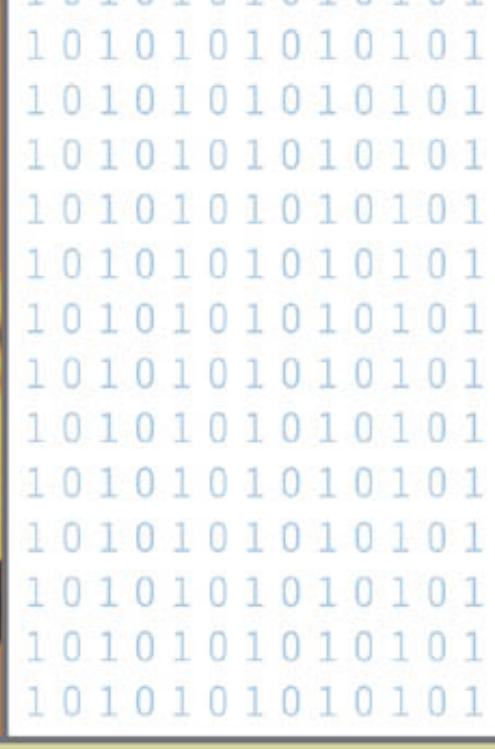
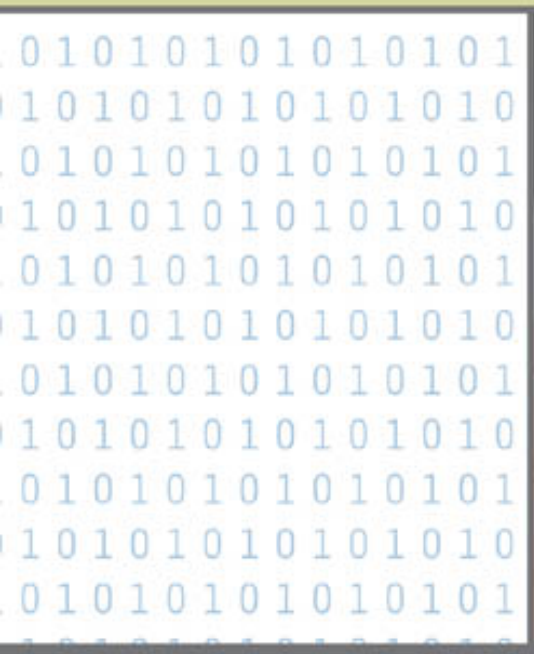


