8 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*. A prerequisite for an overlay network to be useful is that it has good topological properties. Among the most important are:

- *Degree*: Ideally, the degree should be kept small to avoid a high update cost if a node enters or leaves the system.
- *Diameter*: The diameter should be small to allow the fast exchange of information between any pair of nodes in the network.
- *Node expansion*: The node expansion of a graph $G = (V, E)$ is defined as

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

  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.

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. These networks were first investigated in [2, 1]. In a supervised overlay network, the topology is under the control of a special machine (or node) called the *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 operations, however, may be executed without involving the supervisor. In order for a supervised network to be highly scalable, two central requirements have to be fulfilled:

1. The supervisor needs to store at most a polylogarithmic amount of information about the network at any time (i.e. if there are n nodes in the network, storing contact information about $O(\log^2 n)$ of these nodes would be fine, for example), and
2. it takes at most a constant number of communication rounds to include a new node into or exclude an old node from the network.

A *communication round* is over once all the packets that existed at the beginning of the communication round have been delivered. The packets generated by these packets will participate in the next communication round.

We show in the following how these requirements can be achieved, using a general approach called the recursive approach. To simplify the presentation, we assume that all departures are *graceful*, i.e. every node leaving the system informs the supervisor about this and may provide some additional information simplifying the task of the supervisor to repair the network.

8.1 The recursive approach

In the resursive approach, the supervisor assigns a *label* to every node that wants to join the system. The labels are represented as binary strings and are generated in the following order:

$$0, 1, 01, 11, 001, 011, 101, 111, 0001, 0011, 0101, 0111, 1001, 1011, \ldots$$

Basically, when stripping off the least significant bit, then the supervisor is first creating all binary numbers of length 0, then length 1, then length 2, and so on. More formally, consider the mapping $\ell : \mathbb{N}_0 \to \{0,1\}^*$ with the property that for every $x \in \mathbb{N}_0$ with binary representation $(x_d \ldots x_0)_2$ (where d is minimum possible),

$$\ell(x) = (x_{d-1} \ldots x_0 x_d) \ .$$

Then ℓ generates the sequence of labels displayed above. In the following, it will also be helpful to view labels as real numbers in $[0,1)$. Let the function $r : \{0,1\}^* \to [0,1)$ be defined so that for every label $\ell = (\ell_1 \ell_2 \ldots \ell_d) \in \{0,1\}^*$,

$$r(\ell) = \sum_{i=1}^{d} \frac{\ell_i}{2^i} \ .$$

Then the sequence of labels above translates into

$$0,\ 1/2,\ 1/4,\ 3/4,\ 1/8,\ 3/8,\ 5/8,\ 7/8,\ 1/16,\ 3/16,\ 5/16,\ 7/16,\ 9/16,\ \ldots$$

Thus, the more labels are used, the more densely the $[0,1)$ interval will be populated. Furthermore, we will use the function $b : [0,1) \to \{0,1\}^*$ that translates a real number back into a label.

When using the recursive approach, the supervisor aims to maintain the following invariant at every step:

Invariant 8.1 *The set of labels used by the nodes is* $\{\ell(0), \ell(1), \ldots, \ell(n-1)\}$*, where* n *is the current number of nodes in the system.*

This invariant is preserved when using the following simple strategy:

- Whenever a new node v joins the system and the current number of nodes is n, the supervisor assigns the label $\ell(n)$ to v and increases n by 1.
- Whenever a node w with label ℓ wants to leave the system, the supervisor asks the node with currently highest label $\ell(n-1)$ to change its label to ℓ and reduces n by 1.

How does this strategy help us with maintaining dynamic overlay networks? We will see how this works in the following subsections. To keep things simple, we start with a cycle.

