6 Supervised Overlay Networks I

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.
- *Expansion*: The 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 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 investigated, e.g., in [1, 2, 3]. 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 labeling 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.

6.1 The recursive labeling approach

In the recursive labeling 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 $\ell : \mathbb{N}_0 \rightarrow \{0,1\}^*$ 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 $r : \{0,1\}^* \rightarrow [0, 1)$ 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 $b : [0, 1) \rightarrow \{0,1\}^*$ that translates a real number back into a label.

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

Invariant 6.1 *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.

6.2 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* $\text{pred}(v)$ 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* $\text{succ}(v)$ 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

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

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

Invariant 6.2 *Every node v in the system is connected to* $\text{pred}(v)$ *and* $\text{succ}(v)$.

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

Lemma 6.3 *Let n be the current number of nodes in the system, and let $\bar{n} = 2^{\lfloor \log n \rfloor}$. Then for every node $v \in V$:*

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

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

In order to satisfy Invariants 6.2 and 6.4, the supervisor performs the following actions, where v is the node with label $\ell(n-1)$ in the system.

If a new node w joins, then the supervisor

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

If an old node w leaves and reports ℓ_w, $\text{pred}(w)$, and $\text{succ}(w)$ 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. In this implementation, we assume for simplicity that references to relay points can be freely exchanged, i.e., identities are not needed. It will be an assignment to implement the join and leave operations with the identity concept. The following lemma is not difficult to check and will also be an assignment.

Lemma 6.5 *The join and leave operations preserve Invariants 6.2 and 6.4.*

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

Theorem 6.6 *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.*

6.3 Concurrency

The above scheme only allows the supervisor to execute join and leave operations in a strictly sequential manner because it only has sufficient information to integrate or remove one peer at a time. In order to be able to handle d join or leave requests in parallel, we extend Invariant 6.2 with the following rule:

Invariant 6.7 *In addition to Invariant 6.2, every node v in the system is connected to its dth predecessor $\mathrm{pred}_d(v)$ and its dth successor $\mathrm{succ}_d(v)$.*

Furthermore, given that v is the node with label $\ell(n-1)$, Invariant 6.4 needs to be extended to:

Invariant 6.8 *At any time, the supervisor stores the contact information of v, the $2d$ successors of v, and the $3d$ predecessors of v.*

These invariants are preserved in the following way.

```
Supervisor {

 Supervisor() {
    n := 0    # counter
    v := NULL    # node with label ℓ(n − 1)
    pv := NULL    # pred(v)
    sv := NULL    # succ(v)
    ssv := NULL    # succ(succ(v))
 }

 Join(w: Relay) {
    if (n = 0) {
       w ← setup(0, w, w)
       pv := w
       v := w
       sv := w
       ssv := w
    } else {
       w ← setup(ℓ(n), sv, ssv)
       sv ← setSucc(w)
       ssv ← setPred(w)
       pv := sv
       v := w
       sv := ssv
       ssv := ssv ← getSucc()
    }
    n := n + 1
 }

 Leave(ℓ: Int, pw: Relay, sw: Relay) {
    if (n > 0) {
       if (n = 1) {
          pv := NULL, v := NULL
          sv := NULL, ssv := NULL
       } else {
          # remove v from the system
          pv ← setSucc(sv)
          sv ← setPred(pv)
          if (pw = v) pw := pv
          if (sw = v) sw := sv
          # move v into position of w
          if (v ≠ w) {
             v ← setup(ℓ, pw, sw)
             pw ← setSucc(v)
             sw ← setPred(v)
          }
          # update pointers
          if (pv = w) pv := v
          if (sv = w) sv := v
          ssv := sv
          sv := pv
          v := pv ← getPred()
          pv := pv ← getPredPred()
       }
       n := n − 1
    }
 }
}
```

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

Concurrent Join Operation. In the following, let v be the node with label $\ell(n-1)$. Let $\mathrm{succ}_i(v)$ denote the ith successor of v on the cycle and $\mathrm{pred}_i(v)$ denote the ith predecessor of v on the cycle.

