10 Supervised Overlay Networks

Every application run on multiple machines needs a mechanism that allows the machines to exchange information. An easy way of solving this problem is that every machine knows the domain name or IP address of every other machine. While this may work well for a small number of machines, large-scale distributed applications such as file sharing or grid computing systems need a different, more scalable approach: instead of forming a clique (where everybody knows everybody else), each machine should only be required to know some small subset of other machines. This graph of knowledge can be seen as a logical network interconnecting the machines, which is also known as an *overlay network*. Ideally, an overlay network should have a low degree, a small diameter, and a high expansion, but the question is how to realize such an overlay network in a distributed environment where peers may continuously enter and leave the system. This will be the topic of our investigations for the coming weeks.

We start in this section with the study of *supervised* overlay networks. In a supervised overlay network, the topology is under the control of a special machine (or node) called *supervisor*. All nodes that want to join or leave the network have to declare this to the supervisor, and the supervisor will then take care of integrating them into or removing them from the network. All other overlay functions, however, may be executed without involving the supervisor.

10.1 Basic overlay network operations

Any overlay network needs an interface to the application layer to be useful, i.e. it needs to offer a collection of operations that can be called from the application layer. The most common overlay network operations are:

- $p.$JOIN(q): peer p in the system receives a request to join the system from a peer q.
- $p.$LEAVE$()$: peer p leaves the system.
- $p.$SEARCH(key): peer p searches for the peer responsible for key.
- $p.$BROADCAST(m): peer p broadcasts message m to all other peers in the network.

In the following, we assume that nodes always depart *gracefully*, i.e. they wait for the LEAVE operation to complete before leaving the system. This tremendously simplifies our task of designing a LEAVE operation.

Overlay network operations require sending messages between nodes to be effective. Each message received by a node may trigger another message sent out to another node. Hence, the set of messages caused by an overlay network operation can be seen as a *message graph* $M = (V, E)$ where each node in V represents a message and each edge (m_1, m_2) represents an event where a message m_2 is sent based on some prior message m_1.

This message graph can be used to define some important performance measures for operations, such as:

- **Dilation**: What is the longest directed path a message graph of an operation can have? This influences the time an operation needs to complete. So ideally, the dilation should be as small as possible.

- **Congestion**: What is the maximum number of messages a node is involved with when executing an overlay network operation? If we look at multiple operations, then the congestion measures the maximum number of messages a node is involved with when handling all of these operations. Naturally, the congestion should be as small as possible.

- **Work**: What is the maximum possible size the message graph of an operation may have? That is, how many messages in total may have to be sent for that operation? Also the work should be as small as possible.

- **Space**: How much space (for message buffers, tables, or any kind of control information) is necessary in the nodes to perform the operation?

As a prerequisite for achieving a good performance concerning the measures above, JOIN and LEAVE operations have to be implemented so that the overlay network has good topological properties. Among the most important are:

- **Degree**: Ideally, the degree should be kept small to avoid a high update cost if node enter or leave the system.

- **Diameter**: The diameter should be small to enable fast searching and broadcasting.

- **Node Expansion**: The node expansion of a graph $G = (V, E)$ is defined as

$$\beta = \min_{U \subseteq V:\, |U| \leq |V|/2} \frac{|N(U)|}{|U|}$$

  where $N(U)$ is the set of neighbors of U. To ensure a high fault tolerance, the node expansion should be as large as possible.

10.2 A simple example

We start with the construction of a simple overlay network implementing a name service. In the context of a supervised overlay network, this requires the following operations:

- $s.\text{JOIN}(p, \text{ID})$: supervisor s is contacted by peer p in order to join the system under the name ID.

- $s.\text{LEAVE}(p)$: supervisor s is contacted by peer p in order to leave the system.

- $p.\text{LOOKUP}(\text{ID})$: peer p searches for the peer with name ID.