8.2 Recursively maintaining a cycle

We start with some notation. In the following, the label assigned to some node v will be denoted as ℓ_v. Given n nodes with unique labels, we define the *predecessor* $\mathrm{pred}(v)$ of node v as the node w for which $r(\ell_w)$ is closest from below to $r(\ell_v)$, and we define the *successor* $\mathrm{succ}(v)$ of node v as the node w for which $r(\ell_w)$ is closest from above to node $r(\ell_v)$ (viewing $[0,1)$ as a ring in both cases). Given two nodes v and w, we define their *distance* as

$$\delta(v,w) = \min\{(1 + r(\ell_v) - r(\ell_w)) \text{ mod } 1,\ (1 + r(\ell_w) - r(\ell_v)) \text{ mod } 1\}\ .$$

In order to maintain a cycle among the nodes, we simply have to maintain the following invariant:

Invariant 8.2 *Every node v in the system is connected to* $\mathrm{pred}(v)$ *and* $\mathrm{succ}(v)$.

Now, suppose that the labels of the nodes are generated via the recursive strategy above. Then we have the following properties:

Lemma 8.3 *Let n be the current number of nodes in the system, and let $\bar{n} = 2^{\lfloor \log n \rfloor}$. Then for every node $v \in V$:*

- $|\ell_v| \leq \lceil \log n \rceil$ *and*
- $\delta(v, \mathrm{pred}(v)) \in [1/(2\bar{n}), 1/\bar{n}]$ *and* $\delta(v, \mathrm{succ}(v)) \in [1/(2\bar{n}), 1/\bar{n}]$.

So the nodes are approximately evenly distributed in $[0,1)$ and the number of bits for storing a label is almost as low as it can be without violating the uniqueness requirement. But how does the supervisor maintain the cycle? This is implied by the following demand, where n is again the current number of nodes in the system.

Invariant 8.4 *At any time, the supervisor stores the contact information of* $\mathrm{pred}(v)$*,* v*,* $\mathrm{succ}(v)$*, and* $\mathrm{succ}(\mathrm{succ}(v))$ *where v is the node with label $\ell(n-1)$.*

In order to satisfy Invariants 8.2 and 8.4, the supervisor performs the following actions.
If a new node w joins, then the supervisor

- informs w that $\ell(n)$ is its label, $\mathrm{succ}(v)$ is its predecessor, and $\mathrm{succ}(\mathrm{succ}(v))$ is its successor,
- informs $\mathrm{succ}(v)$ that w is its new successor,
- informs $\mathrm{succ}(\mathrm{succ}(v))$ that w is its new predecessor,
- asks $\mathrm{succ}(\mathrm{succ}(v))$ to send its successor information to the supervisor, and
- sets $n = n + 1$.

If an old node w leaves and reports ℓ_w, $\mathrm{pred}(w)$, and $\mathrm{succ}(w)$ to the supervisor (recall that we are assuming graceful departures), then the supervisor

- informs v (the node with label $\ell(n-1)$) that ℓ_w is its new label, $\mathrm{pred}(w)$ is its new predecessor, and $\mathrm{succ}(w)$ is its new successor,

- informs $\mathrm{pred}(w)$ that its new successor is v,
- informs $\mathrm{succ}(w)$ that its new predecessor is v,
- informs $\mathrm{pred}(v)$ that $\mathrm{succ}(v)$ is its new successor,
- informs $\mathrm{succ}(v)$ that $\mathrm{pred}(v)$ is its new predecessor,
- asks $\mathrm{pred}(v)$ to send its predecessor information to the supervisor and to ask $\mathrm{pred}(\mathrm{pred}(v))$ to send its predecessor information to the supervisor, and
- sets $n = n - 1$.

