10 Overlay Networks for Heterogeneous Peers

So far, we have only shown how to construct overlay networks for uniform peers. However, the world is non-uniform. Suppose, for example that we have peers of non-uniform bandwidth. If we then use a network based on hashing, like Chord, in order interconnect the peers, it is very likely that high-bandwidth peers will be isolated from other high-bandwidth peers, making them ineffective. Thus, a different design is needed. On possibility is to simply split the peers into many logical peers of uniform bandwidth, but then high-bandwidth peers may have many logical peers in the network, making it more vulnerable and making it more complex for peers to join and leave. Another solution could be to organize the peers in a multi-tier network. That is, peers are organized into groups of approximately the same bandwidth, and each group is organized into some hash-based overlay network presented before. But it is not clear how to interconnect the networks of the various groups because the group sizes can differ quite significantly from each other.

The solution presented here follows a different direction. Instead of using many logical peers or a multi-tier network to incorporate peers of non-uniform bandwidth, every peer is just associated with a single node, and a simple heap property is used to organize the peers in the system: every parent of a peer must have a bandwidth that is at least as large as the bandwidth of that peer. Thus, local, relative rules are used to organize peers instead of the rather global nature of the rules using logical peers or multi-tier networks (since an agreement on the minimum bandwidth and bandwidth-to-tier assignments is necessary there).

The overlay network behind our approach is called *Pagoda* [1]. We start with the static version of the overlay network before we define its dynamic version. Afterwards, we explain how to use the overlay network for routing, data management, and multicasting.

10.1 The static Pagoda network

Our overlay network is basically a combination of a complete binary tree and a family of leveled graphs that are similar to the well-known Omega network [2], together with some short-cut edges to keep the diameter low. It is called *Pagoda*. We first define a perfect, static form of it before describing dynamic constructions.

Definition 10.1 *Let $d \in \mathbb{N}_0$. The d-*dimensional deBruijn*, $DB(d)$, is an undirected graph with node set $V = [2]^d$ and an edge set $E = \{\{x,y\} \mid x,y \in [2]^d$ and there are $p,q \in \{0,1\}$ so that $x = (b_1, b_2, \ldots, b_{d-1}, p)$ and $y = (q, b_1, b_2, \ldots, b_{d-1})\}$.*

Definition 10.2 *Let $d \in \mathbb{N}_0$. The d-*dimensional deBruijn exchange network*, $DXN(d)$, is an undirected graph with node set $V = [d+1] \times [2]^d$ and an edge set:*

$$\begin{aligned} E &= \{\{(j,x),(j+1,y)\} \mid j \in [d-1], \\ &\quad x,y \in [2]^d, \{x,y\} \in E(DB(d)) \text{ or } x = y\} \end{aligned}$$

Figure 1 presents the 3-dimensional deBruijn, $DB(3)$. $DB(d)$ has 2^d nodes and a maximum degree of 4.

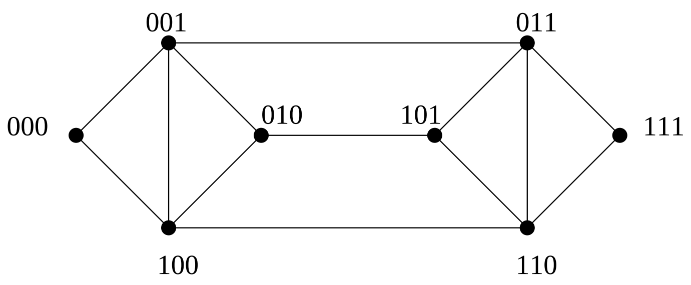

Figure 1: The structure of $DB(3)$

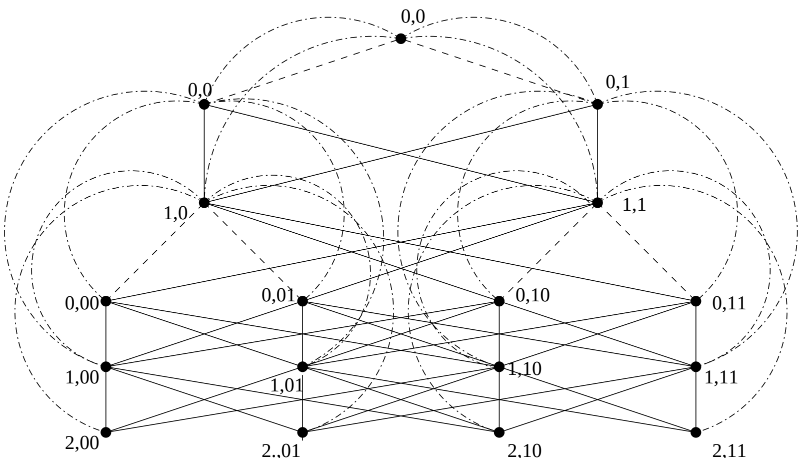

Figure 2: The structure of $PG(2)$.

Definition 10.3 *Let $d \in \mathbb{N}_0$. The d-*dimensional Pagoda*, $PG(d)$, is an undirected graph that consists of $d+1$ deBruijn exchange networks, $DXN(0), \ldots, DXN(d)$, where each node $(i,x) \in [i+1] \times [2]^i$ of $DXN(i)$ is connected to the nodes $(0,x0)$ and $(0,x1)$ in $DXN(i+1)$ and to all nodes $(0,y)$ in $DXN(i+1)$ that have an edge to $(1,x0)$ or $(1,x1)$.*

In addition to this, for every i and $j \in \{0,\ldots,i\}$, every node (j,x) in $DXN(i)$ has short-cut edges to nodes $(j,x0)$, $(j,x1)$, $(j+1,x0)$, and $(j+1,x1)$ in $DXN(i+1)$.

