrt{n}$ messages (or message passes along a spanning tree of G_j).

The algorithm has communication complexity $O(\sqrt{n})$ messages, but cannot guaranty a better upper bound than $O(n)$ message passes. However, for application to the nameserver, we can make the assumption that clients need to locate services usually far more frequently than servers need to post their whereabouts. Then this scheme is fairly optimal in message passes too.

If for some reason we want the connected subgraphs to be pairwise disjoint, then this is not always possible, as shown above. However,

Proposition 7. *Let G be a connected graph with a spanning tree of bounded node degree. Then G can be divided in* $O(\sqrt{n})$ *disjoint connected subgraphs, containing* $O(\sqrt{n})$ *nodes each.*

Proof. Consider a rooted spanning tree T (of G) with node degree $\leq c$, c a constant. Let v be a node farthest from the root with $\geq d$ descendants. By the bounded degree of T it follows that v has $\leq cd \in O(d)$ descendants. Take v and its subtree from the tree as the first connected subgraph of $\Theta(d)$ nodes. Repeat the procedure on the remainder of T. As long as $\geq d$ nodes remain we can separate off another cluster of $\Theta(d)$ nodes. The final remainder can be attached to the preceding cluster. Therefore, we obtain a division of G in $O(n/d)$ disjoint connected subgraphs of $\Theta(d)$ nodes each. Setting $d=\sqrt{n}$ yields the proposition. •

If we settle for the subgraphs having *diameter* $O(\sqrt{n})$, as opposed to *number of nodes* $O(\sqrt{n})$, then we can use a result due to [Problems graph theory optimal design

Proposition 8. *Each connected graph of n nodes can be divided into* $O(\sqrt{n})$ *connected subgraphs of diameter* $O(\sqrt{n})$ *each.*

Proof. Consider a rooted spanning tree T of the original graph G. Choose d and divide T in layers of d levels each. Take the subtrees rooted at level id ($i \geq 0$) which reach to level $(i+1)d-1$. If a subtree does not reach to that level (has depth $<d$) then attach it to the subtree just above it. (Thus, resulting subtrees may have up to $2d-1$ levels.) The set of such subtrees induces a covering of the original graph by pairwise disjoint connected subgraphs of diameter $\leq 4d$ and each $\geq d$

nodes (separate argument for top part). This yield O(n/d) subgraphs of diameter O(d) and yields the mentioned result for $d=\sqrt{n}$. ●

4.2. Manhattan Networks

The network $G=(U,E)$ is laid out as a $p\times q$ rectangular grid of nodes, $U=\{(i,j): 1\le i\le p,\ 1\le j\le q\}$ and there is an edge in E between (i,j) and (i',j') if either $|i-i'|=1$ or $|j-j'|=1$, but not both. $P((i,j))=\{(i,k): 1\le k\le q\}$ is the set of nodes in row i, and $Q((i,j))=\{(k,j): 1\le k\le p\}$ is the set of nodes in column j. Caches are of size O(q) and number of messages (=message passes) for each match-making instance is O($p+q$). For $p=q$ we have $m=2\sqrt{n}$ and caches of size $\sqrt{n}$. For the 3×3 network below,

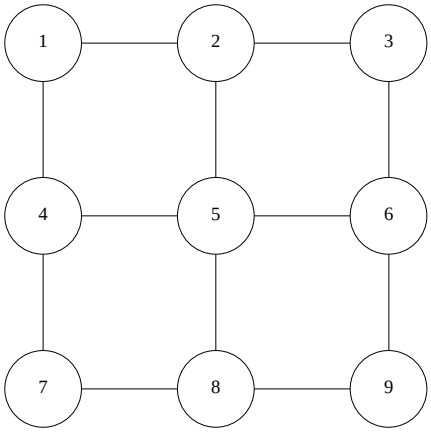

Figure 3.