A detailed implementation of the leave and join operations can be found in Figures 1 and 2. The following lemma is not difficult to check and will be an assignment.

```
Supervisor {

 Supervisor() {
     $n := 0$    # counter
     $v :=$ NULL    # node with label $\ell(n-1)$
     $pv :=$ NULL    # $\mathrm{pred}(v)$
     $sv :=$ NULL    # $\mathrm{succ}(v)$
     $ssv :=$ NULL    # $\mathrm{succ}(\mathrm{succ}(v))$
 }

 Join($w$: Reference) {
     if ($n = 0$) {
         $w \leftarrow$ setup($0, w, w$)
         $pv := w$
         $v := w$
         $sv := w$
         $ssv := w$
     } else {
         $w \leftarrow$ setup($\ell(n), sv, ssv$)
         $sv \leftarrow$ setSucc($w$)
         $ssv \leftarrow$ setPred($w$)
         $pv := sv$
         $v := w$
         $sv := ssv$
         $ssv := ssv \leftarrow$ getSucc()
     }
     $n := n + 1$
 }

 Leave($\ell$: Int, $pw$: Reference, $sw$: Reference) {
     if ($n > 0$) {
         if ($n = 1$) {
             $pv :=$ NULL, $v :=$ NULL
             $sv :=$ NULL, $ssv :=$ NULL
         } else {
             # remove $v$ from the system
             $pv \leftarrow$ setSucc($sv$)
             $sv \leftarrow$ setPred($pv$)
             if ($pw = v$) $pw := pv$
             if ($sw = v$) $sw := sv$
             # move $v$ into position of $w$
             if ($v \neq w$) {
                 $v \leftarrow$ setup($\ell, pw, sw$)
                 $pw \leftarrow$ setSucc($v$)
                 $sw \leftarrow$ setPred($v$)
             }
             # update pointers
             if ($pv = w$) $pv := v$
             if ($sv = w$) $sv := v$
             $ssv := sv$
             $sv := pv$
             $v := pv \leftarrow$ getPred()
             $pv := pv \leftarrow$ getPredPred()
         }
         $n := n - 1$
     }
 }
}
```

Figure 1: Operations needed by the supervisor to maintain a cycle.

```
Peer {

 Peer() {
     $label := 0$    # label of peer $v$
     $succ :=$ NULL    # succ($v$)
     $pred :=$ NULL    # pred($v$)
     $sc :=$ newContact()    # contact point of $v$
     Register($sc$)    # requests to $sc$ forwarded to $v$
 }

 Join($s$: Reference) {
     if ($s \neq$ NULL) {
         $s \leftarrow$ Join(Ref($sc$))
         $super := s$    # current supervisor
     }
 }

 Leave() {
     if ($super \neq$ NULL)
         $super \leftarrow$ Leave($label, pred, succ$)
 }

setup($\ell$ : Int, $p$ : Reference, $s$ : Reference) {
     $label := \ell$
     $pred := p$
     $succ := s$
}

setSucc($w$: Reference) {
     $succ := w$
}

setPred($w$: Reference) {
     $pred := w$
}

getSucc(): Reference {
     return $succ$
}

getPred(): Reference {
     return $pred$
}

getPredPred(): Reference {
     return $pred \leftarrow$ getPred()
}
```

Figure 2: Operations needed by a peer to maintain a cycle.

Lemma 8.5 *The join and leave operations preserve Invariants 8.2 and 8.4.*

Hence, we arrive at the following theorem, which implies that our central requirements on a supervisor are kept.

Theorem 8.6 *At any time, the supervisor only needs to store the current value of n and a constant amount of contact information, and the join and leave operations only need a constant amount of messages and three communication rounds to complete.*

However, one important issue that needs to be pointed out is that the supervisor has to handle leave operations in a strictly sequential manner. Otherwise, it may happen that two neighbors on the cycle leave at the same time, which can lead to a violation of Invariant 8.2 when reconnecting the nodes according to the leave operation above. It will be an assignment to think about ways avoiding a strictly sequential execution of leave requests since this can make the supervisor algorithm less scalable.

8.3 Recursively maintaining a tree

The cycle has a low degree but its diameter and expansion are very bad. The simplest way of achieving a low diameter is to use a tree. Thus, next we discuss how to recursively maintain a tree. As for the cycle, our basic approach will be to preserve Invariant 8.1. We will also preserve Inviarant 8.2, though the edges implied by this Invariant will not be part of the tree. But they will tremendously simplify the task of maintaining a tree, as we will see. Altogether, the following connectivity information has to be preserved.

Invariant 8.7 *Every node v in the system with label $\ell_v = (\ell_1 \ldots \ell_d)$ is connected to*

1. $\mathrm{pred}(v)$ *and* $\mathrm{succ}(v)$ *(to form a cycle) and*
2. *the nodes with labels $(\ell_1 \ldots \ell_{d-2}1)$, $(\ell_1 \ldots \ell_{d-1}01)$, and $(\ell_1 \ldots \ell_{d-1}11)$, if they exist (to form a tree).*

Suppose that this invariant is kept at any time. Then the following lemma follows.