Ignoring the short-cut edges, the Pagoda is a leveled network with the root being at level 0. Levels are consecutively numbered from 0 to $(\sum_{i=0}^{d} i) - 1$. Given a node at level ℓ, the nodes it is connected to in level $\ell - 1$ are called its *parents*, and the nodes it is connected to in level $\ell + 1$ are called its *children*.

The Pagoda network consists of the following types of edges:

- *column edges* connecting (j,x) to $(j+1,x)$ in a DXN,
- *tree edges* connecting (i,x) in $DXN(i)$ to $(0,x0)$ and $(0,x1)$ in $DXN(i+1)$,
- *short-cut edges* connecting (j,x) in $DXN(i)$ to $(j,x0)$, $(j,x1)$, $(j+1,x0)$, and $(j+1,x1)$ in $DXN(i+1)$, and

- *deBruijn edges* representing all remaining edges.

Each type is important for our protocols to work. Column edges and tree edges allow to keep our protocols simple and efficient, deBruijn edges allow to perform efficient routing (and deterministic level balancing in the dynamic Pagoda), and short-cut edges keep the diameter and congestion low.

Basic properties

Figure 2 shows the 2-dimensional Pagoda $PG(2)$. $PG(d)$ has $\sum_{i=0}^{d}(i+1)2^i \approx (d+1)2^{d+1}$ nodes and maximum degree 20. Furthermore, the following fact is easy to see:

Lemma 10.4 *$PG(d)$ has $O(d^2)$ levels and a diameter of $O(d)$.*

$PG(d)$ also has a good expansion. Recall that the node expansion is defined as $\alpha = \min_{U:|U|\le|V|/2} |N(U)|/|U|$ where $N(U)$ is the neighbor set of U.

Lemma 10.5 *$PG(d)$ has an expansion of $\Omega(1/d)$.*

Proof. Using standard techniques, it is not difficult to show that every permutation routing problem in the Pagoda can be routed with congestion $O(d)$. Suppose now that the node expansion is $o(1/d)$. In this case there must be a set U with $|N(U)| = o(|U|/d)$ and $|U| \le n/2$. Then consider the permutation π that requires to send all packets in nodes in U to $\bar{U} = V \setminus U$. In this case, the expected congestion must be $\omega(d)$, contradicting our bound above. Thus, the expansion is $\Omega(1/d)$. $\square$

10.2 The dynamic Pagoda network for uniform nodes

Our basic approach for the dynamic Pagoda network is to keep the nodes interconnected in a network that represents a subnetwork of the static Pagoda network of infinite dimension. In this section, we assume that all nodes have a bandwidth of 1. At any time, the dynamic Pagoda network has to fulfill the following invariant:

Invariant 10.6

(a) For any node in the dynamic Pagoda, all of its parent positions are occupied.

(b) For any pair of nodes v and w in the dynamic Pagoda, v and w are connected in the dynamic Pagoda if and only if v and w are connected in the static Pagoda.

We start with some facts about the dynamic Pagoda network. A node is called *deficient* if it has a missing child along a column or tree edge (i.e. we do not consider missing children reachable via deBruijn edges).

Lemma 10.7 *If Invariant 10.6 is true, then in the dynamic Pagoda with n nodes, the difference between the largest level and the smallest level with deficient nodes is at most $\log n$.*

Proof. Let v be any node of largest level in the Pagoda. Notice that such a node must be deficient. Suppose that v is at position (j, x) in some $DXN(d)$. The fact that every node must have all of its parent positions occupied and the way the *DXN* is constructed ensure that v is connected to at least 2^j nodes at positions $(0, y)$ in $DXN(d)$, where y is either the result of a right shift of x by at most j positions or a left shift of x by at most j positions, padded with arbitrary 0-1 combinations. Thus, if $j = d$, then all positions in row 0 of $DXN(d)$ must be occupied. If $j < d$, then one can easily check that all positions in row j in $DXN(d-1)$ must be occupied. Hence, the difference between the largest level and the smallest level with a deficient node is at most d. Taking this into account, one can show that $d \leq \log n$, which yields the lemma. $\square$

This lemma has some immediate consequences when combining it with results about the static Pagoda:

Lemma 10.8 *If Invariant 10.6 is true, then the dynamic Pagoda with n nodes is a constant degree network and has $O(\log^2 n)$ levels, a diameter of $O(\log n)$, and an expansion of $\Omega(1/\log n)$.*

Next we define local control algorithms that allow nodes to join and leave the system, denoted by the operations JOIN and LEAVE, while preserving Invariant 10.6 at any time (under the condition that nodes depart gracefully).

Join and Leave operations

The basic strategy of the join protocol is to make sure that every new node is inserted at a place that fulfills Invariant 10.6. Suppose that node u wants to join the system. This is done in two stages.

Stage 1 Suppose that node v, at position (j, x) in $DXN(i)$, is initiating JOIN(u) to insert u into the network. If v has a short-cut edge to a node at position $(j, x0)$ in $DXN(i+1)$, then it forwards the request to that node. Let this new node be v'. If v' does not exist then we refer to node v as v'.