the corresponding 9×9 *rendez-vous* matrix is given as Example 8 in the Appendix. 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 [gelernter In the obvious generalization to d-dimensional meshes the method takes $2n^{(d-1)/d}$ message passes. However, at the cost of shifting the load of being a *rendez-vous* node from the interior of the mesh to the surface we can improve matters. Take as example a name server in a 3-dimensional mesh. A server at (x_s,y_s,z_s) sends its advertisement by a shortest path parallel to the x axis to the surface and next circumnavigates the surface of the mesh in the plane $z=z_s$. The client at (x_c,y_c,z_c) sends its query along a shortest path parallel with the y axis to the surface and next circumnavigates the surface in the plane $x=x_c$. The *rendez-vous* nodes are the two nodes on the mesh surface which are incident on the intersection line of the planes $z=z_s$, $x=x_c$. If the mesh has n nodes, $n^{1/3}$ to a side, then this method takes $9n^{1/3}$ message passes in the worst case to make a

match.

4.3. Multidimensional Cubes

A binary d-cube is a network $G=(U,E)$, such that the nodes have addresses of d bits and edges connect nodes of which the addresses differ in a single bit. $n=|U|=2^d$ and $|E|=d\,2^{d-1}$. Assume that d is even. (An obvious modification works for d is odd.) Let node s have address $s_1 s_2 \cdots s_d$.

$$P(s) = \{a_1 a_2 ... a_{\frac{d}{2}} s_{\frac{d}{2}+1} ... s_d \mid a_1, ..., a_{\frac{d}{2}} \in \{0,1\}\} \ .$$

$$Q(s) = \{s_1 s_2 ... s_{\frac{d}{2}} a_{\frac{d}{2}+1} ... a_d \mid a_{\frac{d}{2}+1}, ..., a_d \in \{0,1\}\} \ .$$

For each pair $(s,c) \in \{1, \ldots, n\}^2$, the *rendez-vous* node is given by

$$P(s) \bigcap Q(c) = \{c_1 c_2 ... c_{\frac{d}{2}} s_{\frac{d}{2}+1} ... s_d\} \ .$$

The number $m(s,c)$ of messages is the same for each pair (s,c) of nodes, and therefore $m = m(s,c) = |P(s)| + |Q(c)| = 2\sqrt{n}$.

$m(s,c)$ equals the number of message passes along spanning trees in the two binary $d/2$-cubes induced by $P(s)$ and $Q(c)$, respectively. 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 $(1-\varepsilon)d$ bits. *Cf.* Example 6. For instance, in the case of the name-server we may want 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 [valiant

4.4. Fast Permutation Networks

For various reasons [historical development of circuit switching 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 $m \in O(\sqrt{n \log n})$ and caches of size $\sqrt{n/\log n}$.

4.5. 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. For each node s, $P(s)$ and $Q(s)$ comprise all nodes on an arbitrary line incident on its host node. The common node of two lines is the *rendez-vous* node. Since the nodes are symmetric, it is easy to see that

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

This combination of topology and algorithm is resistant to *failures* of lines, provided no point has all lines passing through it removed. The average necessary cache size is $\sqrt{n}$; but the price we have to pay for the fault tolerance of this method is expressed in the worst-case cache size N.

4.6. Hierarchical Networks

Hierarchy of networks. Local-area networks are often connected, by *gateway* nodes, to wide-area networks, which, in turn, may also be interconnected.* Consider a tree T of k levels, with the root at level k. A node of T at level i consists of a level i network. A level i network consists of n_i nodes, called *gateways*, connecting n_i level $i-1$ networks, for each $1 \le i \le k$. A level 0 network is a node. The obvious strategy for match-making puts the *rendez-vous* node for a pair of nodes in the least common ancestor network of the two. Suppose, we using a $2\sqrt{n_i}$ strategy, as described in the previous sections, in each level i network, $1 \le i \le k$. Let s be a node in a level j network G_j, $1 \le j \le k$. Let $G_{j+1}, \ldots, G_k$ be the chain of networks between G_j and the root network G_k. We define $P(s)$ and $Q(s)$ inductively. *Base*: if $G_j=(U_j,E_j)$, $s \in U_j$, then $P(s) \cap U_j$ and $Q(s) \cap U_j$ are chosen such that it takes an average of $2\sqrt{n_j}$ messages to make a match between a pair of nodes in G_j. *Induction*: if $s' \in U_{j+1}$ is the gateway node through which G_j is connected in its ancestor network $G_{j+1}=(U_{j+1},E_{j+1})$, then