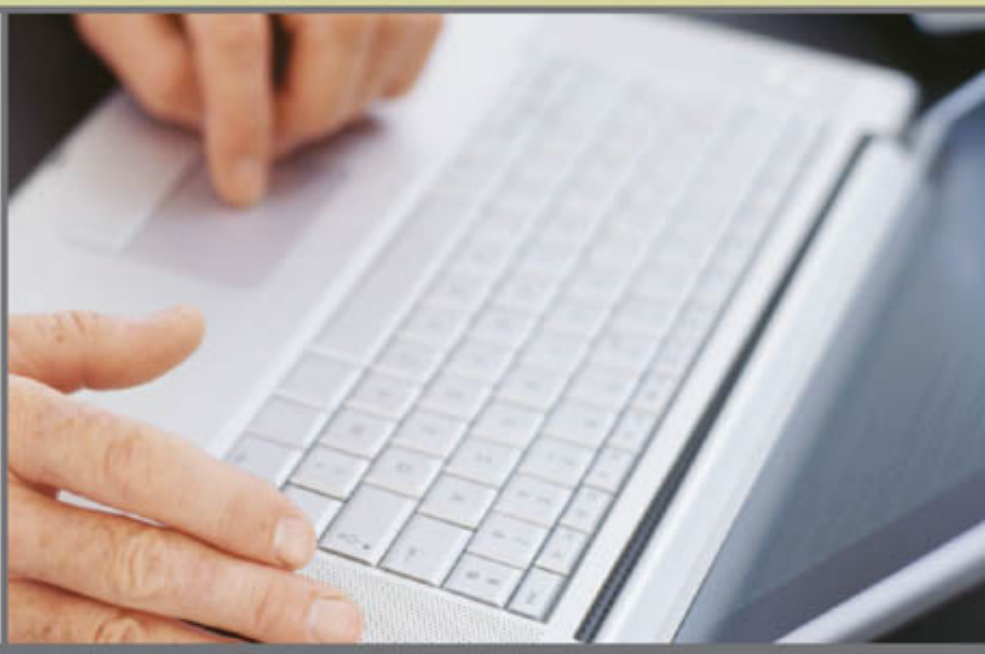
PEARSON NEW INTERNATIONAL EDITION

Distributed Systems
Principles and Paradigms

Andrew S. Tanenbaum Maarten Van Steen

Second Edition





ALWAYS LEARNING

PEARSON

Pearson New International Edition

Distributed Systems
Principles and Paradigms
Andrew S. Tanenbaum Maarten Van Steen
Second Edition

PEARSON

When the home agent receives a packet for the mobile host, it looks up the host's current location. If the host is on the current local network, the packet is simply forwarded. Otherwise, it is tunneled to the host's current location, that is, wrapped as data in an IP packet and sent to the care-of address. At the same time, the sender of the packet is informed of the host's current location. This principle is shown in Fig. 5-3. Note that the IP address is effectively used as an identifier for the mobile host.

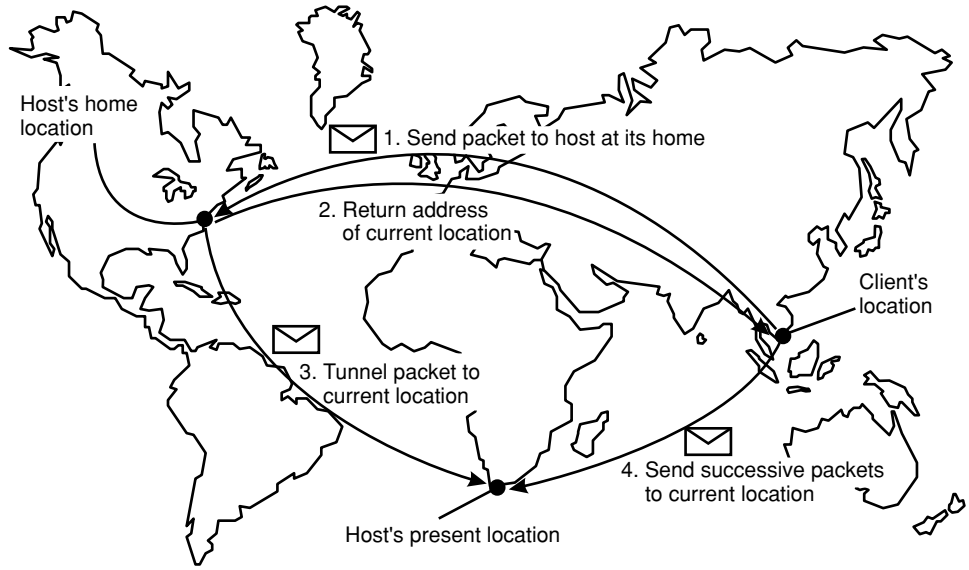


Figure 5-3. The principle of Mobile IP.

Fig. 5-3 also illustrates another drawback of home-based approaches in large-scale networks. To communicate with a mobile entity, a client first has to contact the home, which may be at a completely different location than the entity itself. The result is an increase in communication latency.

A drawback of the home-based approach is the use of a fixed home location. For one thing, it must be ensured that the home location always exists. Otherwise, contacting the entity will become impossible. Problems are aggravated when a long-lived entity decides to move permanently to a completely different part of the network than where its home is located. In that case, it would have been better if the home could have moved along with the host.

A solution to this problem is to register the home at a traditional naming service and to let a client first look up the location of the home. Because the home location can be assumed to be relatively stable, that location can be effectively cached after it has been looked up.

5.2.3 Distributed Hash Tables

Let us now take a closer look at recent developments on how to resolve an identifier to the address of the associated entity. We have already mentioned distributed hash tables a number of times, but have deferred discussion on how they actually work. In this section we correct this situation by first considering the Chord system as an easy-to-explain DHT-based system. In its simplest form, DHT-based systems do not consider network proximity at all. This negligence may easily lead to performance problems. We also discuss solutions for network-aware systems.

