10 Peer-to-peer systems I

Even though they were introduced only a few years ago, peer-to-peer file sharing systems are now one of the most popular Internet applications and have become a major source of Internet traffic. Thus, it is extremely important that these systems be scalable. We will present three alternative designs for peer-to-peer networks, following different philosophies. The first design, called *SPON* (supervised peer-to-peer overlay network), has a special server called *master* that manages the peer-to-peer network. This approach is useful for the organization of storage pools or multicast groups. The second design, called *Chord*, is completely distributed in that any node may introduce new nodes to the system. However, the topology only allows to give probabilistic guarantees about its diameter and congestion. Finally, we present a design, called *Hyperring*, that allows to give deterministic guarantees about properties such as diameter and congestion.

10.1 The SPON system

In this section we present an overlay network that is controlled by a special server called *master*. To simplify the description, we assume that all users that want to join or leave the system have to contact the master. With a bit more effort, one can extend the system so that also unannounced departures of nodes can be handled. The task of the master is to establish an overlay network (i.e. a structure of logical connections) between the users (also simply called nodes in the following) so that the following properties are fulfilled:

- *The nodes in the overlay network must have a small degree.* This would allow to keep the maintenance overhead low.
- *The diameter of the overlay network must be small.* This is important to ensure a fast exchange of information and a fast repair of the overlay network.
- *All arrivals and departures of nodes can be handled by the master with a small amount of messages that can ideally be sent in a single pass.* This is important to make sure that the master can handle a large network.

We will show that the SPON network achieves all of these goals. We start with the description of the basic structure of SPON.

The master uses a special *forest cache* with cache slots numbered $0, 1, 2, \ldots$ to organize the nodes. Each cache position can be occupied by an ID of a node. The positions are organized in pairs of trees of increasing depth. More precisely, we will enforce the rule that for all $i \geq 0$ the *i-level pair* $(2i, 2i+1)$ is reserved for nodes that are roots of binary trees of depth i.

The following events have to be considered to keep the SPON network up to date:

1. A domain node v joins the network.
2. A domain node v currently stored in a cache slot leaves the network.
3. A domain node v not currently stored in a cache slot leaves the network.

Recall that we assume that these events are announced to the master, i.e. if a node joins or wants to leave the network, it will notify the master about this. We want to handle these events in a way that the following invariants are kept:

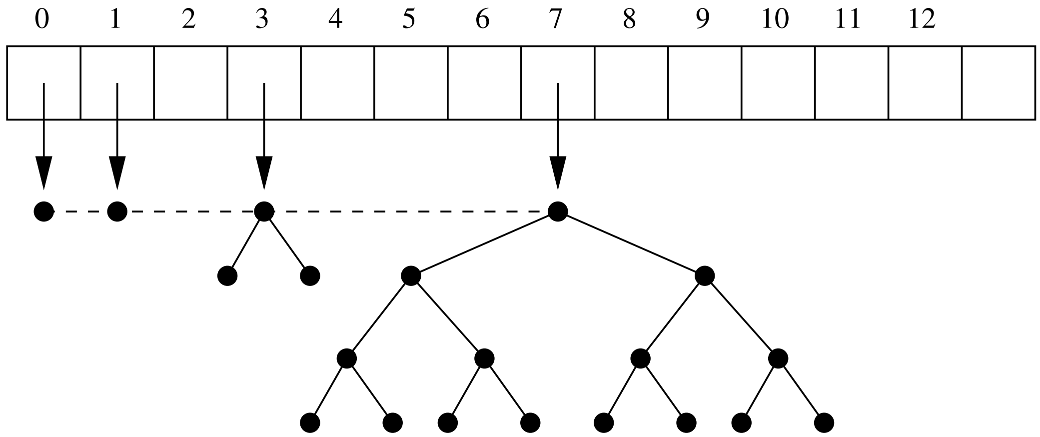

Figure 1: An example of a SPON network.