Lemma 8.8 *At any time, the n nodes (apart from node 0) form a binary tree of depth $\lceil \log n \rceil - 1$.*

Proof. Consider a binary tree with n nodes, and label the edge to the left child of any node “0” and to the right child of any node “1”. Let the label t_v of every node v in this tree be the sequence of edge labels when moving along the unique path from the root to v. Then every node v with label $(\ell_1 \ldots \ell_d)$ is connected to the node with label $(\ell_1 \ldots \ell_{d-1})$ (its parent), if it exists, and is also connected to the nodes with labels $(\ell_1 \ldots \ell_d 0)$ and $(\ell_1 \ldots \ell_d 1)$ (its children), if they exist. Defining t_v as ℓ_v (the label of v in our network) without the least significant bit, we see that Invariant 8.7(2) fulfills the connectivity requirements of a tree. Since it follows from Lemma 8.3 that every node has a label of size at most $\lceil \log n \rceil$, the depth of the tree can be at most $\lceil \log n \rceil - 1$ (when ignoring node 0). □

Next we specify the connectivity information the supervisor needs in order to maintain the tree.

Invariant 8.9 *At any time, the supervisor stores the contact information of* $\mathrm{pred}(v)$, v, $\mathrm{succ}(v)$, *and* $\mathrm{succ}(\mathrm{succ}(v))$ *where v is the node with label $\ell(n)$.*

Hence, the supervisor does not need any further connectivity information beyond what it needs for the cycle. In order to satisfy Invariants 8.7 and 8.9, the supervisor performs the following actions. If a new node w joins, then the supervisor

- informs w that $\ell(n+1)$ is its label, $\mathrm{succ}(v)$ is its predecessor, and $\mathrm{succ}(\mathrm{succ}(v))$ is its successor, and $\mathrm{succ}(v)$ resp. $\mathrm{succ}(\mathrm{succ}(v))$ is its parent (depending on $\ell(n+1)$),
- informs $\mathrm{succ}(v)$ that w is its new successor,
- informs $\mathrm{succ}(\mathrm{succ}(v))$ that w is its new predecessor,
- asks $\mathrm{succ}(\mathrm{succ}(v))$ to send its successor information to the supervisor, and
- sets $n = n + 1$.

Hence, from the point of view of the supervisor, the inclusion of a new node is almost identical to the cycle.

If an old node w leaves and reports ℓ_w, $\mathrm{pred}(w)$, $\mathrm{succ}(w)$, parent(w), lchild(w), and rchild(w) to the supervisor, then the supervisor again executes almost the same steps as for the cycle.