$$P(s) \cap U_{j+1} = P(s') \cap U_{j+1} \ ,$$

$$Q(s) \cap U_{j+1} = Q(s') \cap U_{j+1} \ .$$

This gives $m \in O(\sum_{i=1}^{k} \sqrt{n_i})$ for a hierarchical network with a total of $n = \prod_{i=1}^{k} n_i$ nodes. Assume for simplicity that $n_i=\alpha$, $1 \le i \le k$. Then the total number of nodes in the network is $n = \alpha^k$, and $m \in O(k\sqrt{\alpha})$. Therefore,

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

---

* 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 node symmetric $\sqrt{n}$ 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 (e.g., memory management); of the remaining inter-host communication, most will be confined to a local-area network (e.g., temporary file store/swap service), and so on, up the network hierarchy (e.g., mail). 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.

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

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

is reached for $k = \frac{1}{2}\log n$. This message 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 entries as there are nodes in the subtree it dominates.†

Hierarchy of nodes. Many wide-area computer networks are not completely designed at the outset but grow and change dynamically. Some networks resemble an undirected tree with a core in which we can imagine the root, and with some additional edges thrown in. The extra edges would typically occur between geographically near nodes. Nodes nearer to the core of the tree tend to be of higher degree than others. In Usenet, the degree of super-backbone sites like *ihnp4* is in the order of 1000, of backbone sites like *mcvax* in the order of 100, and of feeder sites in the 10s.

Suppose we have a balanced tree with n nodes and l levels, 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:

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

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

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

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

Setting $d(l) = c\,2^{\varepsilon l}$, for constants $c, \varepsilon > 0$ yields:

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

Therefore,

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

---

† In a network hierarchy, as we have sketched, services are often exclusively accessed by local clients. It is natural for the system to provide a way to restrict the availability of a particular service to some subhierarchy of processes. Then, 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.

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

The obvious strategy in such trees is: the *rendez-vous* node of a pair of nodes is the least common ancestor. Thus, if $P(i)=Q(i)$ is the set of nodes on the path from node i to the root, then $m \in O(l)$. 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 (Section 5).

5. Conclusion and Further Research

This paper reports on initial investigations in a new theoretical problem area. We have isolated and formalized the problem of 'distributed match-making' as a new paradigm in distributed control. The complexity analysis gives theoretical limitations to the efficiency of solutions for many practical problems in the area. Even the appropriate formulation of such limitations was not previously understood. The exhibited algorithms, which are optimal or nearly optimal, may serve as guidelines for feasibility and design of applied algorithms. Below we indicate a few avenues for further research in match-making, like probabilistic algorithms, hashing and fault-tolerance. 'Lighthouse Locate' actually preceded everything else in this paper.

5.1. Lighthouse Locate

We give a probabilistic method for the name server. We imagine the processors as discrete coordinate points in the 2-dimensional Euclidean plane grid spanned by $(\varepsilon,0)$ and $(0,\varepsilon)$. 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 1 every unit of time. Each trail left by such a beam disappears instantly at the end of d time units. That is, a node discards an address posted by a server after d time units. Assume that the time for a message to run through a path of length 1 is so small in relation to d that the trail appears and disappears instantaneously.

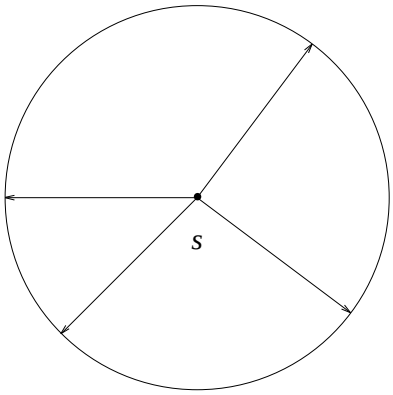

Figure 4.

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 $l(1)=1$ and the intervals are $\delta(1)$. After each unsuccessful trial, the client increases its effort by doubling the length of the inquiry beam and the intervals between them.* Thus, the ith trial has $\delta(i)=l(i)=2^{i-1}$, $i \geq 1$.

