11 Peer-to-Peer Systems II

Finally, we present a distributed approach for organizing nodes in an overlay network with deterministic guarantees. The topology maintained in this approach is called a *Hyperring*. It supports primitives:

- Add(v): This adds node v to the Hyperring next to the node that v contacted.
- Remove(v): This removes node v from the Hyperring. We assume here that a node can only remove itself.

First, we present methods that allow to execute these primitives in a sequential way, and afterwards we demonstrate that it is also possible to construct mechanisms so that they can be executed massively in parallel.

11.1 The sequential Hyperring scheme

We start with a description of the Hyperring topology. The Hyperring has a subtle but important feature distinguishing it from other dynamic topologies: It does not only have a ring of nodes as its basic structure, but also the skip list is organized as a hierarchical structure of rings. Suppose that the network currently consists of n nodes. Then the Hyperring will consist of approximately $\log n$ levels of rings, starting with level 0. Each level $i \geq 0$ consists of 2^i directed cycles (i.e. cycles in which all edges have the same orientation) of approximately $n/2^i$ nodes, which we call *rings*. All rings have the same orientation, and for every ring at level i, two rings of level $i+1$ share its nodes in an intertwined fashion. A ring at level i will also be called an i*-ring* in the following, and a level i edge will simply be called an i*-edge*. Central conditions for the Hyperring topology are:

> The nodes have to form a single, directed cycle at level 0. Furthermore, for every i-ring that has $(i+1)$-edges on top of it, it must have exactly two intertwined $(i+1)$-rings and the edge orientation of the $(i+1)$-rings must conform to the edge orientation of the i-ring. Also, every node must be the outgoing and incoming node of *exactly one* $(i+1)$-edge. This ensures that the rings are indeed directed cycles and that every node must participate in exactly one of them for each level.

Apart from this we need conditions on how $(i+1)$-edges bridge i-edges for every i. Consider some i-ring R and let (u, v, w, x) be four consecutive nodes on R. We say that (u, v, w, x) form an i*-bridge* (or simply a *bridge* if i is clear from the context) if there is an $(i+1)$-edge from u to x and an $(i+1)$-edge from v to w. An $(i+1)$-edge is called *perfect* if it bridges exactly two i-edges. Another central condition we will need is:

> For every $i \geq 0$, every $(i+1)$-edge must bridge at least one and at most three i-edges, and imperfect $(i+1)$-edges *only* occur in bridges.

Apart from this condition we will also make sure that bridges are sufficiently far apart from each other so that the discrepancy between the size of the two $(i+1)$-rings on top of an i-ring is not too high. Note that zigzag patterns of $(i+1)$-rings on top of i-rings are not possible, since we demand that all edges must have the same orientation.

We start with a claim showing that the Hyperring rules above can in principle be fulfilled for every ring of sufficiently large size:

Claim 11.1 *For every $r \geq 4$ it holds: for every i-ring R of size r, two intertwined $(i+1)$-rings R' and R'' can be constructed on top of R whose lengths differ by at most one and that use at most one bridge.*

Proof. If r is even, then perfect $(i+1)$-edges certainly suffice to prove the claim. In this case, $|R'| = |R''|$. Otherwise, we first ignore a node, say v, in R and construct R' and R'' out of perfect $(i+1)$-edges on top of the remaining nodes. Then we add v back to R and choose either its left or right neighbor, say v', to move the endpoint of the edge from v' bridging v to v and to create an $(i+1)$-edge $\{v, v'\}$. This creates a single bridge with inner edge $\{v, v'\}$, and $|R'|$ and $|R''|$ only differ by 1. $\sqcap$

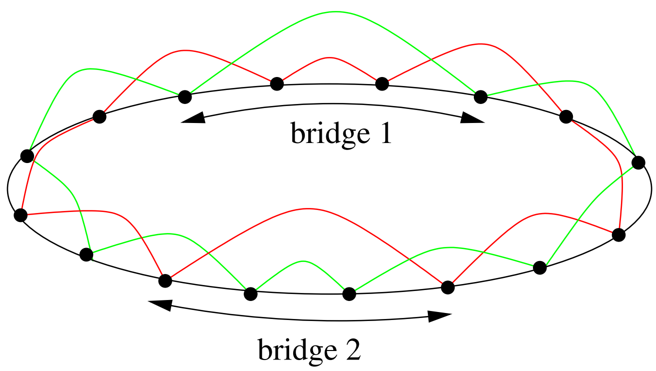

Figure 1: An example of a Hyperring with two levels and two 0-bridges.