We are now at some node v', at position (j', x') in $DXN(i')$. If v' has a short-cut edge to a node at position $(j', x'1)$ in $DXN(i'+1)$ (here the column with suffix 1 is used to ensure an even spreading of JOIN requests), then it forwards the request to that node. Let this node be the new v'. We repeat this until no new v' exists. Call this last node v''.

We are now at some node v'', at position (j'', x'') in $DXN(i'')$. If v'' is not deficient then v'' forwards the request to the node at position $(j''+1, x'')$ in $DXN(i'')$ if $j'' < i''$, and else it forwards the request to the node at position $(0, x''1)$ in $DXN(i''+1)$. This is the new v''. This is repeated until no new v'' exists. Call this last node w. At this point stage 1 ends and we proceed with stage 2 on this node.

Stage 2 Initially, the JOIN request must be at some deficient node w. If $w = (i, y)$ in some $DXN(d)$ with $0 < i < d$, then w requests information about the column child (i.e. the child reachable via the column edge) from all parents of w. If all parents report an existing child, w can integrate u as its column child without violating Invariant 10.6(a). Otherwise, w forwards the JOIN request for u to any parent w' reporting a missing column child, i.e. it is deficient.

If $i = 0$, then w requests information from its parents about each tree child that is a parent of its column child. If all relevant tree children exist, w can integrate u as its column child, and otherwise w forwards the JOIN request to any parent w' reporting a missing tree child.

Finally, if $i = d$, then w picks any of its missing tree children v and requests information from w's parents about each column child that is a parent of v. If all relevant column children exist, w can integrate u at the position of v, and otherwise w forwards the JOIN request to any parent w' reporting a missing column child.

This is continued until u can be integrated.

Suppose that a node u wants to leave the Pagoda. This is also done in two stages. Stage 1 is the same as stage 1 for the JOIN protocol.

Stage 2 Initially, the LEAVE request must be at some deficient node w. If w has a child, then w forwards the request to any one of its children. This is continued until w does not have any children. Once this is the case, w exchanges its position with u so that u can leave the network.

The JOIN and LEAVE protocols above achieve the following result.

Theorem 10.9 *Any isolated* JOIN *or* LEAVE *operation can be executed in $O(\log n)$ time and with constant topological update work.*

Proof. Consider any JOIN request starting at some node v. From the construction, it can be seen that the request is transferred through at most d short-cut edges until the request reaches a node v' in $DXN(d-1)$ (the second largest *DXN* in the system). From a node in $DXN(d-1)$, at most $O(\log n)$ column or tree edges have to be traversed to reach a deficient node w in $DXN(d)$ or $DXN(d-1)$. From node w on, every time the request is transferred to a deficient node, the level of the node w' receiving the request decreases by one. Hence, it follows from Lemma 10.7 that the JOIN request can be transferred along at most $\log n$ deficient nodes. Thus, an isolated JOIN request can be executed in $O(d) = O(\log n)$ time.

Also every LEAVE request is sent along at most d short-cut edges and $O(d)$ column or tree edges until it reaches a deficient node w. From w, it takes at most $\log n$ further nodes to reach a node without children, at which the LEAVE request can be finished. Hence, also any isolated LEAVE request can be executed in $O(d) = O(\log n)$ time.

The bound on the update work (i.e. the number of edge changes) is obvious. □

10.3 Routing

Suppose that we want to route unicast messages in the Pagoda network. Consider any such unicast packet p with source $s = (j, x)$ in $DXN(i)$ and destination $t = (j', z)$ in $DXN(i')$. First, p picks a random pair of real values $(c, r) \in [0, 1)^2$ (a precision of $\log n$ bits for each is sufficient). Then, p is sent in three stages:

1. **Spreading stage:** First, send p from s along column edges and a tree edge to $(i-1, x/2)$ in $DXN(i-1)$. Then, send p upwards to the node $(0, y)$ in $DXN(i-1)$ with y being the closest prefix of r. From there, forward p to the node $(k, y/2)$ in $DXN(i-2)$ with $k/(i-2)$ being closest to c.

2. **Shuttle stage:** Forward p along short-cut edges across nodes (k', y') with k' being closest to c and y' being the closest prefix of r until a node (k', y') in $DXN(i'-2)$ is reached.

3. **Combining stage:** Perform stage 1 in reverse direction (with s replaced by t) to forward p to t.

Notice that as long as s and t are non-deficient nodes, this strategy is successful even *while* nodes join and leave the system, because the position of every node that is an non-deficient node will be fixed in the Pagoda. Also, whenever a node leaves, the node replacing it can inherit its packets so that no packet gets lost. More general strategies for ensuring reliable communication even while nodes are moving, using the concept of virtual homes, can be found in Section 10.8.

With these facts in mind, one can easily design a protocol based on the random rank protocol (see, e.g., [3]) to show the following result:

Theorem 10.10 *If every node wants to send at most one packet, the packets have random destinations, and every node being the destination of a packet does not move for $O(\log n)$ steps, the routing strategy above can route the packets in $O(\log n)$ time, with high probability.*

10.4 Data management

Finally, we show how to dynamically manage data in Pagoda. We use a simple trick to distribute data evenly among the nodes of the Pagoda so that it is searchable. Suppose that we have a (pseudo-)random hash function mapping each data item to some real vector $(c, r) \in [0, 1)^2$. The current place of a data item d is always the lowest possible position (j, x) in the Pagoda where x is the closest prefix of r and $j/|x|$ is closest to c among all $j'/|x|$ with $0 \leq j' \leq |x|$ ($|x|$ denotes the length of x, and thus the dimension of the *DXN* owning (j, x)).

