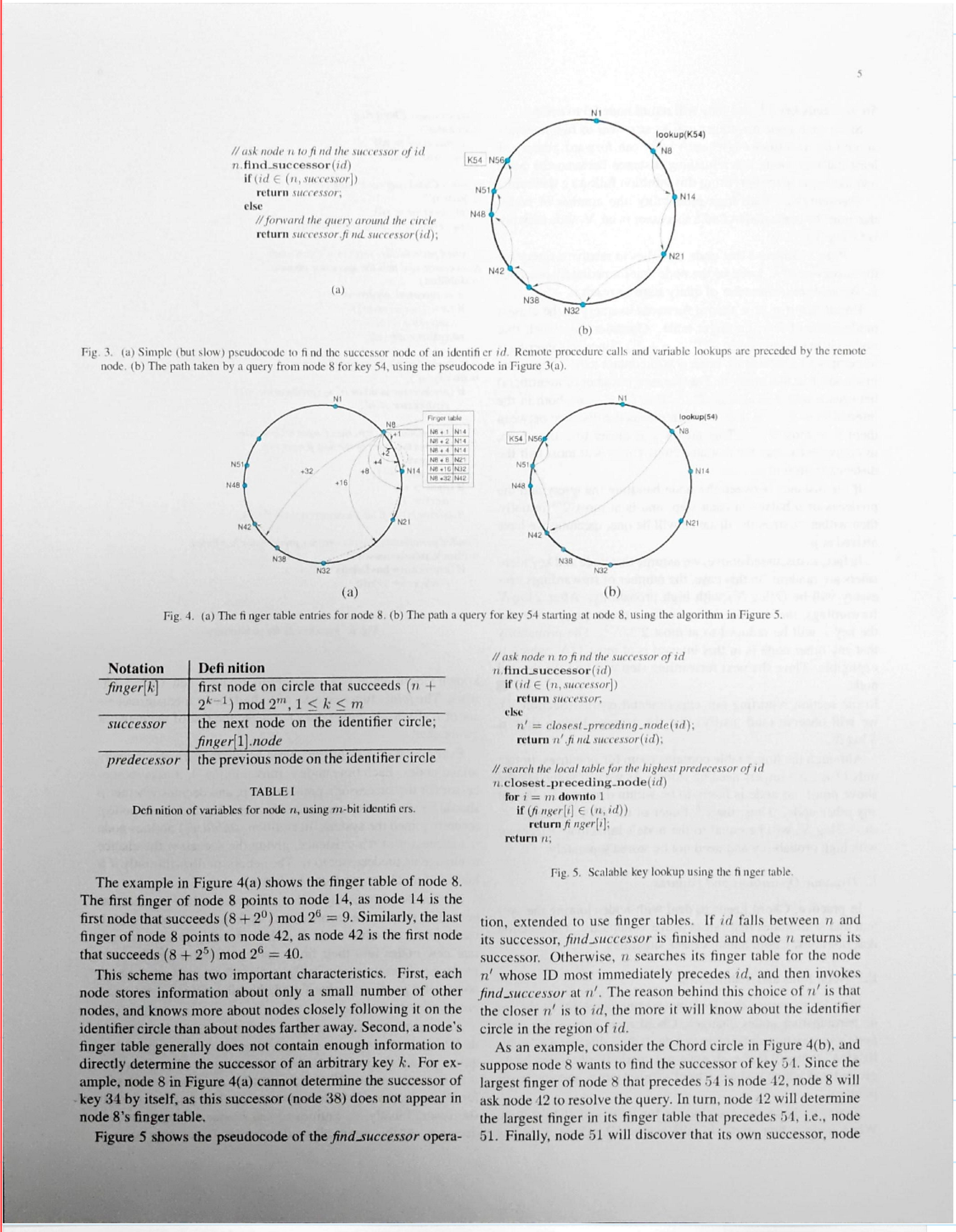
23-04-19 Mi

Wednesday, April 19, 2023 12:53 AM

GRABACIÓN DE CLASE

Chord




```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    return successor.find_successor(id);
```

(a)

(b)

Fig. 3. (a) Simple (but slow) pseudocode to find the successor node of an identifier id. Remote procedure calls and variable lookups are preceded by the remote node. (b) The path taken by a query from node 8 for key 54, using the pseudocode in Figure 3(a).

(a) (b)

Fig. 4. (a) The finger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

Notation	Definition
finger[k]	first node on circle that succeeds $(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
successor	the next node on the identifier circle; *finger*[1]*.node*
predecessor	the previous node on the identifier circle

TABLE I

Definition of variables for node n, using m-bit identifiers.