To realize these operations, we simply organize the peers in an ordered, doubly linked list, where the supervisor serves as a handle into the list and is assumed to have the smallest possible name. We assume that each peer v in this list stores its predecessor in $\text{pred}(v)$ and its successor in $\text{succ}(v)$. For an implementation of the operations to realize this list, see Figure 1. Notice that implementing such a linked list is more complicated than in a single-machine environment, even when using a supervisor, because nodes may send simultaneously requests to the supervisor to join or leave the system, which has to be handled with care in order to avoid inconsistencies. This is complicated by the fact that in a distributed environment it is possible that messages may have a high delay or may change the order in which they arrive. To keep the presentation simple, we ignore these complications here and only present the code for isolated JOIN and LEAVE requests.

Obviously, this overlay network has the following performance:

```
JOIN(u, ID):   // u wants to join the system
 if not supervisor then
     call s.JOIN(u, ID)   // forward to supervisor
 else
     call v.INTEGRATE(u, ID)


INTEGRATE(u, ID):
 if ID < ID(succ(v)) then
     call u.NEWPRED(v)
     call u.NEWSUCC(succ(v))
     call succ(v).NEWPRED(u)
     succ(v) = u
 else
     call succ(v).INTEGRATE(u, ID)


LEAVE(u):   // u wants to leave the system
 if not supervisor then
     call s.LEAVE(u)   // forward to supervisor
 else
     call v.REMOVE(u)

REMOVE(u):
 call pred(u).NEWSUCC(succ(u))
 call succ(u).NEWPRED(pred(u))

NEWPRED(u):
 pred(v) = u

NEWSUCC(u):
 succ(v) = u

SEARCH(u, ID):   // u searches for ID
 // always starts with u calling s.SEARCH(u, ID)
 if ID(v) > ID then
     call u.REPLY(ID, ∅)
 else
     if ID(v) < ID then
         call succ(v).SEARCH(u, ID)
     else
         call u.Reply(ID, v)

REPLY(ID, v):
// handles the search result
```

Figure 1: Operations needed to maintain an ordered list of peers. v is the node currently executing the operation, and s denotes the supervisor.

Theorem 10.1 *Given n peers,* JOIN *and* SEARCH *have a dilation and work of $O(n)$, and* LEAVE *has a dilation and work of $O(1)$. The degree of the network is 2, the diameter is n, and the expansion is $2/n$.*

Certainly, this is not the best possible overlay network for distributed name service. We will get to know networks that can perform such a service in $O(\log n)$ dilation and work for each operation.

Next we will present two supervised overlay networks with a much better performance than the network above. One is suitable for broadcasting, and the other is suitable for distributed data management or grid computing.

10.3 A supervised tree network

In this section we present a supervised tree network that can be used for efficient broadcasting (see also [2]). That is, we want to implement the following operations:

- s.JOIN(p): supervisor s is contacted by peer p in order to join the system.
- s.LEAVE(p): supervisor s is contacted by peer p in order to leave the system.
- p.BROADCAST(m): peer p wants to broadcast m to all other peers

To achieve this, the supervisor maintains $2 \log n$ root slots, which are divided into $\log n$ pairs labeled from 0 to $\log n - 1$. Each slot can store information about a peer. The peers stored in the slots of pair i are denoted by $s_i(1)$ and $s_i(2)$. The supervisor tries to maintain the following invariant at any time:

Invariant 10.2 *Any peer stored in a slot of pair i is the root of a complete binary tree of depth i, and at any time, at most one slot pair is fully occupied, and below this pair there is no occupied slot.*