This strategy implies that if $DXN(d)$ represents the largest exchange network that has occupied positions in the Pagoda, then all data items will be stored at nodes in $DXN(d-2)$, $DXN(d-1)$, or $DXN(d)$. Since every node will at most have to store an $O(1/(d \cdot 2^d))$ fraction of the data and $d \cdot 2^d = \Theta(n)$, we get:

Theorem 10.11 *The data management strategy ensures that every node is only responsible for an expected $O(1/n)$ fraction of the data at any time, and this bound even holds with high probability if there are at least $n \log n$ data items in the system.*

Notice that none of the DHT-based systems can achieve the bounds above in their basic form – they only achieve a bound of $O(\log n/n)$. Combining the data management strategy with our routing strategy above, requests to arbitrary, different data items with one request per node can be served in $O(\log n)$ time, w.h.p. The results in Section 10.6 imply that this also holds for cases in which some nodes want to access the same data item, i.e. we have a multicast problem, if requests can be combined.

10.5 The dynamic Pagoda network for non-uniform nodes

Next we show that the Pagoda network can also be used for arbitrary non-uniform node bandwidths. In this case, we want to maintain the following heap property to allow efficient multicasting.

Invariant 10.12 *For any node v in the Pagoda,*

(a) all of its parent positions are occupied, and

(b) the bandwidth of v is at most the bandwidth of any of its parents.

Similar to the uniform case, we require these invariants to be fulfilled *while* nodes join and leave the system. Because of item (b), we cannot just do a single exchange operation to integrate or remove a node but we have to be more careful. First, we describe the JOIN and LEAVE operations for the isolated case, and then we consider the concurrent case.

Join and Leave operations

For any node u in the Pagoda, *max-child(u)* refers to the child of maximum bandwidth and *min-parent(u)* refers to the parent with minimum bandwidth.

Suppose that node v is executing JOIN(u) to insert a new node u with bandwidth $b(u)$ into the network. This is done in three stages. Stages 1 and 2 are identical to the uniform case. So it remains to describe stage 3 which is similar to inserting a node in a binary heap.

Stage 3 Once the JOIN request for u has reached a deficient node with an empty column or tree child position in which u can be integrated without violating Invariant 10.12(a), u is integrated there with *active bandwidth* $a(u)$ equal to the minimum of $b(u)$ and the bandwidth of its min-parent. The active bandwidth is the bandwidth it is allowed to use without violating Invariant 10.12(b). Then, u repeatedly compares $b(u)$ with $a(u)$. If $a(u) < b(u)$, it replaces its position with the position of its min-parent and afterwards updates $a(u)$ to $\min\{b(u), b(\text{min-parent}(u))\}$. Once u reaches a position with $a(u) = b(u)$, the JOIN protocol terminates. The process of moving u upwards is called *shuffle-up*.

Suppose that a node u wants to leave the Pagoda. Then it first sets its active bandwidth to $b(u)$. Afterwards, u repeatedly replaces its position with its max-child and updates its active bandwidth to $a(u) = b(\text{max-child}(u))$ until it reaches a position with no child. At this point, u is excluded from the system so that Invariant 10.12 is maintained. The process of moving u downwards is called *shuffle-down*.

Bandwidth changes

If the bandwidth of some node u increases, we use the shuffle-up procedure, and if the bandwidth of some node u decreases, we use the shuffle-down procedure to repair the invariant.

Isolated update requests have the following performance.

Theorem 10.13 *Any isolated join operation, leave operation, or bandwidth change of a node needs* $O(\log^2 n)$ *time and work to repair the invariant.*

Proof. First, consider the insertion of some node u. The process of moving the request of u downwards only needs $O(\log n)$ time. According to Lemma 10.8, u is integrated at some level $\ell = O(\log^2 n)$. Hence, the shuffle-up process only requires $O(\log^2 n)$ messages and edge changes because each exchange of positions between u and some parent v to repair Invariant 10.12 moves u one level upwards and requires updating only a constant number of edges. Every shuffle operation maintains the invariant for all nodes involved in it. Hence, the total time and work is $O(\log^2 n)$.

Similar arguments can be used for node departures and bandwidth changes. □

The concurrent JOIN protocol

The only difference between the isolated and concurrent JOIN protocol is that we are more careful about exchanging positions. If a node u wants to replace its position with some parent v, then u checks whether v is a node that has not finished its JOIN operation or bandwidth increase operation yet (i.e. $a(v) < b(v)$). If so, u does nothing. Otherwise, u replaces its position with v.

The concurrent LEAVE protocol

Also the concurrent LEAVE protocol is similar to the isolated LEAVE protocol, with the only difference that if some node u in the process of leaving the network wants to replace its position with some child v, u first checks whether v is a node that has not finished its LEAVE operation or bandwidth decrease yet (i.e. $a(v) > b(v)$). If so, u does nothing. Otherwise, u replaces its position with v.

Bandwidth increase or decrease is handled similarly. The next lemma shows that the concurrent operations always terminate with a work that is at most the sum of the work for isolated update operations.