When using the code for the supervisor given in Figure 3 and the code for the peers given in Figure 4, it is not difficult to prove the following lemma.

```
Supervisor {

 Supervisor() {
    $n := 0$    # counter
    $v :=$ NULL    # node with label $\ell(n)$
    $pv :=$ NULL    # pred($v$)
    $sv :=$ NULL    # succ($v$)
    $ssv :=$ NULL    # succ(succ($v$))
 }

 Join($w$: Reference) {
    if ($n = 0$) {
       $w \leftarrow$ setup(0, $w$, $w$, NULL, NULL, NULL)
       $pv := w$
       $v := w$
       $sv := w$
       $ssv := w$
    } else {
       if ($\ell(n)\&2 = 0$) {
          $w \leftarrow$ setup($\ell(n)$, $sv$, $ssv$, $ssv$, NULL, NULL)
          $ssv \leftarrow$ setRightChild($w$)
       } else {
          $w \leftarrow$ setup($\ell(n)$, $sv$, $ssv$, $sv$, NULL, NULL)
          $sv \leftarrow$ setLeftChild($w$)
       }
       $sv \leftarrow$ setSucc($w$)
       $ssv \leftarrow$ setPred($w$)
       $pv := sv$
       $v := w$
       $sv := ssv$
       $ssv := ssv \leftarrow$ getSucc()
    }
    $n := n + 1$
 }

 Leave($\ell$: Int, $pw$: Reference, $sw$: Reference,
          $fw, lcw, rcw$: Reference) {
    if ($n > 0$) {
       if ($n = 1$) {
          $pv :=$ NULL, $v :=$ NULL
          $sv :=$ NULL, $ssv :=$ NULL
       } else {
          # remove $v$ from tree
          if ($\ell(n-1)\&2 = 0$) $sv \leftarrow$ setRightChild(NULL)
                          else $pv \leftarrow$ setLeftChild(NULL)
          $pv \leftarrow$ setSucc($sv$)
          $sv \leftarrow$ setPred($pv$)
          if ($pw = v$) $pw := pv$
          if ($sw = v$) $sw := sv$
          if ($lcw = v$) $lcw :=$ NULL
          if ($rcw = v$) $rcw :=$ NULL
          # move $v$ into position of $w$
          if ($v \neq w$) {
             $v \leftarrow$ setup($\ell, pw, sw, fw, lcw, rcw$)
             $pw \leftarrow$ setSucc($v$)
             $sw \leftarrow$ setPred($v$)
             if ($\ell\&2 = 0$)
                $fw \leftarrow$ setRightChild($v$)
             else
                $fw \leftarrow$ setLeftChild($v$)
             if ($lcw \neq$ NULL) $lcw \leftarrow$ setParent($v$)
             if ($rcw \neq$ NULL) $rcw \leftarrow$ setParent($v$)
          }
          # update pointers
          if ($pv = w$) $pv := v$
          if ($sv = w$) $sv := v$
          $ssv := sv$
          $sv := pv$
          $v := pv \leftarrow$ getPred()
          $pv := pv \leftarrow$ getPredPred()
       }
       $n := n - 1$
    }
 }
}
```

Figure 3: Operations needed by the supervisor to maintain a tree.

Lemma 8.10 *The join and leave operations preserve Invariants 8.7 and 8.9.*

Hence, we arrive at the following theorem.

Theorem 8.11 *At any time, the supervisor only needs to store the current value of n and a constant amount of contact information, and the join and leave operations only need a constant amount of messages and three communication rounds to complete.*

Broadcasting

The dynamic tree can be used for efficient broadcasting. Suppose that some node v wants to broadcast information to all other nodes in the system. One way of solving this is that it forwards the broadcast message directly to the supervisor (so that the supervisor can authorize the broadcast, for example) and the supervisor initiates sending the broadcast message down the tree. A prerequisite for this is that

```
Peer {
 Peer() {
     $label := 0$     # label of peer $v$
     $succ :=$ NULL     # succ($v$)
     $pred :=$ NULL     # pred($v$)
     $parent :=$ NULL
     $lchild :=$ NULL
     $rchild :=$ NULL
     $sc :=$ newContact()     # contact point of $v$
     Register($sc$)     # requests to $sc$ forwarded to $v$
 }

 Join($s$: Reference) {
     if ($s \neq$ NULL) {
         $s \rightarrow$ Join(Ref($sc$))
         $super := s$     # current supervisor
     }
 }

 Leave() {
     if ($super \neq$ NULL)
         $super \leftarrow$ Leave($label, pred, succ, parent, lchild, rchild$)
 }

 setup($\ell$ : Int, $p$ : Reference, $s$ : Reference, $f$: Reference,
             $lc$: Reference, $rc$: Reference) {
     $label := \ell$
     $pred := p$
     $succ := s$
     $parent := f$
     $lchild := lc$
     $rchild := rc$
 }

 setSucc($w$: Reference) {
     $succ := w$
 }

 setPred($w$: Reference) {
     $pred := w$
 }

 setParent($w$: Reference) {
     $parent := w$
 }

 setLeftChild($w$: Reference) {
     $lchild := w$
 }

 setRightChild($w$: Reference) {
     $rchild := w$
 }

 getSucc(): Reference {
     return $succ$
 }

 getPred(): Reference {
     return $pred$
 }

 getPredPred(): Reference {
     return $pred \leftarrow$ getPred()
 }
```

Figure 4: Operations needed by a peer to maintain a tree.

the supervisor remembers the node with label 0, called $root$ by it. If this is the case, then the code in Figure 5 will be executed correctly.