Let the d new peers be $w_1, w_2, \ldots w_d$. Then the supervisor integrates w_i between $\mathrm{succ}_i(v)$ and $\mathrm{succ}_{i+1}(v)$ for every $i \in \{1, \ldots, d\}$. As is easy to check, this will violate Invariant 6.7 for the $2d$ closest successors of v and the $d-2$ closest predecessors of v. But since the supervisor knows all of these nodes, it can directly inform them about the change. In order to repair Invariant 6.8, the supervisor will request information about the dth successor from the d furthest successors of v and will set v to w_d.

Concurrent Leave Operation. Let the d peers that want to leave the system be $w_1, w_2, \ldots, w_d$. For simplicity, we assume that they are outside of the peers known to the supervisor and that they are not in the neighborhood of each other, but our strategy below can also be extended to these cases. The strategy of the supervisor is to replace w_i by $\mathrm{pred}_{2(i-1)}(v)$ for every i. As is easy to check, this will

```
Peer {

 Peer() {
     label := 0     # label of peer v
     succ := NULL    # succ(v)
     pred := NULL    # pred(v)
     sr := new Relay()    # relay point of v
 }

 Join(s: Relay) {    # relay of supervisor
     if (s ≠ NULL) {
         s ← Join(sr)
         super := s    # current supervisor
     }
 }

 Leave() {
     if (super ≠ NULL)
         super ← Leave(label, pred, succ)
         super := NULL
 }

setup(ℓ : Int, p : Relay, s : Relay) {
    label := ℓ
    pred := p
    succ := s
}

setSucc(w: Relay) {
    succ := w
}

setPred(w: Relay) {
    pred := w
}

getSucc(): Relay {
    return succ
}

getPred(): Relay {
    return pred
}

getPredPred(): Relay {
    return pred ← getPred()
}
```

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

violate Invariant 6.7 for the d closest successors of v and the $3d$ closest predecessors of v. But since the supervisor knows all of these nodes, it can directly inform them about the change. In order to repair Invariant 6.7, the supervisor will request information about the dth predecessor from the d furthest predecessors of v and their dth predecessors and will set v to $\mathrm{pred}_{2d}(v)$.

The operations have the following performance.

Theorem 6.9 *The supervisor needs at most $O(d)$ work and $O(1)$ time (given that the work can be done in parallel) to process d join or leave requests.*

6.4 Multiple Supervisors

If a supervised network becomes so large that a single supervisor cannot manage all of the join and leave requests, one can easily extend the supervised cycle to multiple supervisors. Suppose that we have k supervisors $S_0, S_1, \cdots S_{k-1}$. Then the $[0,1)$-ring is split into the k regions $R_i = [(i-1)/k, i/k)$, $1 \leq i \leq k$, and supervisor S_i is responsible for region R_i. Every supervisor manages its region as described for a single supervisor above, i.e., it treats it like a $[0,1)$-interval, except for the borders, and the borders are maintained by communicating with the neighboring supervisors on the ring. The supervisors themselves form a completely interconnected network.

Each time a new node v wants to join the system via some supervisor S_i, S_i forwards it to a random supervisor to integrate v into the system. Each time a node v under some supervisor S_i wants to leave the system, S_i replaces that node with the last node it inserted into R_i. Using standard Chernoff bounds, we get:

Theorem 6.10 *Let n be the total number of nodes in the system. If the join-leave behavior of the nodes is independent of their positions, then it holds for every $i \in \{1, \ldots, k\}$ that the number nodes currently placed in R_i is in the range $n/k \pm O(\sqrt{(n/k) \log k} + \log k)$, with high probability.*

Hence, if n is sufficiently large compared to k, then the multi-supervised cycle has basically the same properties as the single-supervised cycle above. If the join-leave behavior of the nodes is adversarial, then the rules of assigning every new node to the least loaded region R_i and replacing every leaving node with the node inserted last into the most loaded region R_i will keep a balanced distribution of the nodes among the regions.

6.5 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 something similar to Invariant 6.1, with the only difference that we want to keep the labels from $\ell(1)$ to $\ell(n)$ (instead of $\ell(0)$ to $\ell(n-1)$). We will also preserve Inviarant 6.2, 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 6.11 *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 6.12 *At any time, the n nodes 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 6.11(2) fulfills the connectivity requirements of a tree. Since it follows from Lemma 6.3 that every node has a label of size at most $\lceil \log n \rceil$, the depth of the tree can be at most $\lceil \log n \rceil - 1$. □

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

Invariant 6.13 *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 6.11 and 6.13, 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. Notice that for simplicity, we assume again that relay points can be freely exchanged.

