# 9 Group management

So far, we have only looked at how to design an overlay network for a single application, such as broadcasting or a distributed hash table service. However, what would be a good way of handling multiple applications? Here, two basic approaches are possible: maintaining a separate overlay network for each application, or maintaining a single overlay network used by all applications. If the peers are sufficiently stable and do not run too many applications at the same time, then the first approach might be preferable, because the performance of an application is certainly maximized when using an overlay network that was specifically designed to meet the needs of that application. On the other hand, if the join/leave rate of the peers is high and many applications have to be handled at the same time, then the maintenance overhead can become too high for the peers to handle if an extra overlay network is maintained for each of them. In this case, it is much better just to use a single overlay network to interconnected the peers and to embed all applications on top of it. In this section, we will review a variety of possible solutions to run multiple peer-to-peer applications. Peers belonging to a specific application are called a *group* in the following.

## 9.1 Shared space

The most straightforward solution to our problem is to provide a shared space on top of the overlay network by implementing a distributed hash table. This, in principle, allows one to run arbitrary shared memory programs on top of a peer-to-peer system that can be used to implement any distributed application. Such an approach is pursued, for example, in I3 [8], OpenHash [4], KBR [2], and PeerWare [1]. However, remember that accessing a location in the shared memory can take $\Theta(\log n)$ hops in an overlay network of size $n$, so that shared memory accesses have to be used with great care to make sure that a distributed program is running efficiently. Therefore, it is usually much better to use other, more direct ways of realizing distributed applications. A basic building block for all of the strategies presented in this section is a distributed tree management strategy realizing so-called *transparent* trees.

## 9.2 Transparent trees

As the basis for our construction, we assume that every peer $p$ is mapped to a (pseudo-)random point $x_p \in [0,1)$ and that the peers are organized in an overlay network that consists of a doubly linked cycle, in which the peers are ordered according to their points in $[0,1)$, and a dynamic deBruijn graph as specified in the section about the continuous-discrete approach. Every peer $p$ is associated with a *region* $R_p$ that ranges from $x_p$ to the point $x_q$ of its successor $q$ on the cycle.

Our goal is to embed a tree into this network so that every edge in the tree is also an edge in the overlay network and the functionality of the tree can be maintained as long as the functionality of the overlay network can be maintained. We call such trees *transparent trees*.

Let $T$ be an infinite binary tree in which every edge to a left child is labeled with 0 and every edge to a right child is labeled with 1. Let the label of a node $v$ in $T$, $\ell_v$, be the sequence of the labels of all edges encountered when moving along the unique path from $v$ to the root of $T$. For example, the leftmost node in level 2 in $T$ has label 00, and the rightmost node in level 3 of $T$ has label 111. Every tree associated with a key $k \in U$ is denoted by $T_k$ in the following. We first show how to embed a tree $T_k$ into $[0,1)$.

**Embedding of $T_k$ into $[0, 1)$**

Suppose that we are given a hash function $h : U \to \{0, 1\}^*$ that maps keys to (potentially) infinite bit strings. For any two bit strings $s_1$ and $s_2$, let $s_1 \circ s_2$ be the concatenation of $s_1$ and $s_2$. For example, if $s_1 = 0010$ and $s_2 = 1110$, then $s_1 \circ s_2 = 00101110$. Recall the function $r : \{0, 1\}^* \to [0, 1)$ in the section about supervised overlay networks. Every node $v$ in $T_k$ is mapped to the point $x_v = r(\ell_v \circ h(k))$, and a peer $p$ is responsible for node $v$ if and only if $x_v \in R_p$.

**Storing information in $T_k$**

We assume that information is subdivided into atomic entities called *entries*. These entries have to be stored in $T_k$ so that the following invariant is satisfied at any time.

**Invariant 9.1** *For every tree $T_k$, every node in $T_k$ stores at most one entry, and for every node $v$ that stores an entry, also the parent node of $v$ must store an entry.*

The invariant makes sure that the storage load is evenly distributed among the peers and that information is always stored in the tree in a compact form. It also specifies how to recover from faults. If an entry gets lost, then the entries below it are moved upwards so that the invariant is satisfied again.