General Mechanism

Various DHT-based systems exist, of which a brief overview is given in Balakrishnan et al. (2003). The Chord system (Stoica et al., 2003) is representative for many of them, although there are subtle important differences that influence their complexity in maintenance and lookup protocols. As we explained briefly in Chap. 2, Chord uses an m-bit identifier space to assign randomly-chosen identifiers to nodes as well as keys to specific entities. The latter can be virtually anything: files, processes, etc. The number m of bits is usually 128 or 160, depending on which hash function is used. An entity with key k falls under the jurisdiction of the node with the smallest identifier $id \geq k$. This node is referred to as the *successor* of k and denoted as *succ(k)*.

The main issue in DHT-based systems is to efficiently resolve a key k to the address of *succ(k)*. An obvious nonscalable approach is let each node p keep track of the successor *succ(p+1)* as well as its predecessor *pred(p)*. In that case, whenever a node p receives a request to resolve key k, it will simply forward the request to one of its two neighbors—whichever one is appropriate—unless $pred(p) < k \leq p$ in which case node p should return its own address to the process that initiated the resolution of key k.

Instead of this linear approach toward key lookup, each Chord node maintains a **finger table** of at most m entries. If FT_p denotes the finger table of node p, then

$$FT_p[i] = succ(p+2^{i-1})$$

Put in other words, the i-th entry points to the first node succeeding p by at least 2^{i-1}. Note that these references are actually short-cuts to existing nodes in the identifier space, where the short-cutted distance from node p increases exponentially as the index in the finger table increases. To look up a key k, node p will then immediately forward the request to node q with index j in p's finger table where:

$$q = FT_p[j] \leq k < FT_p[j+1]$$

or $q = FT_p[1]$ when $p < k < FT_p[1]$. (For clarity, we ignore modulo arithmetic.)

To illustrate this lookup, consider resolving $k = 26$ from node 1 as shown Fig. 5-4. First, node 1 will look up $k = 26$ in its finger table to discover that this value is larger than $FT_1[5]$, meaning that the request will be forwarded to node $18 = FT_1[5]$. Node 18, in turn, will select node 20, as $FT_{18}[2] \leq k < FT_{18}[3]$. Finally, the request is forwarded from node 20 to node 21 and from there to node 28, which is responsible for $k = 26$. At that point, the address of node 28 is returned to node 1 and the key has been resolved. For similar reasons, when node 28 is requested to resolve the key $k = 12$, a request will be routed as shown by the dashed line in Fig. 5-4. It can be shown that a lookup will generally require O(log (N)) steps, with N being the number of nodes in the system.

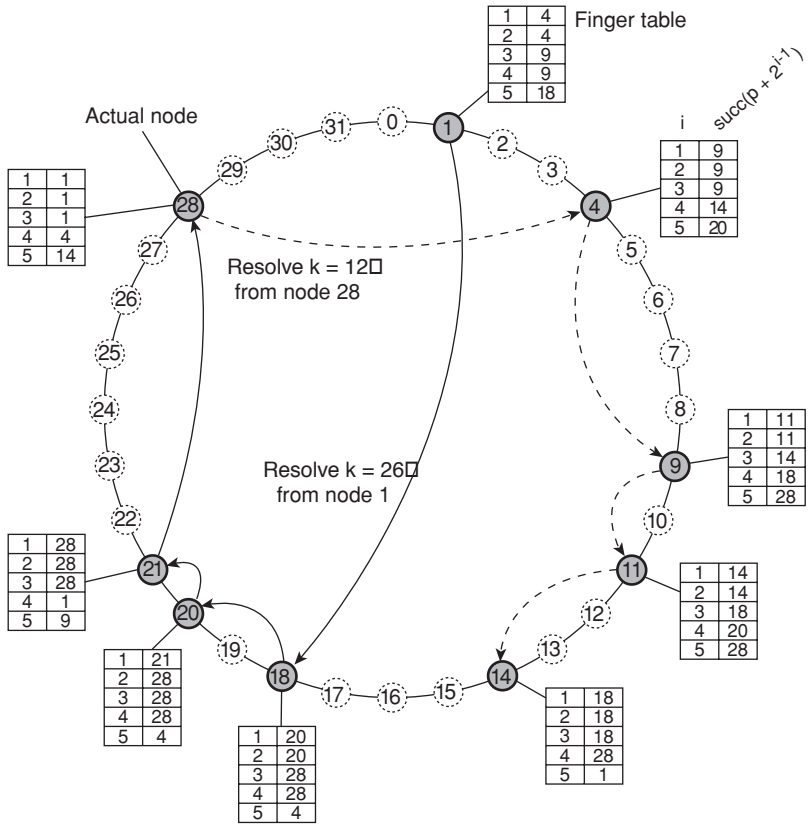


Figure 5-4. Resolving key 26 from node 1 and key 12 from node 28 in a Chord system.