In order to maintain this property, we implement JOIN and LEAVE as shown in Figure 2. Again, notice that these operations may only work if JOIN and LEAVE requests do not overlap.

```
JOIN(u):    // u wants to join the system
 if not supervisor then
     call s.JOIN(u)    // forward to supervisor
     exit
 call u.NEWPARENT(s)
 if there is a fully occupied pair i then
     let s_{i+1}(j) be an empty slot in pair i + 1
     s_{i+1}(j) = u
     call u.NEWLEFTCHILD(s_i(1))
     call u.NEWRIGHTCHILD(s_i(2))
     call s_i(1).NEWPARENT(u)
     call s_i(2).NEWPARENT(u)
     s_i(1) = NULL
     s_i(2) = NULL
 else
     // some slot s_0(j) must be empty
     s_0(j) = u

NEWPARENT(u):
 parent(v) = u
```

```
NEWLEFTCHILD(u):
 left_child(v) = u

NEWRIGHTCHILD(u):
 right_child(v) = u

LEAVE(u):    // u wants to leave the system
 if not supervisor then
     call s.LEAVE(u)    // forward to supervisor
     exit
 let s_i(j) be the lowest filled slot
 if i > 0 then
     s_{i-1}(1) = left_child(s_i(j))
     s_{i-1}(2) = right_child(s_i(j))
     call s_{i-1}(1).NEWPARENT(s)
     call s_{i-1}(2).NEWPARENT(s)
 call s_i(j).NEWPARENT(parent(u))
 call s_i(j).NEWLEFTCHILD(left_child(u))
 call s_i(j).NEWRIGHTCHILD(right_child(u))
 s_i(j) = NULL
```

Figure 2: Operations needed to maintain Invariant 10.2. v is the node currently executing the operation, and s denotes the supervisor.

Lemma 10.3 *The* JOIN *and* LEAVE *operations preserve Invariant 10.2 at any time.*

Proof. We start with the claim that each node in a slot of pair i is the root of a complete binary tree of depth i. This can be shown by complete induction over the number of JOIN and LEAVE requests executed so far.

Consider the situation that node u wants to join the network. If the supervisor places it in a slot $s_0(j)$, we are done. So assume that the supervisor places it in a slot $s_{i+1}(j)$ with $i > 0$. In this case, u obtains as its children the nodes in $s_i(1)$ and $s_i(2)$. According to the invariant, these must be the root of a complete binary tree of depth i. Hence, u will become the root of a complete binary tree of depth $i + 1$.

Next, consider the situation that a node u wants to leave the network. Then the supervisor fills its position with the node in the lowest filled slot $s_i(j)$. If $i = 0$, we are done. So assume that $i > 0$. In this case, the supervisor takes the children of $s_i(j)$ and places them in the slot pair $i - 1$ (which is possible according to the invariant). Since, also according to the invariant, these children must be roots of a complete binary tree of depth $i - 1$, the proof about complete binary trees is complete.

Hence, it remains to show that there will only be one fully occupied slot pair, and below this pair there is no occupied slot. This can also be directly shown by complete induction. □

Hence, Invariant 10.2 is maintained, and therefore given n peers, the largest slot pair that can be occupied is pair $\lfloor \log n \rfloor$, because a complete binary tree of depth $\lfloor \log n \rfloor$ contains $2^{\lfloor \log n \rfloor + 1} - 1 \geq n$ nodes. Since the diameter of a complete binary tree of depth d is $2d$, we obtain the following result.

Theorem 10.4 *Given n peers,* JOIN *and* LEAVE *have a dilation and work of $O(1)$. Furthermore, the degree of the tree network is 3 (apart from the supervisor), the diameter is $O(\log n)$, and the expansion is $O(1/n)$.*

Finally, we present a broadcast function in Figure 3 so that information can be disseminated in the network.

```
BROADCAST(m):
 if not supervisor then
     call s.BROADCAST(m)     // forward to supervisor
 else
     for all occupied s_i(j) do
         call s_i(j).BROADCASTDOWN(m)

BROADCASTDOWN(m):
 if left_child ≠ NULL then
     call left_child.BROADCASTDOWN(m)
 if right_child ≠ NULL then
     call right_child.BROADCASTDOWN(m)
 // handle broadcast message
```

Figure 3: Implementation of BROADCAST. v is the node currently executing the operation.