**Routing a request to $T_k$**

In order to route a request to $T_k$, we use the deBruijn routing strategy given in the section about the continuous-discrete approach. Suppose that a request to $T_k$ starts at position $x \in [0, 1)$, and let $\ell_x$ be the binary representation of $x$. Then the request is forwarded to $r(b_1 \circ \ell_x)$, $r(b_2 b_1 \circ \ell_x)$, and so on, using (pseudo-)random bits $b_i$, till a peer is reached that is responsible for some point $r(b \circ h(k))$ for some bit sequence $b \in \{0, 1\}^*$. Afterwards, the request is moved upwards the tree $T_k$ till a node is reached that stores an entry or the root of $T_k$ is reached. From that point on, the request is processed according to its specifications that depend on the particular context in which the tree is used.

We demonstrate the transparent tree concept by applying it to various problems, starting with a rendezvous service. In the following, given an finite bit sequence $\ell$, $\overline{\ell}$ denotes its reverse, and given an infinite bit sequence $\ell = \ell_1 \ell_2 \ldots$, $\text{prefix}_i(\ell) = \ell_1 \ell_2 \ldots \ell_i$ and $P_\ell$ denotes the unique path starting at the root of some tree $T_k$ that follows the edges with labels $\ell_1$, $\ell_2$, and so on.

## 9.3 Rendezvous service

For certain applications, it is a much better idea to form a dedicated overlay network for each of the applications. But for this a core overlay network and a rendezvous service has to be provided that allows peers that want to join a specific application to find other peers that are already part of that application. This idea is pursued, for example, by JXTA, a Java library created by SUN to provide peer-to-peer services.

JXTA is organized in three layers. The Platform Layer (JXTA Core), the Services Layer, and the Applications Layer. The platform layer encapsulates minimal and essential primitives that are common to peer-to-peer networking, including discovery, transport (including firewall handling), the creation of peers and peer groups, and associated security primitives. A peer group is a collection of peers that have agreed upon a common set of services. Peers self-organize into peer groups, each identified by a unique peer group ID.

All JXTA network resources – such as peers, peer groups, pipes, and services – are represented by an *advertisement*. Advertisements are language-neutral metadata structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existence of peer resources. Peers discover resources by searching for their corresponding advertisements.

The JXTA core network is an ad hoc, multi-hop, and adaptive network composed of connected peers. Peers are separated into three classes: edge peers, rendezvous peers, and relay peers. Every edge peer maintains a connection to a rendezvous peer. Each rendezvous peer maintains its own list of known rendezvous peers. A rendezvous peer may retrieve rendezvous information from a predefined set of bootstrapping, or seeding, rendezvous peers. Rendezvous peers periodically select a random number of rendezvous peers and send them a random list of their known rendezvous peers. Rendezvous peers also periodically purge non-responding rendezvous peers. Thus, they maintain a loosely consistent, random network of known rendezvous peers. Relay peers are used for communication between peers that cannot communicate directly (because of NAT boxes or firewalls).

Rendezvous peers maintain an index of advertisements published by edge peers. Edge peers send search and discovery requests to rendezvous peers, which in turn broadcast requests they cannot answer to other known rendezvous peers. The discovery process continues until one peer has the answer or the request dies. Messages have a default time-to-live (TTL) of seven hops. Loopbacks are prevented by maintaining a list of already visited peers.

Hence, overlay maintenance, routing, and the management of advertisements is rather ad hoc in JXTA whereas our goal will be to design an overlay network for the core that is based on formally analyzed methods.

First, we specify the basic operations that are needed for a rendezvous service. For any key $k$, let $G_k$ be the set of all peers that have registered for the group ID $k$. A distributed rendezvous service must provide the following operations:

- REGISTER($k$): this registers a peer $p$ for the group key $k$. Formally, $G_k = G_k \cup \{p\}$.
- DEREGISTER($k$): this deregisters a peer $p$ for the group key $k$. Formally, $G_k = G_k \setminus \{p\}$
- JOIN($k$): this returns information about any peer that has registered for $k$, i.e. any peer in $G_k$ is returned.

Notice that implementing a rendezvous service is more complex than implementing a shared space because in a rendezvous service there can be multiple entries for a key. The easiest solution here would certainly be to implement a distributed hash table functionality and to store all entries for a particular key $k$ at the node currently responsible for $k$. This, however, can cause a high load imbalance if there are groups that have many registered peers. A better approach is to use transparent trees. In the following, we explain how the operations above act on these trees. We assume that we are given a (pseudo-)random hash function $g : V \times U \rightarrow \{0,1\}^*$ that takes as argument a peer $p$ and a key $k$ and outputs a random bit string of (potentially) infinite length.