Lemma 6.14 *The join and leave operations preserve Invariants 6.11 and 6.13.*

Hence, we arrive at the following theorem.

Theorem 6.15 *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 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 6.16 *The broadcast operation has a dilation of* $O(\log n)$ *and requires a work of* $O(n)$*.*

```
Supervisor {

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

 Join(w: Relay) {
    n := n + 1
    if (n = 1) {
       w ← setup(0, w, w, NULL, NULL, NULL)
       pv := w
       v := w
       sv := w
       ssv := w
    } else {
       if (ℓ(n)&2 = 0) {
          w ← setup(ℓ(n), sv, ssv, ssv, NULL, NULL)
          ssv ← setRightChild(w)
       } else {
          w ← setup(ℓ(n), sv, ssv, sv, NULL, NULL)
          sv ← setLeftChild(w)
       }
       sv ← setSucc(w)
       ssv ← setPred(w)
       pv := sv
       v := w
       sv := ssv
       ssv := ssv ← getSucc()
    }
 }

 Leave(ℓ: Int, pw: Relay, sw: Relay,
          fw, lcw, rcw: Relay) {
    if (n > 0) {
       if (n = 1) {
          pv := NULL, v := NULL
          sv := NULL, ssv := NULL
       } else {
          # remove v from tree
          if (ℓ(n − 1)&2 = 0) sv ← setRightChild(NULL)
                               else pv ← setLeftChild(NULL)
          pv ← setSucc(sv)
          sv ← 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 ≠ w) {
             v ← setup(ℓ, pw, sw, fw, lcw, rcw)
             pw ← setSucc(v)
             sw ← setPred(v)
             if (ℓ&2 = 0)
                fw ← setRightChild(v)
             else
                fw ← setLeftChild(v)
             if (lcw ≠ NULL) lcw ← setParent(v)
             if (rcw ≠ NULL) rcw ← setParent(v)
          }
          # update pointers
          if (pv = w) pv := v
          if (sv = w) sv := v
          ssv := sv
          sv := pv
          v := pv ← getPred()
          pv := pv ← getPredPred()
       }
       n := n − 1
    }
 }
}
```

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

References

[1] K. Kothapalli and C. Scheideler. Supervised peer-to-peer systems. In *Proc. of the 2005 International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2005.

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

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

```
Peer {
  Peer() {
    label := 0    # label of peer v
    succ := NULL    # succ(v)
    pred := NULL    # pred(v)
    parent := NULL
    lchild := NULL
    rchild := NULL
    sr := new Relay()    # relay point of v
  }

  Join(s: Relay) {
    if (s ≠ NULL) {
      s → Join(sr)
      super := s    # current supervisor
    }
  }

  Leave() {
    if (super ≠ NULL)
      super ← Leave(label, pred, succ, parent, lchild, rchild)
      super := NULL
  }

  setup(ℓ : Int, p : Relay, s : Relay, f: Relay,
        lc: Relay, rc: Relay) {
    label := ℓ
    pred := p
    succ := s
    parent := f
    lchild := lc
    rchild := rc
  }

  setSucc(w: Relay) {
    succ := w
  }

  setPred(w: Relay) {
    pred := w
  }

  setParent(w: Relay) {
    parent := w
  }

  setLeftChild(w: Relay) {
    lchild := w
  }

  setRightChild(w: Relay) {
    rchild := w
  }

  getSucc(): Relay {
    return succ
  }

  getPred(): Relay {
    return pred
  }

  getPredPred(): Relay {
    return pred ← getPred()
  }
```

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

```
# operations of supervisor

  Broadcast(m : Message) {
      root ← sendDown(m)
  }

# operations of peer

  Broadcast(m : Message) {
      if (super ≠ NULL) super ← Broadcast(m)
  }

  sendDown(m : Message) {
      if (lchild ≠ NULL) lchild ← sendDown(m)
      if (rchild ≠ NULL) rchild ← sendDown(m)
      # handle broadcast message
  }
```

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