With the clients strategy governed by the integer sequence $012\cdots$ of exponents, we spend exponentially longer lasting trials with exponentially longer beams, while missing a server which was nearby all the time, or has migrated to a nearby location. A better strategy is to prevent this by an evenhanded treatment of locations far and near, independent of which interval of consecutive trials we consider. To obtain this, we govern the length of the client's locate beam (and its duration) by integer sequence 51 in [Sloane Sequence

$$0102010301020104010201030102010501 02\cdots$$

This sequence has an interesting property: for *each* $i \geq 0$, *each* uninterrupted subsequence of length 2^{i+1} contains *precisely one* integer i. Let $d(i)$ be the digit in position i, and let in the ith trial $\delta(i)=l(i)=2^{d(i)}$. Then, in each subsequence of 2^{d+1} trials there are 2^{d-j} trials with beams of

* Before Lighthouse Locate 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 [reverse path forwarding dalal 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.

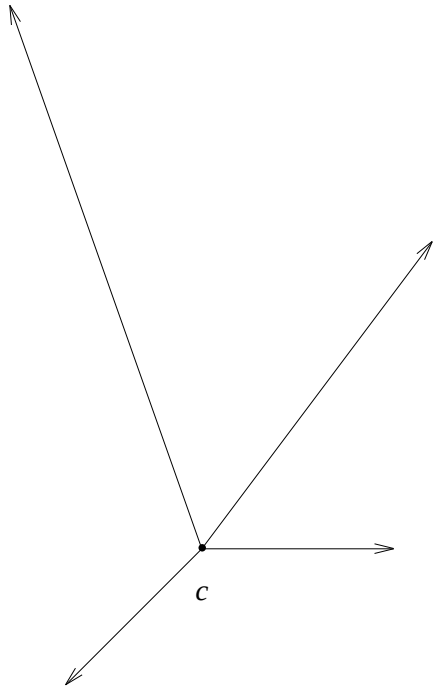

Figure 5.

length 2^j ($0 \leq j \leq d$). This 'binary carry schedule' can conveniently be maintained by a binary counter. A binary counter is initialized with 0. In each step, it is incremented by 1. Digit $d(i)$ equals the position of the most significant bit which needs to be changed in the ith step.

5.2. Hash Locate and Beyond

Let us consider the name server again. Let in a given network $G=(U,E)$ the set of ports (i.e., types of services available) be Π. In *Hash Locate* we construct hash functions that map service names onto network addresses. That is,

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

This technique is very efficient. Each server s posts its (service, address) at the node(s) $P(\pi)$, if π is the service offered by s, and a client in need for a service π queries the node(s) in $Q(\pi)$. 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 of our old method. 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.

We can define the functions P and Q using both addresses *and* ports. This generalizes both Hash locate and the method in the previous sections.

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

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, thus balancing the processing load more evenly over the hosts at each level of the network hierarchy.

5.3. 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. Centralized match-making (Example 3) is very *efficient*, but if the linchpin host crashes then match-making is impossible between any pair of nodes. It is one of the advantages of truly distributed algorithms that they may continue in the presence of faults. Below we distinguish two distinct criteria for robustness, apart from the problem of how, or whether it is still possible, to route the match-making messages to their destinations in the surviving subnetwork.

- Match-making should be *distributed* in the sense that node crashes do not take out the general facility of match-making in the surviving network (or as little as possible). This rules out a centralized match-maker, but the distributed Examples 1, 2, 4, 5, 6 satisfy this requirement in various degrees. It is lack of robustness according to this criterion that makes the efficient Hash Locate so fragile.

- The match-maker should be *redundant* in the sense that a bounded number of node crashes cannot prevent individual node pairs from the capability of making a match. For example, by chosing P and Q such that, for all $1 \le i, j \le n$,

$$|P(i) \bigcap Q(j)| \ge f + 1 \ ,$$

then the match-maker can tolerate up to f nodes being down at any time in the network.

Thus, robustness necessitates *inefficiency* twice: once because we have to *distribute* the algorithm and the next because we have to make it *redundant*. The most robust solution is, trivially, the most inefficient one: $P(i) \cap Q(j) = U$ $(1 \le i, j \le n)$.