1. Every node v stored in an i-level pair is the root of a *complete* binary tree of trusted nodes of depth i. If $i \geq 1$, the master also knows v's two children.

2. Every node stored in the cache does not have a father. On the other hand, every node v that is not stored in the cache must have a father w whose level is by one larger than v.

3. Every node knows its level, its father (if it exists), and its two children (if they exist).

Assuming that the first invariant is true, it would immediately imply the following result.

Claim 10.1 *The largest occupied i-level pair must fulfill $i \leq \lfloor \log(n+1) \rfloor - 1$, where n is the current number of users.*

Thus, if the invariants are true, the space necessary to organize the network would be very low. Next we show that the three events above can be handled so that the invariants are always fulfilled.

Case 1: A node v joins the network

First, we look for the i-level pair with largest i that is completely occupied. If there is no such i, we simply insert v in an available slot in the 0-level pair. Otherwise, let w_1 and w_2 be the two domain nodes occupying the slots of the largest full i-level pair. Then we establish the connections $\{v, w_1\}$ and $\{v, w_2\}$ (by sending corresponding messages to v, w_1, and w_2), remove w_1 and w_2 from the i-level pair, and insert v in an available slot in the $(i+1)$-level pair. Certainly, this strategy implies that the rules for the arrival of a new node preserve the invariants and require only a constant number of messages that can be sent in one pass.

Case 2: A domain node v in the cache leaves the network

In this case, v will be deleted from the slot. If v has children u_1 and u_2, we will incorporate them into the cache in the following way:

Let u_1 and u_2 be i-level nodes. If both slots in the i-level pair are empty, we can place u_1 and u_2 in these slots and we are done.

So suppose that at least one of the i-level nodes is empty. In this case, we take the minimum j for which a j-level pair is occupied by at least one node, say x. Such a pair must clearly exist with $j \leq i$. We take x and use it as a replacement for v to form an $(i+1)$-level tree with u_1 and u_2. If $j > 0$, then we take in addition to this the two children of x, y_1 and y_2, and insert them into the two (empty) $(j-1)$-level slots.

To perform all these changes, the master may have to send messages to nodes u_1, u_2, x, y_1 and y_2; and u_1, u_2, y_1 and y_2 may have to report about their children. Thus, also the rules for the departure of a cache node preserve the invariants and require only a constant number of messages that can be sent in one pass.

Case 3: A domain node v not in the cache leaves the network

Suppose that v is an i-level node. Let w be the father of v and u_1 and u_2 be the children of v (if they exist).

Let j be the lowest j-level pair with a node, say x (such a j exists because w is still in the system). If $j = 0$, we simply remove x from the slot and establish the connections $\{w, x\}$, $\{x, u_1\}$ and $\{x, u_2\}$. Otherwise, we check whether x is actually w. If so, and $i = 0$ (i.e.u_1 and u_2 do not exist), then we simply move w and its remaining child to the two empty 0-level slots. If $i = 1$, we place the remaining child of w in a 1-level slot, establish the connections $\{w, u_1\}$ and $\{w, u_2\}$, and place w in the other (empty) 1-level slot. Otherwise (i.e. $i \geq 2$), we move w to a 0-level slot, the remaining child of w to an empty i-level slot, and u_1 and u_2 to the empty $(i-1)$-level slots.

If x is not equal to w, we move the children of x to the (empty) $(j-1)$-level pair and then remove x to establish the connections $\{w, x\}$, $\{x, u_1\}$ and $\{x, u_2\}$.

To perform these changes, the master may have to send messages to nodes w, x, u_1, and u_2, which can all be done in one pass.

Combining all three cases, we arrive at the following result.