In large distributed systems the collection of participating nodes can be expected to change all the time. Not only will nodes join and leave voluntarily, we also need to consider the case of nodes failing (and thus effectively leaving the system), to later recover again (at which point they join again).

Joining a DHT-based system such as Chord is relatively simple. Suppose node p wants to join. It simply contacts an arbitrary node in the existing system and requests a lookup for *succ(p+1)*. Once this node has been identified, p can insert itself into the ring. Likewise, leaving can be just as simple. Note that nodes also keep track of their predecessor.

Obviously, the complexity comes from keeping the finger tables up-to-date. Most important is that for every node q, $FT_q[1]$ is correct as this entry refers to the next node in the ring, that is, the successor of $q+1$. In order to achieve this goal, each node q regularly runs a simple procedure that contacts *succ(q+1)* and requests to return *pred(succ(q+1))*. If $q = pred(succ(q+1))$ then q knows its information is consistent with that of its successor. Otherwise, if q's successor has updated its predecessor, then apparently a new node p had entered the system, with $q < p \leq succ(q+1)$, so that q will adjust $FT_q[1]$ to p. At that point, it will also check whether p has recorded q as its predecessor. If not, another adjustment of $FT_q[1]$ is needed.

In a similar way, to update a finger table, node q simply needs to find the successor for $k = q + 2^{i-1}$ for each entry i. Again, this can be done by issuing a request to resolve *succ(k)*. In Chord, such requests are issued regularly by means of a background process.

Likewise, each node q will regularly check whether its predecessor is alive. If the predecessor has failed, the only thing that q can do is record the fact by setting *pred(q)* to "unknown." On the other hand, when node q is updating its link to the next known node in the ring, and finds that the predecessor of *succ(q+1)* has been set to "unknown," it will simply notify *succ(q+1)* that it suspects it to be the predecessor. By and large, these simple procedures ensure that a Chord system is generally consistent, only perhaps with exception of a few nodes. The details can be found in Stoica et al. (2003).

Exploiting Network Proximity

One of the potential problems with systems such as Chord is that requests may be routed erratically across the Internet. For example, assume that node 1 in Fig. 5-4 is placed in Amsterdam, The Netherlands; node 18 in San Diego, California; node 20 in Amsterdam again; and node 21 in San Diego. The result of resolving key 26 will then incur three wide-area message transfers which arguably could have been reduced to at most one. To minimize these pathological cases, designing a DHT-based system requires taking the underlying network into account.

Castro et al. (2002b) distinguish three different ways for making a DHT-based system aware of the underlying network. In the case of **topology-based assignment of node identifiers** the idea is to assign identifiers such that two nearby nodes will have identifiers that are also close to each other. It is not difficult to imagine that this approach may impose severe problems in the case of relatively simple systems such as Chord. In the case where node identifiers are sampled

from a one-dimensional space, mapping a logical ring to the Internet is far from trivial. Moreover, such a mapping can easily expose correlated failures: nodes on the same enterprise network will have identifiers from a relatively small interval. When that network becomes unreachable, we suddenly have a gap in the otherwise uniform distribution of identifiers.

With **proximity routing**, nodes maintain a list of alternatives to forward a request to. For example, instead of having only a single successor, each node in Chord could equally well keep track of r successors. In fact, this redundancy can be applied for every entry in a finger table. For node p, $FT_p[i]$ points to the first node in the range $[p+2^{i-1}, p+2^i-1]$. There is no reason why p cannot keep track of r nodes in that range: if needed, each one of them can be used to route a lookup request for a key $k > p+2^i-1$. In that case, when choosing to forward a lookup request, a node can pick one of the r successors that is closest to itself, but also satisfies the constraint that the identifier of the chosen node should be smaller than that of the requested key. An additional advantage of having multiple successors for every table entry is that node failures need not immediately lead to failures of lookups, as multiple routes can be explored.

