General mechanism

Many DHT-based systems have been developed in the past decade, with the Chord system [Stoica et al., 2003] being a typical representative. 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 $succ(k)$. To keep our notation simple and consistent, in the following we refer to a node with identifier p as node p.

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 to 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** containing $s \leq m$ entries. If $\mathsf{FT_p}$ denotes the finger table of node p, then

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

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 shortcuts 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:

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

or $q = \mathsf{FT_p}[1]$ when $p < k < \mathsf{FT_p}[1]$. (For clarity, we ignore modulo arithmetic.) Note that when the finger-table size s is equal to 1, a Chord lookup corresponds to naively traversing the ring linearly as we just discussed.

To illustrate this lookup, consider resolving $k = 26$ from node 1 as shown in Figure 5.4. First, node 1 will look up $k = 26$ in its finger table to discover that this value is larger than $\mathsf{FT_1}[5]$, meaning that the request will be forwarded to node $18 = \mathsf{FT_1}[5]$. Node 18, in turn, will select node 20, as $\mathsf{FT_{18}}[2] \leq k < \mathsf{FT_{18}}[3]$. 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 Figure 5.4. It can be shown that a lookup will generally require $\mathcal{O}(\log(N))$ steps, with N being the number of nodes in the system.

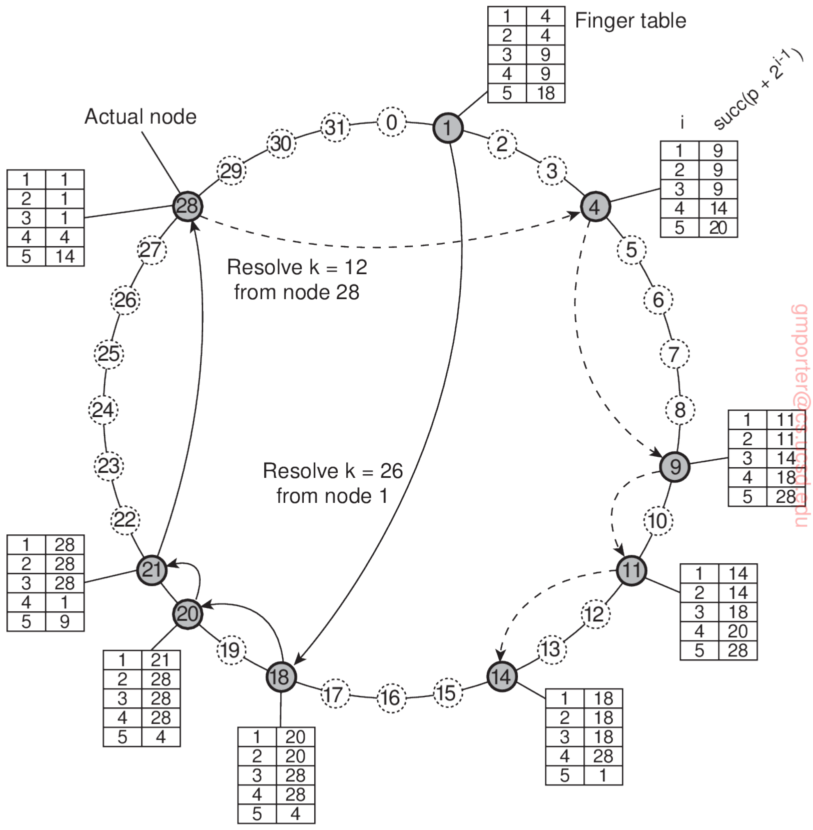

Figure 5.4: Resolving key 26 from node 1 and key 12 from node 28 ina Chord system.

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 rejoin).

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, $\mathsf{FT}_\mathsf{q}[1]$ 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 $\mathsf{FT_q}[1]$ to p. At that point, it will also check whether p has recorded q as its predecessor. If not, another adjustment of $\mathsf{FT_q}[1]$ 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].

Note 5.4 (Advanced: Chord in Python)

```python
class ChordNode:
  def finger(self, i):
    succ = (self.nodeID + pow(2, i-1)) % self.MAXPROC    # succ(p+2^(i-1))
    lwbi = self.nodeSet.index(self.nodeID)               # self in nodeset
    upbi = (lwbi + 1) % len(self.nodeSet)                # next neighbor
    for k in range(len(self.nodeSet)):                   # process segments
      if self.inbetween(succ, self.nodeSet[lwbi]+1, self.nodeSet[upbi]+1):
        return self.nodeSet[upbi]                        # found successor
      (lwbi,upbi) = (upbi, (upbi+1) % len(self.nodeSet)) # next segment

  def recomputeFingerTable(self):
    self.FT[0]  = self.nodeSet[self.nodeSet.index(self.nodeID)-1] # Pred.
    self.FT[1:] = [self.finger(i) for i in range(1,self.nBits+1)] # Succ.

  def localSuccNode(self, key):
    if self.inbetween(key, self.FT[0]+1, self.nodeID+1): # in (FT[0],self]
      return self.nodeID                                 # responsible node
    elif self.inbetween(key, self.nodeID+1, self.FT[1]): # in (self,FT[1]]
      return self.FT[1]                                  # succ. responsible
    for i in range(1, self.nBits+1):                     # rest of FT
      if self.inbetween(key, self.FT[i], self.FT[(i+1) % self.nBits]):
        return self.FT[i]                                # in [FT[i],FT[i+1])
```

Figure 5.5: The essence of a Chord node expressed in Python.

Coding Chord in Python is remarkably simple. Again, omitting many of the nonessential coding details, the core of the behavior of a Chord node can be described as shown in Figure 5.5. The function `finger(i)` computes $succ(i)$

for the given node. All nodes known to a specific Chord node are collected in a local set `nodeSet`, which is sorted by node identifier. The node first looks up its own position in this set, and that of its right-hand neighbor. The operation `inbetween(k,l,u)` computes if $k \in [l, u)$, taking modulo arithmetic into account. Computing `inbetween(k,l+1,u+1)` is therefore the same as testing whether $k \in (l, u]$. We thus see that `finger(i)` returns the largest existing node identifier less or equal to i.

Every time a node learns about a new node in the system (or discovers that one has left), it simply adjusts the local `nodeSet` and recomputes its finger table by calling `recomputeFingerTable`. The finger table itself is implemented as a local table `FT`, with `FT[0]` pointing to the node's predecessor. `nBits` indicates the number of bits used for node identifiers and keys.

The core of what a node does during a lookup is encoded in `localSuccNode(k)`. When handed a key k, it will either return itself, its immediate successor `FT[1]`, or go through the finger table to search the entry satisfying `FT[i]` $\leq k <$ `FT[i+1]`. The code does not show what is done with the returned value (which is a node identifier), but typically in an *iterative scheme*, the referenced node will be contacted to continue looking up k, unless the node had returned itself as the one being responsible for k. In a *recursive scheme*, the node itself will contact the referenced node.

Note 5.5 (Advanced: 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 Figure 5.5 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. [2002a] 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