Lemma 10.14 *For any set of k concurrent insertions, deletions, and bandwidth changes of nodes, the work and time required to repair Invariant 10.12 is $O(k \log^2 n)$.*

Proof. The work bound is obvious. Thus, it remains to prove the time bound.

Consider k concurrent update requests. From the analysis in the uniform case we know that $O(k \log n)$ work is necessary for nodes of JOIN requests to be integrated into the system. Each time step progress is made here until all JOIN requests are integrated.

Afterwards, we mark all nodes with 1 that have not completed their JOIN or bandwidth increase operation yet, all nodes with -1 that have not completed their LEAVE or bandwidth decrease operation yet, and all other nodes with 0. Suppose that there is at least one node marked as 1. Then let v be any of these nodes of minimum level. Since the level of v must be at least 1 (as the root cannot be a 1-node), it can replace its position with its min-parent, thereby making progress. On the other hand, suppose that there is at least one node marked as -1. Then let v' be any of these nodes of maximum level. If v' does not have any children, then v' can leave, and otherwise it can replace its position with its max-child, thereby making progress in any case.

Hence, we make progress in every step. Since the total work of the shuffle-up, shuffle-down, and departure operations is bounded by $O(k \log^2 n)$, the time spent for executing these operations is also bounded by $O(k \log^2 n)$. □

10.6 Multicasting

Finally, we study how well the non-uniform Pagoda supports arbitrary concurrent multicasting.

Competitiveness

In this section we show that the Pagoda network is $O(\Delta_{\text{OPT}} + \log n)$-competitive with respect to congestion in the best possible network of degree Δ_{OPT} when the multicast problem is posed as a flow problem. We are given a set of client-server-demand triples called *streams*, (T_k, s_k, D_k), where T_k is a set of client nodes served by a server node s_k and D_k is a demand vector which specifies the flow demanded of s_k by each client node. We start by constructing a flow system for one server, s_k, and one client $t \in T_k$. We name this flow system, $f_{k,t}$. We assume that s_k is a node in $DXN(i)$ and t is a node in $DXN(j)$.

1. **Spreading stage:** This stage spreads flow originating at s_k in $DXN(i)$ evenly among the nodes in $DXN(i-2)$. This is done in three steps.

   a. Move the flow from s_k along column edges to the top node in $DXN(i)$.

   b. Move the flow upwards to the bottom node in $DXN(i-1)$ along the tree edge connecting the two DXN's. From there, cut the flow into 2^{i-1} flow pieces of uniform size and send piece i upwards to node $(0, i)$ along the unique path of deBruijn edges representing right shifts.

   c. Move all flow from the top nodes in $DXN(i-1)$ to the bottom nodes in $DXN(i-2)$ along tree edges. Every bottom node in $DXN(i-2)$ sends flow along its column edges so that each node in the column gets the same fraction of flow. That is, at the end every node in $DXN(i-2)$ has a $1/((i-1)2^{i-2})$ fraction of the flow of s_k.

2. **Shuttle stage:** Short-cut edges are used to send the flows forward to $DXN(j-2)$ (which may be upwards or downwards in the Pagoda) so that the flows remain evenly distributed among the nodes in each exchange network visited from $DXN(i-2)$ to $DXN(j-2)$.

3. **Combining stage:** This stage is symmetric to stage 1, i.e. we reverse stage 1 to accumulate all flow in t.

This results in a flow system, $f_{k,t}$, for a source s_k and a destination $t \in T_k$. Let $f_{k,t}(e)$ be the flow through any edge e in this flow system. The procedure is repeated for each client $t \in T_k$. We now construct a flow system, f_k, for the stream k. We lay the flow systems $f_{k,t}$ one on top of the other. The flow through an edge in system f_k is the maximum flow through the same edge in each $f_{k,t}$. That is, let $f_k(e)$ be the flow through any edge e in flow system f_k. Then $f_k(e) = \max_{t \in T_k} f_{k,t}(e)$. Note that we select the maximum flow because if there are two flows of the same stream going through an edge then we simply keep the one with the higher bandwidth (the lower bandwidth stream may be reconstructed from the higher one). We use flow system f_k to route multicast flow for stream k. We show that this strategy yields a low congestion.

Theorem 10.15 *The Pagoda network on n nodes of non-uniform bandwidth that satisfies Invariant 10.12 has a competitive ratio of $O(\Delta_{\text{OPT}} + \log n)$ for any multicast flow problem compared to the congestion in an optimal network for this problem whose degree is bounded by Δ_{OPT}.*

Proof. Let OPT be a network that routes the given flow system with minimum possible congestion C_{OPT}, i.e. that minimizes the maximum amount of flow through a node. W.l.o.g. we assume that every demand is at most the bandwidth of the source and destination.