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n' = closest_preceding_node(id);
    return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n, id))
      return finger[i];
  return n;
```

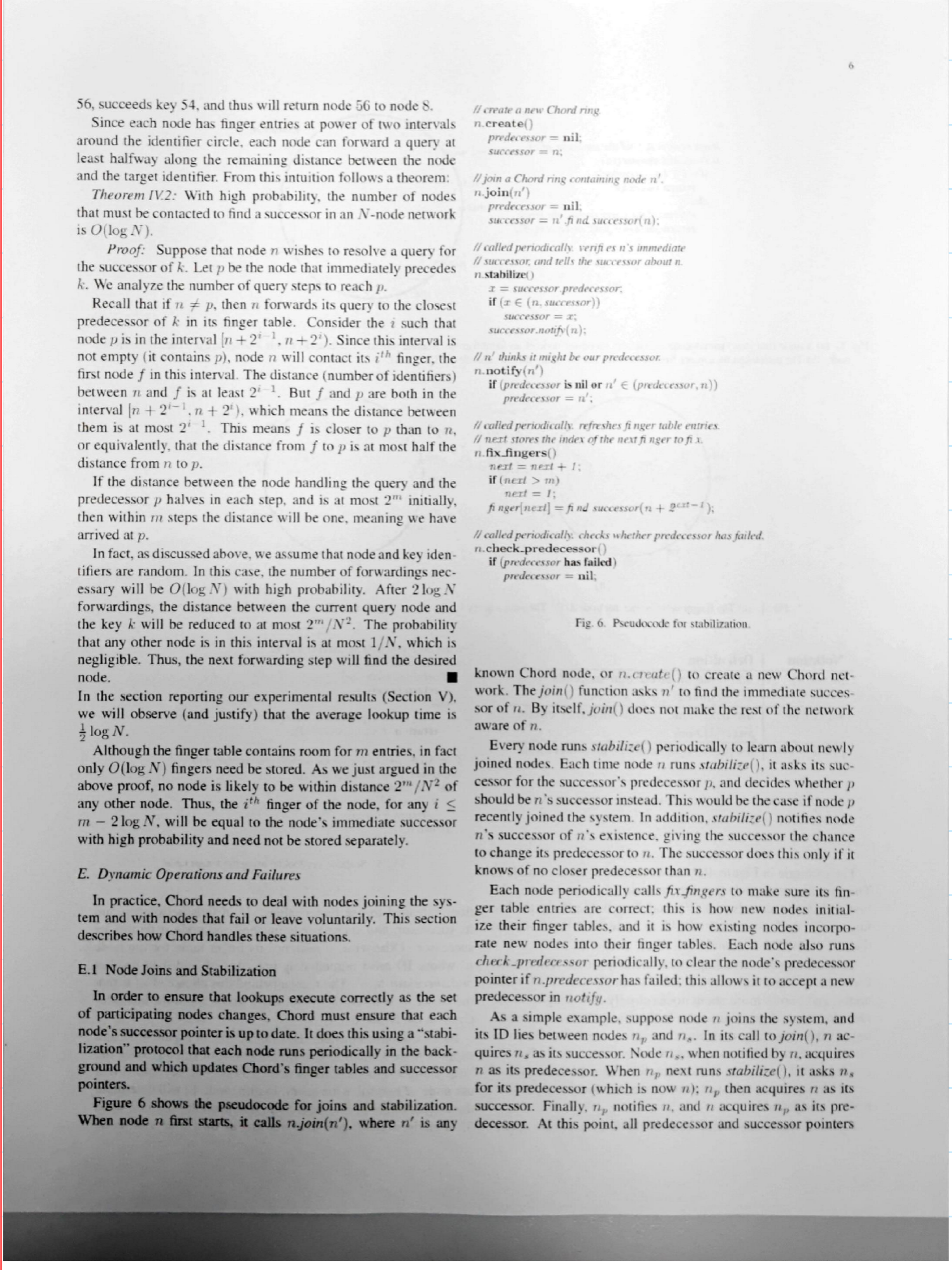
Fig. 5. Scalable key lookup using the finger table.

The example in Figure 4(a) shows the finger table of node 8. The first finger of node 8 points to node 14, as node 14 is the first node that succeeds $(8 + 2^0) \bmod 2^6 = 9$. Similarly, the last finger of node 8 points to node 42, as node 42 is the first node that succeeds $(8 + 2^5) \bmod 2^6 = 40$.

This scheme has two important characteristics. First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away. Second, a node's finger table generally does not contain enough information to directly determine the successor of an arbitrary key k. For example, node 8 in Figure 4(a) cannot determine the successor of key 34 by itself, as this successor (node 38) does not appear in node 8's finger table.

Figure 5 shows the pseudocode of the *find_successor* operation, extended to use finger tables. If id falls between n and its successor, *find_successor* is finished and node n returns its successor. Otherwise, n searches its finger table for the node n' whose ID most immediately precedes id, and then invokes *find_successor* at n'. The reason behind this choice of n' is that the closer n' is to id, the more it will know about the identifier circle in the region of id.

As an example, consider the Chord circle in Figure 4(b), and suppose node 8 wants to find the successor of key 54. Since the largest finger of node 8 that precedes 54 is node 42, node 8 will ask node 42 to resolve the query. In turn, node 42 will determine the largest finger in its finger table that precedes 54, i.e., node 51. Finally, node 51 will discover that its own successor, node



56, succeeds key 54, and thus will return node 56 to node 8.

Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target identifier. From this intuition follows a theorem:

Theorem IV.2: With high probability, the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.

Proof: Suppose that node n wishes to resolve a query for the successor of k. Let p be the node that immediately precedes k. We analyze the number of query steps to reach p.