### Registering a peer

If a peer $p$ executes REGISTER($k$), then a registration request is routed to $T_k$ as explained in Section 9.2, using $g(p,k)$ for the random bit string $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If $v$ is the root of $T_k$ and does not store an entry, then $p$'s ID is stored in $v$. Otherwise, $p$'s ID is stored at the highest descendent of $v$ in $T_k$ that belongs to the path specified by $g(p,k)$ and does not have an entry yet w.r.t. $T_k$. (We assume here that a peer does not register twice in $T_k$, which can be ensured if each peer remembers all of its current registrations.)

### Deregistering a peer

If a peer $p$ executes DEREGISTER($k$), then a deregistration request is routed to $T_k$ as explained in Section 9.2, using $g(p,k)$ for the random bit string $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or the root of $T_k$. The request is first sent downwards along the path in $T_k$ specified by $g(p,k)$. If an entry for $p$ is not found there, the request is sent towards the root of $T_k$ until an entry for $p$ is found or the root is reached. If an entry for $p$ is found at some node $v$, then that entry is deleted and the entries stored in the descendants of $v$ along the path $P_{g(p,k)}$ in $T_k$ are moved to their parents. Once a node along $P_{g(p,k)}$ in $T_k$ without an entry is reached, it is checked whether its sibling still has an entry. If so, we do downwards along a path from the sibling, using any child that has an entry, and move its entry to the parent, until we reach a node whose children do not store any entry. Then the deregistration operation terminates. In this way, Invariant 9.1 is preserved.

The deregistration protocol above always finds the entry with $p$'s ID if it exists because the registration protocol makes sure that an entry for $p$ is only stored in a node along the path $P_{g(p,k)}$ in $T_k$. Hence, our registration and deregistration protocols are working correctly. It remains to bound their work.

### Work and load bounds

We start with the following result.

**Theorem 9.2** *A* REGISTER*($k$) and* DEREGISTER*($k$) operation needs at most $O(\log n)$ time and communication work to be processed.*

**Proof.** The routing to the root of the tree $T_k$ takes $O(\log n)$ hops, according to the properties of the deBruijn routing protocol. Suppose that $m \leq n$ peers have registered for $T_k$. Consider some fixed node $v$ in level $\lceil \log m \rceil + i$ in $T_k$. Because there are at least $m \cdot 2^i$ nodes in that level, the probability that $v$ has an entry is at most $1/2^i$. But if $v$ has an entry, then also its parent must have an entry due to Invariant 9.1. The probability for this is at most $1/2^{i-1}$. Continuing this argument with the ancestors still level $\lceil \log m \rceil + 1$, we get that the probability that a node $v$ at level $\lceil \log m \rceil + i$ has an entry is at most

$$\prod_{j=1}^{i} \frac{1}{2^j} = 2^{-\sum_{j=1}^{i} j} \leq 2^{-i^2/2} \ .$$

Because there are at most $m \cdot 2^{i+1}$ nodes in level $\lceil m \rceil + i$ in the tree, the probability that there exists a node $v$ in level $\lceil \log m \rceil + i$ that has an entry is at most

$$m \cdot 2^{i+1} \cdot 2^{-i^2/2}$$

which is polynomially small in $n$ if $i = \Theta(\sqrt{\log n})$ is sufficiently large. There, the largest level at which an entry may be stored in $T_k$ is $\log m + O(\sqrt{\log n})$. Thus, also the insert or removal parts of the protocols only take $O(\log n)$ time and work, completing the proof. □

The next result is much more complicated to prove. See [6] for details.

**Theorem 9.3** *No matter how the registrations are distributed among the groups, if the total number of registrations is $m$ and the total number of peers in the system is $n$, then every peer responsible for an interval of size $s/n$ only has to store $O(s \cdot m/n + \log n)$ entries, with high probability.*

Thus, the storage load is (almost) indeed evenly distributed among the peers. Because every peer is only responsible for an interval of size $O((\log n)/n)$, with high probability, the maximum load at a peer is $O((m/n) \log n)$, with high probability. Having a low maximum load is important to make sure that join and leave operations of peers can be executed with a low overhead.

### Joining a group

Finally, we describe how to execute a JOIN($k$) operation. This is simply done by routing join request to $T_k$ as explained in Section 9.2, using a random bit string $b$. It leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or the root of $T_k$. If $v$ stores an entry, then the ID of the peer in that entry is returned, otherwise NULL is returned.

**Claim 9.4** *The* JOIN*($k$) operation needs at most $O(\log n)$ time and communication work to be processed.*

When join requests for the same key $k$ that meet at a peer $q$ are combined by $q$, one can also guarantee a congestion of $O(\log^2 n)$, with high probability, of routing an arbitrary set of $n$ concurrent join requests, one per peer.