Select any node u in pagoda. Let it be in exchange network $DXN(i)$. We show that the congestion at this node due to the flow system resulting from our routing strategy above is no more than $O(\log n) \cdot C_{\text{OPT}}$ due to stages 1 and 3 and $O(\Delta_{\text{OPT}}) \cdot C_{\text{OPT}}$ due to stage 2. We show these bounds in parts. We first bound the congestion at u due to stage 1, $c_1(u)$. The flows through u due to stage 1 are the sum of the flows that originate in $DXN(i)$, $DXN(i+1)$ and $DXN(i+2)$. Let the congestion due to each of these be $c_{1a}(u)$, $c_{1b}(u)$ and $c_{1c}(u)$ respectively. Clearly, $c_1(u) = c_{1a}(u) + c_{1b}(u) + c_{1c}(u)$. We bound each of these three separately:

Stage 1a: Node u receives flow from nodes that are below it (in the same column) in exchange network $DXN(i)$. We call this set S. The flow is $\sum_k \max_{v \in S}\{d_k(v)\}$. Note that the max term is used since flows belonging to the same stream are combined, resulting in a flow of largest demand among these. Therefore, the congestion at u is $c_{1a}(u) = \frac{1}{b(u)} \sum_k \max_{v \in S}\{d_k(v)\} \le \sum_{v \in S} \sum_k \frac{d_k(v)}{b(v)} \le |S| \cdot C_{\mathrm{OPT}}$. The set S contains at most $\log n$ nodes. Therefore $c_{1a}(u) \le \log n \cdot C_{\mathrm{OPT}}$.

Stage 1b: Node u receives flow from the bottom nodes of *DXN*(i). Let $f'_k(\cdot)$ be the flow sent up by a bottom node. Thus, each bottom node sends a flow of $f'_k(\cdot)/n$ to each top node. Note that f' is purely the spreading caused by stage 1b.

Let S be the set of bottom nodes with paths crossing u, and let D be the set of top nodes with paths crossing u. We bound $|S|$ and $|D|$ as follows:

Let u be in level h of *DXN*(i). There are 2^i nodes in each level of *DXN*(i), and each node has an address of i bits. Due to the bit-shift routing of the de Bruijn graphs, the nodes in set S must have the same $i - h$ first bits as u has last bits. Thus, the first h bits can be anything, and $|S| = 2^h$. In a similar manner, the nodes in D must have the same first h bits as the nodes in S, thus $|D| = 2^{i-h}$. Now, the number of paths crossing u is $|S| \cdot |D| = 2^i$.

The flow from each node $v \in S$ that reaches u is $\frac{f'_k(v) \cdot |D|}{2^i}$, which is the number of nodes in D times the amount of flow destined for each node in the top row of $DXN(i)$. Since $\frac{|D|}{2^i} = \frac{1}{|S|}$, this becomes $\frac{f'_k(v)}{|S|}$.

Since flows belonging to the same multicast group merge into one flow equal to the maximum of the two it follows that the flow that reaches u is $\sum_k \max_{v \in S} \frac{f'_k(v)}{|S|}$. Assuming v_1 and v_2 are the two tree children of v, the congestion at u is

$$\begin{aligned} c_{1b}(u) &= \frac{1}{b(u) \cdot |S|} \sum_k \max_{v \in S} f'_k(v) \le \sum_{v \in S} \frac{\sum_k f'_k(v)}{b(v) \cdot |S|} \\ &\le \frac{1}{|S|} \sum_{v \in S} c_{1a}(v_1) + c_{1a}(v_2) \le 2 \log n \cdot C_{OPT} \end{aligned}$$

Stage 1c: Node u receives flow from the bottom node in its column. Therefore, the congestion at u, $c_{1c}(u)$ is at most the congestion at the bottom node in the exchange network. The bottom node receives flow from its two descendants in *DXN*$(i + 1)$. Note that the two descendants will send up equal flows, let one of them be v. So, $c_{1c}(u) \le 2c_{1c}(v) \le 4 \log n \cdot C_{\mathrm{OPT}}$.

We show the bounds for flows due to stage 2 with the help of Lemma 10.16. We need to lower bound the congestion that an optimal network can achieve. We do this by showing how an optimal network with bounded degree has limited bandwidth to send flows.

Lemma 10.16 *Let E_{OPT} be the set of edges in the optimum network. For any pair of sets X and Y that are subsets of the set of nodes, let $D(X, Y) = \sum_{s_k \in X} max_{v \in T_k \cap Y} \{d_k(v)\}$ and $B(X, Y) = \sum_{(u,v) \in E_{\mathrm{OPT}} \cap X \times Y} min\{b(u), b(v)\}$. Then $C_{\mathrm{OPT}} \ge D(X, Y)/B(X, Y)$.*

Proof. Consider any pair of sets $X, Y \subseteq V$. $B(X, Y)$ as defined in the statement measures the bandwidth between sets X and Y. Note that it is not necessary that X and Y form a cut. Similarly,

$D(X,Y)$ is the demand that X asks of Y. The ratio of $B(X,Y)$ to $D(X,Y)$ is the average congestion. The max congestion must be at least the average congestion. Therefore $C_{\text{OPT}} \geq \frac{D(X,Y)}{B(X,Y)}$. □

Stage 2: Let U be the set of nodes in the Pagoda which belong to all exchange networks above and including *DXN*$(i+1)$. Let Z be all nodes in exchange network *DXN*$(i+2)$. Let V be all nodes below and including exchange network *DXN*$(i+3)$. Let the collective flow through exchange network *DXN*(i) be f. Any stream whose source is in $U \cup Z$ and has a destination in $V \cup Z$ must go through *DXN*(i). The expression for the flow is:$f = \sum_{s_k \in U \cup Z} \max_{v \in V \cup Z} d_k(v)$. Due to lemma 10.16 we bound f as follows: $f \leq (|U| \Delta_{\text{OPT}} \max_{i \in V \cup Z}\{b_i\} + |U \cup Z| \Delta_{\text{OPT}} \max_{i \in V}\{b_i\} + |Z| \Delta_{\text{OPT}} \max_{i \in Z}\{b_i\}) \cdot C_{\text{OPT}}$.