Finally, in **proximity neighbor selection** the idea is to optimize routing tables such that the nearest node is selected as neighbor. This selection works only when there are more nodes to choose from. In Chord, this is normally not the case. However, in other protocols such as Pastry (Rowstron and Druschel, 2001), when a node joins it receives information about the current overlay from multiple other nodes. This information is used by the new node to construct a routing table. Obviously, when there are alternative nodes to choose from, proximity neighbor selection will allow the joining node to choose the best one.

Note that it may not be that easy to draw a line between proximity routing and proximity neighbor selection. In fact, when Chord is modified to include r successors for each finger table entry, proximity neighbor selection resorts to identifying the closest r neighbors, which comes very close to proximity routing as we just explained (Dabek at al., 2004b).

Finally, we also note that a distinction can be made between **iterative** and **recursive lookups**. In the former case, a node that is requested to look up a key will return the network address of the next node found to the requesting process. The process will then request that next node to take another step in resolving the key. An alternative, and essentially the way that we have explained it so far, is to let a node forward a lookup request to the next node. Both approaches have their advantages and disadvantages, which we explore later in this chapter.

5.2.4 Hierarchical Approaches

In this section, we first discuss a general approach to a hierarchical location scheme, after which a number of optimizations are presented. The approach we present is based on the Globe location service, described in detail in Ballintijn

(2003). An overview can be found in van Steen et al. (1998b). This is a general-purpose location service that is representative of many hierarchical location services proposed for what are called Personal Communication Systems, of which a general overview can be found in Pitoura and Samaras (2001).

In a hierarchical scheme, a network is divided into a collection of **domains**. There is a single top-level domain that spans the entire network. Each domain can be subdivided into multiple, smaller subdomains. A lowest-level domain, called a **leaf domain**, typically corresponds to a local-area network in a computer network or a cell in a mobile telephone network.

Each domain *D* has an associated directory node *dir(D)* that keeps track of the entities in that domain. This leads to a tree of directory nodes. The directory node of the top-level domain, called the **root (directory) node**, knows about all entities. This general organization of a network into domains and directory nodes is illustrated in Fig. 5-5.

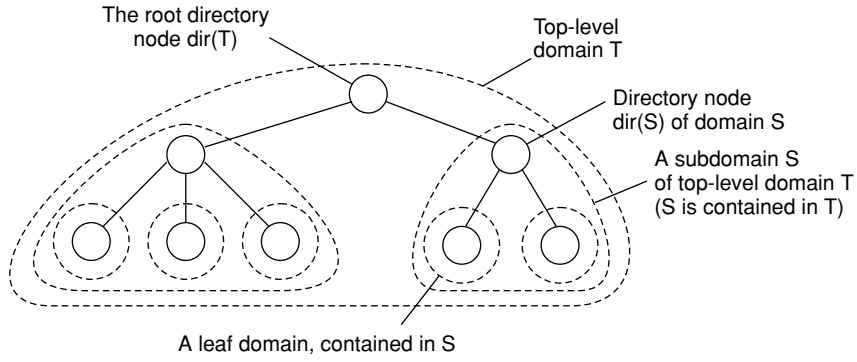


Figure 5-5. Hierarchical organization of a location service into domains, each having an associated directory node.

To keep track of the whereabouts of an entity, each entity currently located in a domain *D* is represented by a **location record** in the directory node *dir(D)*. A location record for entity *E* in the directory node *N* for a leaf domain *D* contains the entity's current address in that domain. In contrast, the directory node N' for the next higher-level domain D' that contains *D*, will have a location record for *E* containing only a pointer to *N*. Likewise, the parent node of N' will store a location record for *E* containing only a pointer to N'. Consequently, the root node will have a location record for each entity, where each location record stores a pointer to the directory node of the next lower-level subdomain where that record's associated entity is currently located.

An entity may have multiple addresses, for example if it is replicated. If an entity has an address in leaf domain D_1 and D_2 respectively, then the directory node of the smallest domain containing both D_1 and D_2, will have two pointers,