Inspecting the code, we arrive at the following result, which is optimal for broadcasting in constant degree networks. Here, the *dilation* means the longest path taken by a message in the broadcast operation.

Theorem 8.12 *The broadcast operation has a dilation of $O(\log n)$ and requires a work of $O(n)$.*

8.4 Recursively maintaining a deBruijn graph

The tree may be sufficient for broadcasting (if the rate of peers entering and leaving the system is not too high and departures are graceful), but it is insufficient for applications such as large-scale distributed computing (also known as grid computing) or file sharing. To take the file sharing example,

```
# operations of supervisor

 Broadcast($m$ : Message) {
     $root \leftarrow$ sendDown($m$)
 }

# operations of peer

 Broadcast($m$ : Message) {
     if ($super \neq$ NULL) $super \leftarrow$ Broadcast($m$)
 }

 sendDown($m$ : Message) {
     if ($lchild \neq$ NULL) $lchild \leftarrow$ sendDown($m$)
     if ($rchild \neq$ NULL) $rchild \leftarrow$ sendDown($m$)
     # handle broadcast message
 }
```

Figure 5: Implementation of a broadcast operation in the dynamic tree.

imagine we want to come up with a supervised form of the Napster service. That is, instead of having a server that knows every peer and stores an index table of all files offered by the peers for download, we just have a supervisor that takes care of the topology, but all other operations such as storing the index table and insert, delete, and search operations on this index table have to be handled by the peers. To solve this problem in an efficient way, we need a network that allows low-congestion routing, such as the hypercube or the deBruijn graph. Since the deBruijn graph has the advantage that it has a constant degree, we will focus on a solution using the deBruijn graph.

Topology control

We will use a recursive approach similar to the dynamic tree. That is, we will maintain a cycle as our basic structure and will use this cycle to keep the deBruijn edges up-to-date as nodes enter and leave the system. Here, we will use the following interesting geometric representation of a deBruijn graph.

Definition 8.13 *The (binary) deBruijn graph of dimension d, $DB(d)$, is an undirected graph $G = (V, E)$ with node set $V = \{j/2^d \mid j \in \{0, \dots, 2^d - 1\}\}$ and edge set E containing all edges $\{x, y\}$ with y being the closest predecessor in V to $x/2$ or $(1 + x)/2$.*

By “closest predecessor to z”, or $\mathrm{epred}(z)$, we mean here a node with number y so that either y is equal to z or y is the closest number from below to z.

The case $x/2$ represents the right-shift operation with new leftmost bit 0, and the case $y = (1+x)/2$ represents the right-shift operation with new leftmost bit 1. This becomes apparent when looking at the bit strings $b(x)$, $b(x/2)$, and $b((1 + x)/2)$. Suppose that $b(x) = (\ell_1 \dots \ell_d)$. Then $b(x/2) = (0\ell_1 \dots \ell_d)$ and $b((1 + x)/2) = (1\ell_1 \dots \ell_d)$. Hence, removing the least significant bit from the bit strings results in bit strings that are neighbors of $b(x)$ in the deBruijn graph.

Thus, the following connectivity information has to be preserved at every node to form a dynamic deBruijn graph.

Invariant 8.14 *Every node v in the system with label $\ell_v = (\ell_1 \dots \ell_d)$ is connected to*

1. $\mathrm{pred}(v)$ *and* $\mathrm{succ}(v)$ *(to form a cycle),*
2. $\mathrm{epred}(r(\ell_v)/2)$ *and* $\mathrm{epred}((1 + r(\ell_v))/2)$ *(to form the right-shift edges), and*
3. *all nodes w that have a right-shift edge to v (to form the left-shift edges).*

Suppose that this invariant is kept at any time. Then it follows from our previous observations that the following lemma is true.

Lemma 8.15 *At any time, the n nodes adhere to the connectivity requirements in Definition 8.13 and therefore form a dynamic deBruijn graph.*

Next we specify the connectivity information that the supervisor needs to maintain in order to preserve Invariant 8.14.