The first term accounts for bandwidth between U and $V \cup Z$, the second term for bandwidth between V and $U \cup Z$, and the third term for bandwidth within Z. Hence, $f \leq 3\,|U \cup Z| \Delta_{\text{OPT}} \max_{i \in V \cup Z}\{b_i\} \cdot C_{\text{OPT}} \leq 3\,|U \cup Z| \Delta_{\text{OPT}}\, b_u \cdot C_{\text{OPT}}$.

Since the Pagoda spreads tree flow evenly across all nodes in each exchange network, the flow through u is at most $\frac{f}{|DXN(i)|}$. Therefore $c_2(u) \leq \frac{f}{|DXN(i)| \cdot b_u}$. The construction of the Pagoda implies that $|U \cup Z| < 2\,|Z|$, and $|DXN(i)| \geq \frac{|Z|}{12}$. Thus, $c_2(u) \leq 72\,\Delta_{\text{OPT}} \cdot C_{\text{OPT}}$.

The congestion at u due to stage 3 is identical to the congestion due to stage 1 because the two cases are symmetric. Hence, $c(u) = 2\,c_1(u) + c_2(u) \leq (14 \log n + 72\,\Delta_{\text{OPT}}) \cdot C_{\text{OPT}}$. The theorem follows. □ □

10.7 Turning multicast flows into trees

In practice, it may be expensive or impossible to divide and recombine streams. Instead, we choose a pseudo-random hash function h that maps every node v in the Pagoda to a pair of real values $(c, r) \in [0,1)^2$. Similar to the routing strategy in Section 10.3, we can then adapt the multicast scheme in the following way for a source s and target t:

1. **Spreading stage:** (a) is the same as above, but instead of spreading the flow in (b), we route all flow to the node $(0, y)$ in $DXN(i-1)$ with y being the closest prefix of r. From there, forward the flow to the node $(k, y/2)$ in $DXN(i-2)$ with $k/(i-2)$ being closest to c.

2. **Shuttle stage:** Forward the flow along short-cut edges across nodes (k', y') with k' being closest to c and y' being the closest prefix of r until a node (k', y') in $DXN(j-2)$ is reached.

3. **Combining stage:** Reverse the spreading stage to send the flow to t.

Multicast flows that belong to the same stream are combined so that for every edge e, the flow for that stream through e is the maximum demand over all flows of targets t that are part of that stream.

Using this rule, it is not surprising that the expected congestion of our integral flow scheme is equal to the congestion of the divisible flow scheme above.

Theorem 10.17 *The integral multicast flow scheme has an expected competitive ratio of $O(\Delta_{\text{OPT}} + \log n)$ compared to an optimal network with degree Δ_{OPT}.*

Proof. The theorem can be shown by following the line of arguments in the proof of Theorem 10.15. Here, we just give an intuition of why the theorem is correct. We start with bounding the expected congestion for stages 1 and 3.

Lemma 10.18 *The expected congestion from routing the spreading stage is $O(\log n)$-competitive against an optimal network of degree Δ_{OPT}.*

Proof. Let d_i be the total demand requested by node i across all streams, and let b_i be node i's bandwidth. Consider the congestion on any node in DXN(i) first. Since flow is sent up along column edges, the worst congestion occurs at the top nodes of $DXN(i)$. If d_{max} is the largest demand of any node in some node v's column, then v must route at most $(i+1) \cdot d_{max}$ demand, under the worst case assumption that the demands are for different streams and cannot be combined. Since v has at least the bandwidth of every node with demand d_{max}, this is $O(\log n)$-competitive. This analysis also applies to routing along column edges in the final stage.

Now consider the congestion on any node in DXN($i-1$) caused by the spreading stage. We know that the nodes on the bottom of DXN($i-1$) are $O(\log n)$-competitive, because their congestion is at most twice the congestion at the nodes at the top of DXN(i). Since each stream is going to a random, independently selected location in DXN($i-2$), each is going to a random node at the top of DXN($i-1$). Thus, the expected congestion at the top is balanced and therefore is $O(\log n)$-competitive. Furthermore, congestion is caused by streams crossing nodes in the middle of the DXN, but the self-routing properties of the deBruijn graph (which extend to the DXN) imply that the maximum expected congestion in the middle is between the maximum expected congestion in the bottom and the maximum expected congestion in the top part and therefore no worse. Hence, also the nodes in the middle of the graph are $O(\log n)$-competitive in congestion. □

Next we consider stage 2.

Lemma 10.19 *The expected congestion from routing flow in the shuttle stage is $O(\Delta_{\text{OPT}} + \log n)$-competitive against an optimal network of degree Δ_{OPT}.*

Proof. Consider the boundary between any two DXN networks. The flows crossing this boundary upwards (resp. downwards) along short-cut edges must have a set of sources S and a set of destinations T with $S \cap T = \emptyset$. Hence, there is a cut in the optimal network that all these flows have to cross. Furthermore, since we are sending exactly one copy of the stream across the cut, we are sending no more flow than OPT must send. The same upper bound on the amount of flow across a cut holds as in the divisible flow case. Since the nodes along which the flows travel are randomly selected, the expected congestion at any node is a fraction of the total flow proportional to the number of nodes in the DXN, which implies that the congestion is expected to be $O(\Delta_{\text{OPT}} + \log n)$-competitive. □

Combining the two lemmata yields Theorem 10.17. □ □

10.8 Multicast streaming

Next, we address the issue of how to use the multicasting capabilities for multimedia streaming where peers can enter and leave a multicast stream at any time. To ensure reliable streaming, a mechanism is needed to join and leave a multicast stream, to reserve bandwidth in the nodes along that stream, and to use a local admission control rule for admitting multicast stream requests in a fair and transparent way.