Inspecting the code, we arrive at the following result, which is optimal for broadcasting in constant degree networks.

Theorem 10.5 *The* BROADCAST *operation has a dilation of $O(\log n)$ and requires a work of $O(n)$.*

10.4 A supervised DeBruijn network

Next we present a supervised network based on the DeBruijn graph (see also [1]). We want to use this network to search for data or processes. To achieve this, we need to implement the following operations:

- s.JOIN(p): supervisor s is contacted by peer p in order to join the system.
- s.LEAVE(p): supervisor s is contacted by peer p in order to leave the system.
- p.SEARCH(key): peer p searches for the peer responsible for key

We start with a description of how to implement JOIN and LEAVE. Afterwards, we describe how to dynamically place data in the DeBruijn network, and then we describe how to search for the data.

Joining and leaving the network

Recall the definition of the static DeBruijn graph:

Definition 10.6 (DeBruijn) *The* d-dimensional DeBruijn graph $DB(d)$ *is an undirected graph* $G = (V, E)$ *with node set* $V = [2]^d$ *and edge set* E *that contains all edges* $\{v, w\}$ *with the property that* $v = (v_{d-1}, \ldots, v_0)$ *and* $w \in \{(v_{d-2}, \ldots, v_0 x) : x \in \{0, 1\}\}$.

For the dynamic DeBruijn graph we want to keep the following invariant at any time:

Invariant 10.7 *At any point in time, the nodes are numbered in a consecutive way from 0 to* $n - 1$. *Every node* $x = (x_{d-1} \ldots x_0)_2$ *in the system (where* $(x_{d-1} \ldots x_0)_2$ *is the most compact binary representation of* x*, i.e.* $x_{d-1} = 1$*) is connected to the nodes*

(a) $x - 1$ *and* $x + 1$ *(if they exist),*

(b) $(x_{d-2} \ldots x_0 z)$ *with* $z \in \{0, 1\}$*, and*

(c) $(x_{d-1} \ldots x_0 z)$ *with* $z \in \{0, 1\}$ *(if they exist).*

Notice that the nodes $(x_{d-2} \ldots x_0 z)$ with $z \in \{0, 1\}$ always exist: Consider any x whose highest 0-bit is at position j. Then $(x_{d-2} \ldots x_0 z)$ must have this bit at position $j + 1$, which means that it must be less than x. If x does not have a 0-bit, then it is also easy to see that $(x_{d-2} \ldots x_0 z)$ is at most x.

The invariant gives the following result:

Lemma 10.8 *If Invariant 10.7 is true, then the average degree is constant, the worst-case degree is logarithmic, and for every* $k \leq \lfloor \log n \rfloor$*, the nodes from 0 to* $2^k - 1$ *contain* $DB(k)$.

Proof. First of all, when looking at the connection requirements as directed edges, then the connection rule in Invariant 10.7 specifies at most 6 outgoing edges for each node x in the system. Hence, the total number of edges in the system can be at most $6n$, and therefore, the average degree must be constant. Furthermore, any node $x = (x_{k-1} \ldots x_0)$ can only have incoming edges from nodes with numbers $x - 1$, $x + 1$, $(x_{k-2} \ldots x_1)$, and all nodes of the form

$$(y_{d-1} \ldots y_{i+1} x_i \ldots x_1) \quad \text{with } \sum_{j=i+1}^{d-1} y_j = 1,$$

where $d = \lceil \log n \rceil$. Hence, the indegree of any node is at most $d + 3$, which completes the bound on the degree.