Of course, if we always wanted to make sure that every i-ring contains at most one $(i+1)$-bridge, we would create too much update work. Therefore, we only demand that $(i+1)$-bridges are sufficiently far apart from each other. Figures 2 and 4 show how this can be preserved when inserting and deleting a node. The algorithms presented there use the following notation:

For every node v, let $\deg(v)$ denote the out-degree of v. (If v is participating in i levels, then $\deg(v) = i$.) Furthermore, let $\mathsf{succ}_i(v)$ be the successor of v in level i and $\mathsf{pred}_i(v)$ be the predecessor of v in level i. For every ring R and every node v on R, $v_{<R}$ represents all endpoints of edges in v with level less than R and $v_{>R}$ represents all endpoints of edges in v with level higher than R. The parameter r_d and the procedure $\mathsf{Select}()$ used in the algorithms will be determined later.

Next we analyze the performance of the algorithms. Our implementations of $\mathsf{Add}(u)$ and $\mathsf{Remove}(u)$ have the following important property.

Lemma 11.2 *The Hyperring insertion and removal schemes preserve the central conditions for the Hyperring. Furthermore, they make sure that whenever a bridge B is constructed on some i-ring R with parameter d, there is no other bridge on R in the r_d-neighborhood of B.*

Proof. Suppose that before the insertion or deletion of a node the lemma is still true. Then it is easy to check by inspection of the algorithms that also afterwards the lemma is true. $\sqcap$

As we will see, the lemma allows us to prove several important requirements for the Hyperring to be able to achieve a small space, message, and time complexity. These are:

- The *ring distortion* must be small. I.e., for every level i, every ring at level i should have close to $n/2^i$ nodes. This would also make sure that the *degree distortion* is small, i.e. every node has approximately the same degree.
- The *edge distortion* must be small. I.e., for every i-edge e the number of 0-edges bridged by e should be $O(2^i)$.
- The *congestion* must be small. I.e., for any permutation routing problem the expected number of packets traversing a node during the routing should be a $O(\log n)$ for every node.

The tricky aspects in proving these properties are that the nodes do not use any global parameter when performing updates and that updates that were performed at different time steps may have been based on completely different values for parameter d. Nevertheless, we show that all three requirements above can be fulfilled.

11.2 Variants of the basic scheme

For every ring R, $|R|$ denotes the number of edges (or nodes) it consists of, and for every edge e, $|e|$ denotes the number of 0-edges bridged by it. We consider two variants of the basic Hyperring scheme. A deterministic variant and a randomized variant:

- *Deterministic variant:* The neighborhood used for insertions and deletions with parameter d is $r_d = 6(d+3)$, and $\mathsf{Select}(\text{pred}_i(u), \text{succ}_i(u))$ always chooses $\text{pred}_i(u)$.
- *Randomized variant:* The neighborhood used for insertions and deletions with parameter d is $r_d = 15(\log d + 2)$, and $\mathsf{Select}(\text{pred}_i(u), \text{succ}_i(u))$ chooses a node uniformly at random from $\text{pred}_i(u)$ and $\text{succ}_i(u)$.

Suppose for the moment that both the deterministic and the randomized variant fulfill the following properties (with high probability):

Proposition 11.3 *At any time we have:*

1. *for every i-ring R, $|R| \in [\frac{1}{2} \cdot n/2^i - 1, 2 \cdot n/2^i + 1]$ and*
2. *for every i-edge e, $|e| \leq 4 \cdot 2^i$.*

Then this would suffice to bound the message and time complexity of our algorithms.