## 9.4 Implicit overlay networks

Sometimes, it is too expensive to create an overlay network for each application. In this case, the idea is to use a single overlay network and to embed all the other overlay networks into this single overlay network. One solution to this problem has recently been suggested by Karger and Ruhl [3]. They use the Chord network as the overlay network and describe a method that allows to implicitly embed Chord networks of subgroups of peers running specific applications into that overlay network. We describe an alternative method here.

Consider again the dynamic deBruijn network. Every group of peers running a particular application is again associated with a key $k$. For each key $k$, we again embed a tree $T_k$ into the deBruijn network as described above. However, the management of $T_k$ and the operations differ significantly from the previous approach. Now, the following operations have to be provided:

- JOIN($k$): this allows a peer $p$ to join the group with key $k$.
- LEAVE($k$): this allows a peer $p$ to leave the group with key $k$.
- ROUTE($k, x$): this allows to route a request to the peer in group $k$ responsible for the point $x \in [0, 1)$.

The JOIN and LEAVE operations modify $T_k$ so that the following invariant is maintained at any time:

**Invariant 9.5** *For every tree $T_k$, every node stores at most one entry. This entry stores either a special marker or information about a peer. If there is at least one entry in $T_k$ storing a peer ID, it must hold*

- *for every node $v$ storing a peer ID that all ancestors of $v$ store a marker, and*
- *for every node $v$ storing a marker that it has at least one child storing a marker or a peer ID.*

As for the rendezvous service, we assume that we are given a hash function $g : V \times U \to \{0,1\}^*$ and information about a peer $p$ can only be stored in a node along the path $P_{g(p,k)}$ in $T_k$. Next we show how to maintain all of these requirements.

## Joining a group

If a peer $p$ executes JOIN($k$), then a join request is routed to $T_k$ as described in Section 9.2. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. In the second case, $p$'s ID is stored in the root. Otherwise, it is checked if $v$ contains a marker or a peer ID. If it contains a marker, then the path $P_{g(p,k)}$ is followed downwards till the last node $w$ is reached that stores a marker or an ID of some peer $q$. If $w$ stores a marker, then $p$'s ID is stored in the child of $w$ in $P_{g(p,k)}$. Otherwise, we take the peer $q$ stored in $w$ and go downwards along $P_{g(p,k)}$ and place a marker on each visited node till a node is reached at which $P_{g(p,k)}$ and $P_{g(q,k)}$ differ at its children. Then $p$'s ID is stored in the child in $P_{g(p,k)}$ and $q$'s ID is stored in the child in $P_{g(q,k)}$.

## Leaving a group

If a peer $p$ executes LEAVE($k$), then a leave request is routed to $T_k$ as described in Section 9.2. This again leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. In the latter case, we are done. Otherwise, it is checked if $v$ contains a marker or information about a peer. If it contains a marker, then the path $P_{g(p,k)}$ is followed downwards till the last node $w$ is reached that stores a marker or information about some peer $q$. If $w$ just stores a marker, of the peer ID stored in $w$ is not equal to $p$'s ID, we are done ($p$ was not part of the group). Otherwise, $w$ stores $p$'s ID. In this case, the entry is removed from $w$. The request is send upwards and markers are removed till a node $w'$ is reached whose other child stores a marker or a peer. We traverse $T_k$ downwards on the side of the other child till we reach a node that has two children with entries or a node with a peer entry. In the first case, we are done. Otherwise, let the entry store the ID of peer $q$. We remove $q$'s entry from the node, move upwards the tree and remove the markers along the way till a node $w''$ is reached whose other child stores a marker or a peer. At that point the entry of $q$ is placed at the child of $w''$ we came from.

## Routing in a group

If a peer $p$ executes ROUTE($k, x$), then a request is routed to $T_k$ as specified in Section 9.2, using the binary representation $b(x)$ of $x$ as the bit sequence $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$ and the root is empty. In the latter case, we are done. Otherwise, we follow the path $P_{b(x)}$ downwards till we reach the last node, $w$, that stores a marker or a peer ID. If it stores the ID of some peer $p$, then $p$ is currently responsible for $x$. Otherwise, we follow the unique path downwards from $w$ by always using the 0-edge if possible until we end in a node storing the ID of some peer $q$. Then $q$ is currently responsible for handling $x$.