Finally, the fact that the dynamic DeBruijn graphs contains static DeBruijn graphs immediately follows from Invariant 10.7(b). $\square$

In order to maintain the invariant, the supervisor only needs to store the following information:

- the 0-node,
- the node $x = (x_{d-1} \dots x_0)$ with currently largest number,
- the node $(0x_{d-1} \dots x_1) = \lfloor x/2 \rfloor$, and
- the node $(x_{d-2} \dots x_1) = 2x - 2^{\lfloor \log x \rfloor} + 1$.

With this information, JOIN and LEAVE operations can be handled as follows:

The JOIN operation. For every new node u that wants to join the system, the supervisor gives it the number $x+1$ and connects u to the nodes with numbers x, $\lfloor \frac{x+1}{2} \rfloor$, $2(x+1) - 2^{\lfloor \log(x+1) \rfloor}$, and $2(x+1) - 2^{\lfloor \log(x+1) \rfloor} + 1$. Afterwards, it updates its information as described above. This achieves the following result:

Theorem 10.9 JOIN *can be executed with constant work.*

Proof. First, we consider the nodes $x+1$ has to be connected with. The supervisor already stores x, so this is easy. Furthermore, $\lfloor \frac{x+1}{2} \rfloor$ is either a direct neighbor of $\lfloor x/2 \rfloor$ or the node itself. Finally, if $\lfloor \log x \rfloor = \lfloor \log(x+1) \rfloor$, then $2(x+1) - 2^{\lfloor \log(x+1) \rfloor}$ is a direct neighbor of $2x - 2^{\lfloor \log x \rfloor} + 1$ and certainly $2(x+1) - 2^{\lfloor \log(x+1) \rfloor} + 1$ is a direct neighbor of $2(x+1) - 2^{\lfloor \log(x+1) \rfloor}$. Otherwise, $2(x+1) - 2^{\lfloor \log(x+1) \rfloor} = 0$ and $2(x+1) - 2^{\lfloor \log(x+1) \rfloor} + 1 = 1$, which can be accessed via the 0-node. Thus, the work for connecting $x+1$ is constant.

Since the information that has to be updated by the supervisor is a subset of the nodes $x+1$ has to be connected with, also updating the information takes constant work. □

The LEAVE operation For every node u that wants to leave the system, the supervisor reverses the last JOIN operation, which frees up node x, and then lets x take over the role of u. Using Lemma 10.8, we get:

Theorem 10.10 LEAVE *can be executed with at most logarithmic work.*

Dynamic data management

Next, we show how to perform dynamic data management on the supervised DeBruijn network. Suppose that we have a hash function h mapping data items to random real values in $[0, 1)$. Then it follows from Section 8 that if we manage to distribute the interval $[0, 1)$ evenly among the nodes, then we also manage to distribute the data evenly among the nodes. In order to achieve an even distribution of $[0, 1)$, we use a strategy similar to the cut-and-paste strategy:

Initially, the entire $[0, 1)$ interval is stored at node 0. Each time a new node $x = (x_{d-1} \dots x_0)$ is added to the system, it takes the upper half of the ranges from the nodes $(x_{d-2} \dots x_0 0)$ and $(x_{d-2} \dots x_0 1)$.

This simple strategy can achieve the following remarkable result, which implies that the storage load of the nodes deviates by at most a factor of 2.

Theorem 10.11 *At any point in which the system has n nodes, every node in the system stores at least one and at most two subintervals of $[0, 1)$ of size $1/2^{\lceil \log n \rceil}$.*

Proof. We first show by complete induction that for every $k \geq 0$ it holds that for 2^k nodes, every node has exactly one interval of $[0, 1)$ of size $1/2^k$. This certainly true for $k = 0$ and $k = 1$. So suppose that the hypothesis is already known to be true up to some k. Then consider the following lemma:

Lemma 10.12 *For every node $x \in \{0, \ldots, 2^k - 1\}$ there is exactly one node $y \in \{2^k, \ldots, 2^{k+1} - 1\}$ stealing an interval from x. Every node $x \in \{2^k, \ldots, 2^{k+1} - 2\}$ steals an interval from exactly two nodes in $\{0, \ldots, 2^k - 1\}$, and exactly one node in $\{2^k, \ldots, 2^k - 1\}$ steals an interval from x. Node $2^k - 1$ only steals an interval from node $2^k - 2$.*