Acknowledgement

Maria Klawe supplied Proposition 5. Jan van Leeuwen supplied Proposition 7. The comments of the referees resulted in improved presentation. [*LIST*]

Appendix: Examples

1. *Broadcasting*: $m = n + 1$, $s = s_{ave} = 1$.

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9

2. *Inverse Broadcasting*: $m = n + 1$, $s = s_{ave} = 1$.

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	1	2	3	4	5	6	7	8	9
3	1	2	3	4	5	6	7	8	9
4	1	2	3	4	5	6	7	8	9
5	1	2	3	4	5	6	7	8	9
6	1	2	3	4	5	6	7	8	9
7	1	2	3	4	5	6	7	8	9
8	1	2	3	4	5	6	7	8	9
9	1	2	3	4	5	6	7	8	9

3. *Centralized*: $m=2$, $s=n$ and $s_{ave}=1$.

	1	2	3	4	5	6	7	8	9
1	3	3	3	3	3	3	3	3	3
2	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3
4	3	3	3	3	3	3	3	3	3
5	3	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3	3
8	3	3	3	3	3	3	3	3	3
9	3	3	3	3	3	3	3	3	3

4. *Symmetric*: $m=2\sqrt{n}$, $s=s_{ave}=\sqrt{n}$.

	1	2	3	4	5	6	7	8	9
1	1	1	1	2	2	2	3	3	3
2	1	1	1	2	2	2	3	3	3
3	1	1	1	2	2	2	3	3	3
4	4	4	4	5	5	5	6	6	6
5	4	4	4	5	5	5	6	6	6
6	4	4	4	5	5	5	6	6	6
7	7	7	7	8	8	8	9	9	9
8	7	7	7	8	8	8	9	9	9
9	7	7	7	8	8	8	9	9	9

5. *Hierarchical*. The elements are ordered by $1,2,3<7$; $4,5,6<8$; $7,8<9$. The rendez-vous element is the least ancestor.

	1	2	3	4	5	6	7	8	9
1	1	7	7	9	9	9	7	9	9
2	7	2	7	9	9	9	7	9	9
3	7	7	3	9	9	9	7	9	9
4	9	9	9	4	8	8	9	8	9
5	9	9	9	8	5	8	9	8	9
6	9	9	9	8	8	6	9	8	9
7	7	7	7	9	9	9	7	9	9
8	9	9	9	8	8	8	9	8	9
9	9	9	9	9	9	9	9	9	9

6. For all $a,b,c \in \{0,1\}$, $P(abc) = \{axy \mid x,y \in \{0,1\}\}$ and $Q(abc) = \{xbc \mid x \in \{0,1\}\}$:

	000	001	010	011	100	101	110	111
000	000	001	010	011	000	001	010	011
001	000	001	010	011	000	001	010	011
010	000	001	010	011	000	001	010	011
011	000	001	010	011	000	001	010	011
100	100	101	110	111	100	101	110	111
101	100	101	110	111	100	101	110	111
110	100	101	110	111	100	101	110	111
111	100	101	110	111	100	101	110	111

7. Variant with $m(4,1)=2$ and $m(3,9)=2n$.

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	7
2	1	2	3	1	2	3	1	2	8
3	1	2	3	4	5	6	7	8	9
4	1	1	1	1	1	1	1	1	1
5	1	5	6	4	5	6	4	5	2
6	1	5	6	4	5	6	4	5	3
7	1	8	9	7	8	9	7	8	4
8	1	8	9	7	8	9	7	8	5
9	1	8	9	7	8	9	7	8	6

8. *Manhattan* : $m=2\sqrt{n}$, $s=s_{ave}=\sqrt{n}$.

	1	2	3	4	5	6	7	8	9
1	1	2	3	1	2	3	1	2	3
2	1	2	3	1	2	3	1	2	3
3	1	2	3	1	2	3	1	2	3
4	4	5	6	4	5	6	4	5	6
5	4	5	6	4	5	6	4	5	6
6	4	5	6	4	5	6	4	5	6
7	7	8	9	7	8	9	7	8	9
8	7	8	9	7	8	9	7	8	9
9	7	8	9	7	8	9	7	8	9