Invariant 8.16 *Any any time, the supervisor stores the contact information of*

1. $\mathrm{pred}(v)$*,* v*,* $\mathrm{succ}(v)$*, and* $\mathrm{succ}(\mathrm{succ}(v))$ *where v is the node with label $\ell(n-1)$, and*
2. $\mathrm{epred}(r(\ell_v)/2)$ *and* $\mathrm{epred}((1 + r(\ell_v))/2)$*.*

Item (2) is needed to update the right-shift edges, because as long as the right-shift edges are up-to-date, Invariant 8.14(3) will make sure that all edges are up-to-date.

When using now the standard scheme in the recursive approach of inserting a node into or removing a node from the system, then the connections in Invariant 8.16 can always be updated efficiently. The reason for this is that for any n,

1. $\mathrm{epred}(r(\ell(n))/2) \in \{\mathrm{epred}(r(\ell(n-1))/2), \mathrm{succ}(\mathrm{epred}(r(\ell(n-1))/2))\}$,
2. $\mathrm{epred}((1 + r(\ell(n)))/2) \in \{\mathrm{epred}((1 + r(\ell(n-1)))/2), \mathrm{succ}(\mathrm{epred}((1 + r(\ell(n-1)))/2))\}$,
3. $\mathrm{epred}(r(\ell(n-2))/2) \in \{\mathrm{epred}(r(\ell(n-1))/2), \mathrm{pred}(\mathrm{epred}(r(\ell(n-1))/2))\}$, and
4. $\mathrm{epred}((1+r(\ell(n-2)))/2) \in \{\mathrm{epred}((1+r(\ell(n-1)))/2), \mathrm{pred}(\mathrm{epred}((1+r(\ell(n-1)))/2))\}$.

Items (1) and (2) imply that in a join operation it suffices for the supervisor to ask about the successors of $\mathrm{epred}(r(\ell_v)/2)$ and $\mathrm{epred}((1 + r(\ell_v))/2)$, and items (3) and (4) imply that in a leave operation it suffices for the supervisor to ask about the predecessors of $\mathrm{epred}(r(\ell_v)/2)$ and $\mathrm{epred}((1 + r(\ell_v))/2)$, in addition to the actions necessary to maintain the cycle (and replace one node by another in the leave operation). Thus, the following theorem can be shown.

Theorem 8.17 *At any time, the supervisor only needs to store the current value of n and a constant amount of contact information, and the join and leave operations only need a constant amount of messages and three communication rounds to complete.*

Dynamic data management

Now, 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 we use a strategy similar to the consistent hashing strategy:

Every node v is responsible for the interval $I_v = (r(\ell_{\text{pred}(v)}), r(\ell_v)]$, and every data item mapped into I_v is stored at v.

Lemma 8.3 implies that this simple strategy has the following nice property.

Lemma 8.18 *At any point, the expected number of data items stored at a node only deviates by a factor of 2 among the nodes.*

The other nice property of the strategy is that it is easy to maintain by the supervisor:

- If a new node w joins and v is currently the node with label $\ell(n-1)$, then w obtains half of the interval of $\text{succ}(\text{succ}(v))$.
- If an old node w leaves, then v takes over the interval of w and gives its old interval back to $\text{succ}(v)$.

Hence, in our strategy is also similar to the cut-and-paste strategy.

Next, we describe how to search for data. Suppose that we want to find the node v owning data item d and that we are currently at node w. Then move to the node w' with the label $\ell_{w'}$ whose suffix is a maximum prefix of $h(d)$. Or more precisely, if $\ell_{w'} = (\ell_1 \ldots \ell_d)$ and $b(h(d)) = (x_1 x_2 \ldots)$ then w' maximizes the k for which $(\ell_{d-k} \ldots \ell_d) = (x_1 \ldots x_k)$.

This routing strategy can be seen as simulating the bit-adjustment strategy for the perfect deBruijn graph via left-shift operations. Using it, the following result can be shown:

Theorem 8.19 *Any search request can be executed with $O(\log n)$ work.*

The same bound can certainly also be achieved for insert and delete requests.

References

[1] C. Riley and C. Scheideler. A distributed hash table for computational grids. In *18th Int. Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

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