Theorem 10.2 *The SPON scheme ensures that the master requires a cache of size at most $2\lfloor\log(n+1)\rfloor$, where n is the current number of nodes. Furthermore, every node has a degree of at most 3 and a distance to a cache node of at most $\lfloor\log(n+1)\rfloor$. Every arrival and departure of a node can be handled with a constant number of messages that can be sent in a single pass.*

To make sure that the forest of trees is connected to a single tree, we simply have to add edges between the cache nodes. This can be maintained so that the maximum degree of a node is at most 5 and all other properties in Theorem 10.2 still hold.

Communicating in SPON

The SPON system does not allow efficient communication between its members, because many message may have to go via the root. However, it is very useful to broadcast information from the master to the other nodes, the master has to forward the information only to one node and every node has to forward the information to at most three other nodes (two because it may have two children, and one to forward it to a lower cache slot).

10.2 The Chord system

In the Chord system [1] there is no master supervising the construction of the network. Instead, a new node may contact any node in the system to be included. The main purpose of Chord is distributed data management. This is done as follows:

Chord is based on a one-way hash function (e.g. SHA-1) mapping node and data IDs to real values in $[0, 1)$. Let us call this function h in the following. We will assume that h has all properties of a random function (although this is a questionable assumption), i.e. the hash values of the nodes and data elements are spread uniformly at random in $[0, 1)$.

A basic requirement of Chord is that the nodes have to be arranged in a doubly linked cycle, and the nodes are ordered on this cycle according to their hash function values. In addition to this, every node v has edges to nodes $p_i(v)$ with

$$p_i(v) = \mathsf{argmin}\{w \in \mathsf{Chord} \mid h(w) \geq h(v) + 1/2^i\}$$

for every $i \geq 1$. To keep this property, every node has to be inserted in the following way:

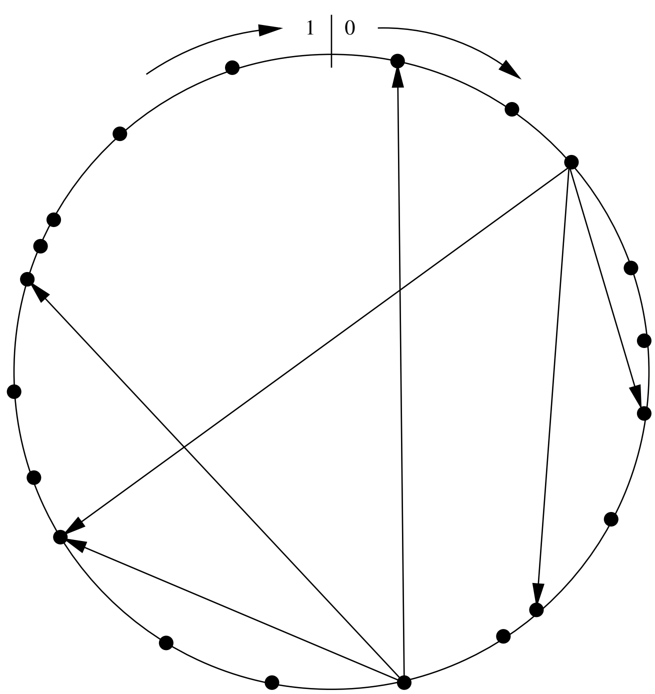

Figure 2: An example of a Chord network (only short-cut pointers for two nodes are given).

Insertion of new nodes

Suppose that a new node v contacts node $w \in \mathsf{Chord}$ to join the system. Then v will first be forwarded to its predecessor concerning $h(v)$ in Chord, say u. (How this is done will be explained below.) Once v has reached u, u will include v between itself and its successor u' in the doubly linked cycle. Also, u' will notify all nodes w' with $p_i(w') = u'$ that should now point to v about this. Afterwards, v uses u's pointers $p_i(u)$ to establish the short-cut pointers $p_i(v)$. For each i this is simply done by following the doubly linked cycle starting with $p_i(u)$ until the right $p_i(v)$ has been found.