Recall that if $n \neq p$, then n forwards its query to the closest predecessor of k in its finger table. Consider the i such that node p is in the interval $[n + 2^{i-1}, n + 2^i)$. Since this interval is not empty (it contains p), node n will contact its i^{th} finger, the first node f in this interval. The distance (number of identifiers) between n and f is at least 2^{i-1}. But f and p are both in the interval $[n + 2^{i-1}, n + 2^i)$, which means the distance between them is at most 2^{i-1}. This means f is closer to p than to n, or equivalently, that the distance from f to p is at most half the distance from n to p.

If the distance between the node handling the query and the predecessor p halves in each step, and is at most 2^m initially, then within m steps the distance will be one, meaning we have arrived at p.

In fact, as discussed above, we assume that node and key identifiers are random. In this case, the number of forwardings necessary will be $O(\log N)$ with high probability. After $2 \log N$ forwardings, the distance between the current query node and the key k will be reduced to at most $2^m/N^2$. The probability that any other node is in this interval is at most $1/N$, which is negligible. Thus, the next forwarding step will find the desired node. ■

In the section reporting our experimental results (Section V), we will observe (and justify) that the average lookup time is $\frac{1}{2} \log N$.

Although the finger table contains room for m entries, in fact only $O(\log N)$ fingers need be stored. As we just argued in the above proof, no node is likely to be within distance $2^m/N^2$ of any other node. Thus, the i^{th} finger of the node, for any $i \leq m - 2 \log N$, will be equal to the node's immediate successor with high probability and need not be stored separately.

E. Dynamic Operations and Failures

In practice, Chord needs to deal with nodes joining the system and with nodes that fail or leave voluntarily. This section describes how Chord handles these situations.

E.1 Node Joins and Stabilization

In order to ensure that lookups execute correctly as the set of participating nodes changes, Chord must ensure that each node's successor pointer is up to date. It does this using a "stabilization" protocol that each node runs periodically in the background and which updates Chord's finger tables and successor pointers.

Figure 6 shows the pseudocode for joins and stabilization. When node n first starts, it calls $n.join(n')$, where n' is any known Chord node, or $n.create()$ to create a new Chord network. The *join*() function asks n' to find the immediate successor of n. By itself, *join*() does not make the rest of the network aware of n.