Because of Invariant 9.5, every point $x \in [0, 1)$ has a peer responsible for it, so the routing always terminates. Also, there is always a unique peer responsible for $x$, namely the peer $p$ with the maximum $i$ so that $\mathrm{prefix}_i(g(p, k)) = \mathrm{prefix}_i(b(x))$ and remaining bit sequence that is smaller than the remaining bit sequences of all other peers $q$ with $\mathrm{prefix}_i(g(q, k)) = \mathrm{prefix}_i(b(x))$. Thus, our routing operation can be used to implement a distributed hash table for the group.

Our protocols have the following performance.

**Theorem 9.6** *The join, leave, and route operations need at most $O(\log n)$ time and work to be processed, with high probability.*

## 9.5 Transparent data structures

Instead of implementing an entire overlay network, suppose that we just want to provide a shared data structure, identified by some key $k$, that can be accessed by any peer in the system. Then one approach would be to implement this data structure on top of a shared space, but this approach may slow down accesses to the data structure significantly. Another approach could be to implement the data structure as an extra overlay network, but then we have to make sure that the overlay network has a high expansion to avoid partitioning problems. This would certainly complicate the design of the data structure. We present a third alternative, which we call *transparent data structures* here. The idea behind these data structures is to *embed* the data structure into the overlay network just as we did with the transparent tree. In this way, maintaining the overlay network also helps to maintain the data structure. For the embedding to work, we need to formulate invariants that allow the data structure to adapt to changes in the overlay network. We do this for several simple data structures including parallel stacks, FIFO queues, heaps, and search trees.

### Pools

A *pool* [5] is a concurrent data-type which supports the following abstract operations

- ENQUEUE($k, e$): adds element $e$ to the pool identified by $k$
- DEQUEUE($k$): deletes and returns some element $e$ from the pool identified by $k$

A stack, for example, is a pool with a last-in-first-out (LIFO) ordering on enqueue and dequeue operations, and a queue is a pool with a first-in-first-out (FIFO) ordering on enqueue and dequeue operations. Since its first introduction by Manber [5], the literature has offered us a variety of possible pool implementations. See [7], for example, for a survey. We will use our transparent tree approach to implement a pool. It works as follows:

If a peer $p$ executes the ENQUEUE($k, e$) operation, then an enqueue request is routed to $T_k$ as described in Section 9.2, using a random bit sequence $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If it is the root and it is empty, $e$ is stored in the root. Otherwise, the request moves down a random path in $T_k$ until it reaches a node $w$ without an entry. Then $e$ is stored in $w$.

If a peer $p$ executes a DEQUEUE($k$) operation, then a dequeue request is routed to $T_k$ as described in Section 9.2, using a random bit sequence $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If it is the root and it is empty, NULL is returned. Otherwise, the request moves down a path in $T_k$ (where a random decision is made if both children of a node store entries) until it reaches a node $w$ that has no children with an entry. Then the entry in $w$ is removed from $T$ and returned to $p$.

If an ENQUEUE$(k, e)$ request meets a DEQUEUE$(k)$ request, the the enqueue request is deleted and $e$ is returned as the answer to the dequeue request.

**Theorem 9.7** *The* ENQUEUE*$(k, e)$ and* DEQUEUE*$(k)$ operations need at most $O(\log \max[n, m_k])$ time and work to be executed where $m_k$ is the number of entries in the pool for key $k$.*

It turns out that many enqueue and dequeue requests can be also handled concurrently with a low congestion (at most $O(\log^2 n)$ in the dynamic deBruijn graph). So transparent trees allow very efficient implementations of pools.

## Call Stacks

A call stack is a special form of pool that is necessary in a distributed environment if a distributed program is executed that spawns a hierarchy of subroutines that provide return values, because then a subroutine can only finish if all of the subroutines spawned by it have returned their values. To distribute the subroutines efficiently among peers interested in executing them, we need a distributed call stack. In order to guarantee that the subroutines are executed in the right order, we assume that each subroutine has a name representing the history of calls creating it. Consider the binary encoding of these names. Let the order "<" be defined so that for any two bit sequences $b$ and $b'$, $b < b'$ if and only if $\text{prefix}_i(b) = \text{prefix}_i(b')$ and $b_{i+1} < b'_{i+1}$ for some $i \geq 0$ or $b = \text{prefix}_{|b|}(b')$ and $|b| < |b'|$. If we can use the bit sequences and the $<$ operator so that subroutines are stored in the nodes of the transparent tree so that every subroutine $r$ with parent $r'$ is stored at a node $v$ that is a descendent of the node storing $r'$, then it suffices for dequeue operations to pick any entry of some node $v$ whose children do not have any entries. Thus, the enqueue and dequeue operations can be implemented as follows:

If a peer $p$ executes the ENQUEUE$(k, e)$ operation, then a request is routed to $T_k$ as described in Section 9.2, using the bit sequence $b(e)$ where $b(e)$ represents the name of $e$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If it is the root and it is empty, $e$ is stored in the root. Otherwise, the request moves down along the path $P_{b(e)}$ in $T_k$ until it reaches a node $w$ without an entry. Then $e$ is stored in $w$. Afterwards, $e$ is replaced with the entry $e'$ at its father if and only if $b(e) < b(e')$, and we continue this upwards until this property is not satisfied any more. The routing stage makes sure that $e$ is stored along the same path as its ancestors in the call hierarchy, and the exchange stage makes sure that the entries are monotonically ordered according to their level in the call history. This satisfies our conditions on the placement of subroutines above.

If a peer $p$ executes a DEQUEUE$(k)$ operation, then a request is routed to $T_k$ as described in Section 9.2, using a random bit sequence $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If it is the root and it is empty, NULL is returned. Otherwise, the request moves down a path in $T_k$ (where a random decision is made if both children of a node store entries) until it reaches a node $w$ that has no children with an entry. Then the entry in $w$ is removed from $T$ and returned to $p$.

The performance of ENQUEUE and DEQUEUE depends on the structure of the call hierarchy and is therefore more difficult to quantify.

### Queues

In order to implement a (relaxed version of a) FIFO queue identified by some key $k$, we also use a transparent tree $T_k$. The following operations need to be implemented for this:

- ENQUEUE$(k, x)$: adds an element $x$ to the FIFO queue
- DEQUEUE$(k)$: removes an element from the FIFO queue

If a peer $p$ executes the ENQUEUE$(k, x)$ operation, then an enqueue request is routed to $T_k$ as described in Section 9.2, using a random bit sequence $b$. This leads the request to a node $v$ in $T_k$ that stores an entry in $T_k$ or that is the root of $T_k$. If it is the root and it is empty, $x$ is stored in the root. Otherwise, the request moves down a random path in $T_k$ until it reaches a node $w$ without an entry. The request moves upwards from here till it reaches the root. In each upwards hop, it moves the entry of the current node to the child the request came from. Finally, the request places $x$ at the root of $T_k$.

If a peer $p$ executes DEQUEUE$(k)$, then this is handled as the DEQUEUE request above.

**Theorem 9.8** *The* ENQUEUE*$(k, x)$ and* DEQUEUE*$(k)$ operations need at most $O(\log \max[n, m_k])$ time and work to be executed where $m_k$ is the number of entries in the queue for key $k$.*

### Heaps

In order to implement a parallel heap, we can also use the transparent tree concept. The following operations need to be implemented for this:

- INSERT$(k, x)$: inserts element $x$ into the heap identified by $k$
- REMOVE(): removes the top entry in the heap

It is quite easy to imagine how to handle these operations so that also a time and work bound of $O(\log \max[n, m_k])$ can be achieved for heap $k$.

### Search trees

Finally, also search trees can be implemented with transparent trees. We leave this as an exercise to the reader.

## 9.6 Fault tolerance and recovery

Finally, after presenting various applications for transparent trees, we summarize why transparent trees are so useful. First of all, they allow the storage load to be evenly distributed among the peers, so that a high join/leave rate can be supported and only a small number of entries is lost if peers fail. Furthermore, as long as the deBruijn network can recover, transparent trees can also recover. But most importantly, transparent trees can be easily turned into self-repairable structures. All that is needed is that the peers continuously check whether Invariant 9.1 is still satisfied for the tree. If not, then they move entries upwards and in addition may execute some application-specific operations for the tree so that it always fully recovers with whatever entries are left.

## References

[1] G. Cugola and G. Picco. PeerWare: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, May 2001.

[2] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[3] D. Karger and M. Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[4] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *Proc. of the 3rd*, 2004.

[5] U. Manber. On maintaining dynamic information in a concurrent environment. *SIAM Journal on Computing*, 15(4):1130–1142, 1986.

[6] C. Scheideler and W. Wang. A load-balanced peer-to-peer registration service and its applications to anycasting and multicasting. Manuscript. Johns Hopkins University, September 2004.

[7] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems, Special Issue*, 30:645–670, 1997.

[8] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. of the ACM SIGCOMM '02*, 2002.