Proof. Follows immediately from the description of the range movement strategy. □

Hence, when going from 2^k to 2^{k+1} nodes, each node in $\{0, \ldots, 2^k - 1\}$ can give half of its interval away so that it stays with an interval of size $1/2^{k+1}$. Furthermore, each node in $\{2^k, \ldots, 2^{k+1} - 2\}$ gets two intervals of size $1/2^{k+1}$ and loses one of them to another node. Finally, node $2^{k+1} - 1$ obtains exactly one interval of size $1/2^{k+1}$. Thus, when reaching 2^{k+1} nodes, all nodes have exactly one interval of size $1/2^{k+1}$, which completes the induction.

Looking at the proof of the induction, we observe that the theorem must be true. □

Theorem 10.11 implies that every node only needs a constant amount of space to store the part of the $[0, 1)$ interval it is responsible for.

Searching for data

Finally, we describe how to search for data. First, consider the situation that there are exactly 2^k nodes in the system for some $k \in \mathbb{N}$. When using the rule above of always stealing the range with largest offset from a node, the following claim holds.

Theorem 10.13 *In any situation in which there are exactly 2^k nodes in the system for some $k \in \mathbb{N}$, every node $v = (v_1 v_2 \ldots v_k) > 0$ is responsible for the range $R_v = [r_v, r_v + 1/2^k)$, where*

$$r_v = -\sum_{i \geq 1} \frac{(-1)^{v_i}}{2^i} ,$$

and node 0 is responsible for the range $[0, 1/2^k)$.

Proof. For $k = 1$ this is obviously true. The remaining proof will be given later... □

With the help of this claim, we can now determine where a data item d is currently located:

Suppose that we have exactly 2^k nodes for some $k \in \mathbb{N}$. If $h(d) < 1/2^k$, we know that node 0 is responsible for d. Otherwise, we start with the number $y_1 = 1/2$ and we work in iterations to recover the node $(v_1 v_2 \ldots)$ responsible for d. In the ith iteration we check if $y_i > h(d)$. If so, we set $v_i = 0$ and $y_{i+1} = y_i - 1/2^{i+1}$, and otherwise we set $v_i = 1$ and $y_{i+1} = y_i + 1/2^{i+1}$. This is continued until we reach iteration k.

Now, suppose that we may have any number n of nodes, and this number may not be known to the nodes in the network. Then first observe that the nodes can use the size of their intervals to get an estimate on the current number of nodes in the system:

- If a node $v \in \mathbb{N}_0$ currently has two intervals of size $1/2^k$, then it knows that the number of nodes in the system must be between v and $2^k + v/2$.
- If a node $v \in \mathbb{N}_0$ currently has one interval of size $1/2^k$, then it knows that the number of nodes in the system must be between $2^k + v/2$ and $2^{k+1} + v/2$.

So v can take the value $n' = 2^{k-1}$ (in the first case) or $n' = 2^k$ (in the second case) as a conservative estimate and first compute the location w of data item d for a system with n' nodes. In a first stage, v sends a search message to that node w. If w is not responsible for d any more, then the message is forwarded in a second stage along shortcut pointers d's range was moved until the request arrives at the node currently storing d. Since both stages only need $O(\log n)$ hops, we arrive at the following result.

Theorem 10.14 *Any* SEARCH *request can be executed with logarithmic work.*

References

[1] C. Riley and C. Scheideler. A distributed hash table for computational grids. Technical report, Johns Hopkins University, October 2003.

[2] C. Riley and C. Scheideler. Guaranteed broadcasting using SPON: A supervised peer overlay network. In *3rd International Zürich Seminar on Communications*, 2004.