Joining and leaving a multicast stream

Consider the situation that node u in the Pagoda wants to join a multicast stream S of source s. Node u then prepares a control packet containing the demand d requested by it and sends the control packet to s as described in Section 10.7. Along its way, the control packet will try to reserve a bandwidth of d. If it succeeds, it will continue to reserve bandwidth along its way until it reaches a point in which for the stream S a bandwidth of at least d is already reserved.

Every node along the multicast stream will only store for each of its incoming edges the client requesting the stream with the largest demand.

Suppose now that some node u wants to leave a multicast stream S. Then it first checks whether it is the client with largest demand for S that traverses itself by checking its incoming edges. If not, u does not need to send any control packet. Otherwise, u checks whether there is a path of some client v for S into u. If so, u prepares a control packet with the largest demand of these clients. Otherwise, u prepares a control packet with demand 0. This control packet is sent towards the source s of S as in Section 10.7. Each time the control packet reaches a node v that is also traversed by other clients to S (that arrive at different incoming edges), the demand of the control packet is updated to the largest demand of these clients. This is continued until the control packet reaches a node v traversed by some client for S with demand larger than the original demand of u.

Rate reservation

For a rate reservation scheme to be transparent and fair, a policy is needed that gives every peer a simple, local admission control rule with the property that if a request is admissible according to this rule, then the rate reservation request should succeed with high probability. We will investigate two such rules:

Suppose that every node v representing a server in the network offers multimedia streams $s_1^{(v)}, s_2^{(v)}, \ldots$ with rates $r_1^{(v)}, r_2^{(v)}, \ldots$ so that $\sum_i r_i^{(v)} \leq b(v)$. Then consider the following rules for some client v.

- **Admission rule 1:** Admit any multicast request to some server w as long as $b(v) \leq b(w)$ and the total demand of the requests in v does not exceed $\epsilon b(v)/\log n$.

- **Admission rule 2:** Admit any multicast request to some server w as long as v is not belonging to any other multicast group and the demand of the request does not exceed $\epsilon \min\{b(v), b(w)\}/\log n$.

Rule 1 will normally be the case in practice because servers of streams usually have a higher bandwidth than clients, but rule 2 would also allow multicasting if this is not true.

Theorem 10.20 *When using admission rule 1 or 2, every request fulfilling this rule can be accommodated in the Pagoda, w.h.p.*

Proof. Recall the integral multicast routing strategy in Section 10.7. Consider any multicast problem that fulfills rule 1 or rule 2. Using the proof of Theorem 10.15, one can easily show that for any node u in the Pagoda, $c_{1a}(u) = c_{1b}(u) = c_{1c}(u) = O(\epsilon)$ and $c_2(u) = O(\epsilon)$. Hence, the expected total amount of demand traversing u is $O(\epsilon b(u))$. Since any single demand through u can be at most $\epsilon b(u)/\log n$ (demands from or to a node v will always traverse only nodes w with $b(w) \geq b(v)$), and the flows for different servers follow paths chosen independently at random, it follows from the well-known Chernoff bounds that the total amount of demand traversing u is also $O(\epsilon)$ with high probability.

Hence, making the constant ϵ small enough, the admission rules 1 and 2 will work correctly with high probability. $\square$

Notice that also a combination of rules 1 and 2 is allowed.

10.9 Multicasting in a dynamic setting: virtual homes

Our multicast tree approach above has several problems. First, it requires to know the position of the server in the Pagoda to join a stream from it, and second, it requires to update the multicast stream each time the server or a client moves. Fortunately, this problem has an easy solution: For every node v, let $h(v) \in [0, 1)^2$ be chosen *independent* on its position in the Pagoda. For example, $h(v)$ may depend on v's IP address. Then v can treat the node closest to $h(v)$ two DXNs above v as its *personal virtual home* that only has to move if v leaves its current DXN.

Suppose that every node continuously informs its virtual home about its current position and that virtual home responsibilities are exchanged whenever nodes exchange positions. Then v only has to update its connection to the multicast stream if it leaves its current DXN. However, when using the short-cut edges, such an update can be done in constant time so that the disruption of service to v is kept at a minimum. While frequent switches between DXNs could cause frequent update operations, a lazy virtual home update strategy can be used to easily solve this problem.

A third problem with dynamic conditions is that intermediate nodes may change their requested bandwidth. We can use *active* bandwidth restrictions to ensure that the previous invariant continues to hold, so that routing is still valid. Since the invariant continues to hold, congestion remains low and the admission control theorems remain true.

References

[1] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *Proc. of the 16th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 170–179, 2004.

[2] D. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, 1975.

[3] C. Scheideler. *Universal Routing Strategies for Interconnection Networks*, volume 1390 of *Lecture Notes in Computer Science*. Springer, 1998.