```
// create a new Chord ring.
n.create()
  predecessor = nil;
  successor = n;

// join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
  x = successor.predecessor;
  if (x ∈ (n, successor))
    successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n' ∈ (predecessor, n))
    predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
  next = next + 1;
  if (next > m)
    next = 1;
  finger[next] = find_successor(n + 2^(next-1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
  if (predecessor has failed)
    predecessor = nil;
```

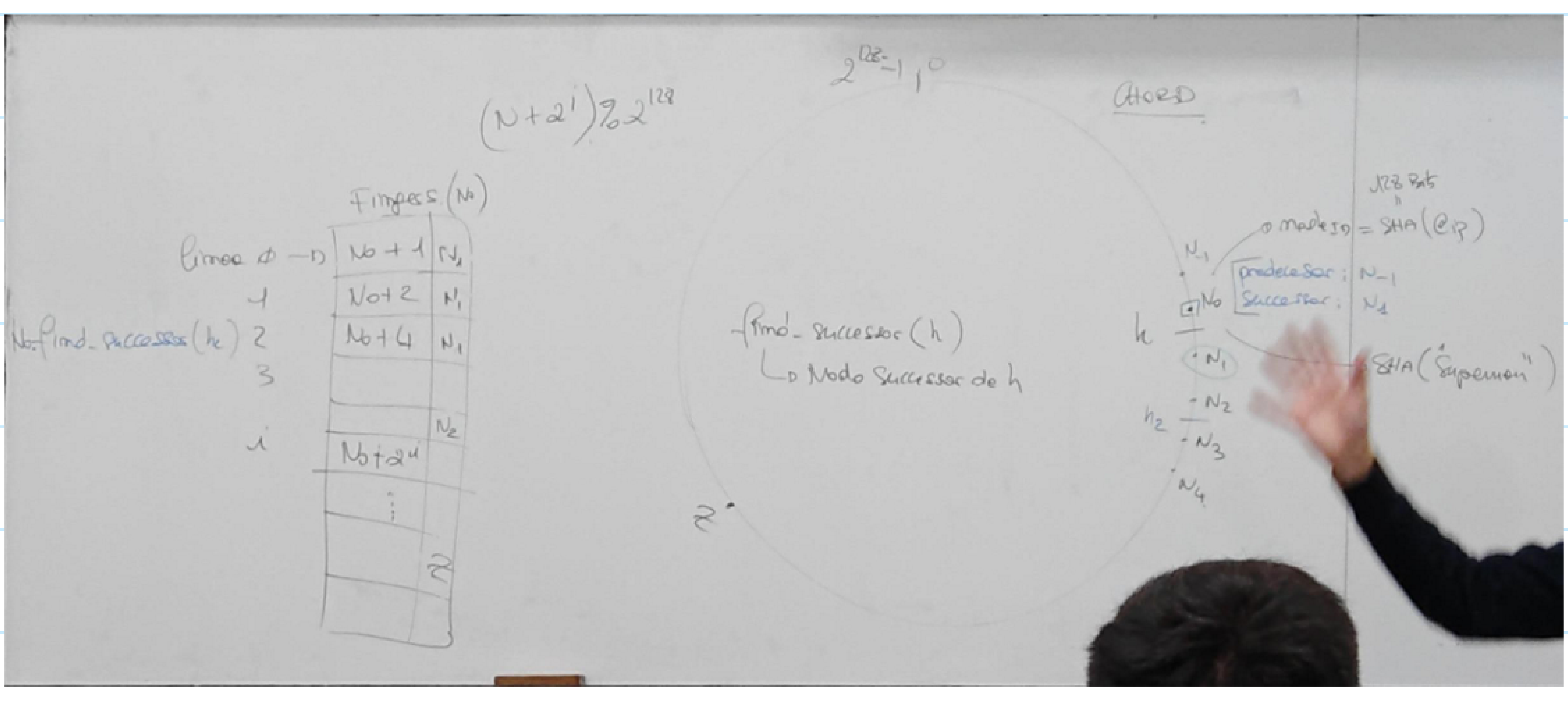
Fig. 6. Pseudocode for stabilization.

Every node runs *stabilize*() periodically to learn about newly joined nodes. Each time node n runs *stabilize*(), it asks its successor for the successor's predecessor p, and decides whether p should be n's successor instead. This would be the case if node p recently joined the system. In addition, *stabilize*() notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n.

Each node periodically calls *fix_fingers* to make sure its finger table entries are correct; this is how new nodes initialize their finger tables, and it is how existing nodes incorporate new nodes into their finger tables. Each node also runs *check_predecessor* periodically, to clear the node's predecessor pointer if *n.predecessor* has failed; this allows it to accept a new predecessor in *notify*.

As a simple example, suppose node n joins the system, and its ID lies between nodes n_p and n_s. In its call to *join*(), n acquires n_s as its successor. Node n_s, when notified by n, acquires n as its predecessor. When n_p next runs *stabilize*(), it asks n_s for its predecessor (which is now n); n_p then acquires n as its successor. Finally, n_p notifies n, and n acquires n_p as its predecessor. At this point, all predecessor and successor pointers

ruteo → fine, se rutea al nodo sucesor, nunca el mas cercano



todo ruteo llega al sucesor del valor de hash de "superman"