Claim 10.3 *Inserting a new node requires $O(\log n)$ time and message transmissions on expectation and $O(\log^2 n)$ time and message transmissions with high probability.*

Proof. Due to symmetry reasons, the expected fraction of the $[0,1)$ ring owned by any particular node v is $1/n$ (i.e. all edges with an endpoint in that area will be assigned to v). Furthermore, the largest possible interval in the $[0,1)$ ring without a node is at most $O((\log n)/n)$ with high probability, because the probability that no node chooses a value in some fixed interval of size $(c \ln n)/n$ is equal to

$$\left(1 - \frac{c \ln n}{n}\right)^n \leq \mathrm{e}^{-\frac{c \ln n}{n} \cdot n} = n^{-c} .$$

Hence, the maximum fraction of the $[0,1)$ ring owned by a node is $O((\log n)/n)$ with high probability.

If a new node receives an area of size k, then on expectation and with high probability it will get $O(k \log n)$ edges. This completes the claim. $\sqcap$

Deletion of a node

If a node v wants to leave the system, it first notifies all nodes u with $p_i(u) = v$ to point to v's successor, and then it removes itself from the doubly linked cycle by connecting its predecessor and successor. Also here we get:

Claim 10.4 *Deleting a new node requires $O(\log n)$ time and message transmissions on expectation and $O(\log^2 n)$ time and message transmissions with high probability.*

Communicating in Chord

Suppose a message M has to be sent to node u. Then the following strategy is used:

Suppose that v is the current location of M. As long as $h(u) \notin [h(v), h(\text{succ}(v)))$ (in which case M has reached its destination), M moves to the node w with maximum i so that $w = p_i(v)$ and $h(w) \leq h(u)$ (if there is no such w, choose the w among v's successors with maximum $h(w)$).

This strategy yields the following result.

Claim 10.5 *For any source-destination pair (s,t), the number of edges to be traversed to send a message from s to t is at most $O(\log n)$ with high probability.*

Proof. Suppose that the message currently has a distance of δ (w.r.t. the $[0,1)$ interval) from $h(t)$. Since there exists an i with $\delta/2 \leq 1/2^i \leq \delta$, the w with largest $h(w) \leq h(t)$ decreases the distance by at least $1/2$. Hence, in $\log n$ steps the distance to $h(t)$ can be at most $1/n$. Since in a $1/n$-range there can be at most $O(\log n)$ nodes with high probability, altogether the number of nodes traversed is at most $O(\log n)$ with high probability. $\sqcap$

Data management in Chord

The data management works very similar to the nearest neighbor strategy presented in Section 8: Every data item d is placed at the node v with

$$v = \mathsf{argmax}\{w \mid h(w) \leq h(d)\} .$$

This strategy shares all the properties of the nearest neighbor strategy. That is, it is faithful, space-efficient, time-efficient, and 2-competitive concerning adaptivity. Hence, using this strategy, data will on expectation be evenly distributed among the nodes and at most a factor of 2 more data than necessary has to be replaced if a node leaves.

When a new data item has to be inserted, or when a data item has to be located, we can use the same strategy as for the insertion of a node to locate the node responsible for it. Hence, $O(\log n)$ communication steps suffice to do this with high probability.

Fault tolerance

In order to achieve a high fault tolerance, every node maintains pointers to its $O(\log n)$ successors on the doubly linked cycle. In this case it holds:

Claim 10.6 *Even if ϵn nodes leave at the same time for some constant $\epsilon < 1$, the probability that a node loses all of its successors is polynomially small in n.*

Proof. If ϵn nodes leave the system, the probability that all $c \log n$ successors of a node are among them is

$$\frac{\epsilon n}{n} \cdot \frac{\epsilon n - 1}{n - 1} \cdot \ldots \cdot \frac{\epsilon n - (c \log n - 1)}{n - (c \log n - 1)} \leq \epsilon^{c \log n} = n^{-c \log(1/\epsilon)} .$$

This is polynomially small if c is sufficiently large compared to ϵ. $\sqcap$

References

[1] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM '01*, pages 149–160, 2001.