Message and time complexity

It immediately follows from the algorithms for Add and Remove that Add and Remove need in the worst case $O(\ell \cdot r_d)$ message transmissions and $O(\ell \cdot \log r_d)$ time to add or remove a node, where ℓ is the maximum level of a ring in the Hyperring. Because it only takes $O(\log r_d)$ parallel steps when using edges of the next $\log r_d$ higher levels to execute the insertion or deletion of a node for level i, both variants need in the worst case $O(\ell \cdot \log r_d)$ time steps to insert or delete a node. Since Proposition 11.3 implies that at any time the degree of a node is $O(\log n)$, we get $d = O(\log n)$ and $\ell = O(\log n)$. Hence, we arrive at the following result.

Theorem 11.4 *For the deterministic variant of the basic Hyperring scheme,* Add *and* Remove *achieve a message complexity of* $O(\log^2 n)$ *and time complexity of* $O(\log n \log\log n)$*, and for the randomized variant,* Add *and* Remove *achieve a message complexity of* $O(\log n \log\log n)$ *and a time complexity of* $O(\log n \log\log\log n)$.

Notice that insertions and deletions can be handled in a pipelined fashion if they are at least $\log r_d$ levels apart to avoid interference with other insertions or deletions. Hence, for example, for the deterministic variant every $O((\log\log n)^2)$ steps a new insertion or deletion of a node can be initiated without interference with other insertions or deletions.

Proposition 11.3 will not be shown here because it is quite involved. More information can be found in [1].

11.3 The concurrent Hyperring scheme

Now we show how to perform concurrent updates. As long as updates are sufficiently far apart, our sequential Hyperring scheme already allows to perform these in a concurrent way. However, if the neighborhoods of update operations overlap, then the problem has to be solved in which order to perform the updates. For this a symmetry breaking mechanism is needed. This is usually a difficult task for nodes organized in a ring. Fortunately, our hierarchical ring structure allows to do this quite easily. First, we need a mechanism to assign numbers to rings, and then we will show how to use this to couple node insertions and deletions. The ring numbering we need simply works in a way that every time two new rings are created by some node v, v will assign to one of them the number 0 and the other one the number 1. For the coupling, we assume that a node either has to execute a set of insertion requests or a single deletion request for itself. If some node v has both, then one of the nodes that request to be inserted takes over the role of v.

Coupling

We start with showing how to concurrently execute update operations on the 0-ring. The cental idea here is to provide a coupling of update operations, because when executing pairs of update operations, *we never have to create a new bridge*, and therefore we do not have to explore the neighborhood of an update pair to make sure that a bridge is not created too close to another bridge. For the insertion events, the coupling works as follows:

Initially, every node stores the number of insertion requests bridged by its outgoing 0-edge (which consists of all requests that are managed the node itself). If this is more than two for some node v, this number is reduced to two (which represents the insertion requests with smallest and largest name in v if the nodes that want to join actually have names).

Then, all 1-edges belonging to a 1-ring with number 0 add up the request number of all 0-edges bridged by them. This is continued for all 2-edges, 3-edges, etc., belonging to rings with number 0 until $\lceil \log r_d \rceil + 1$-level edges are reached. We ensure termination without overlaps of edges by the rule that once an $(i+1)$-edge bridges an i-edge with $\lceil \log r_d \rceil + 1 = i$, this $(i+1)$-edge is not considered any more. Suppose the process terminates at some i-edge e. If e has only one request in nodes bridged by it, then this can be executed on the 0-ring without interference with other update operations, because e bridges at least $2r_d$ nodes. If e has more than one request, then these requests are grouped together to as many pairs as possible (i.e. there may be at most one request bridged by e that cannot be given a partner). The pairs are chosen between direct neighbors of the sequence of nodes with requests. For example, if e bridges requests at nodes $v_1, v_2, \ldots, v_{2k}$, where every v_i has one, and this numbering reflects the order in which these nodes are located on the part of the 0-ring bridged by e, then the pairs are (v_{2i-1}, v_{2i}) for all $i \in \{1, \ldots, k\}$. Since executing a request pair never creates a new bridge, the insertions can be performed concurrently.

If some node has at least four insertion requests, then the two insertion requests with the middle names (if names are available; otherwise any two are chosen) are also included in the concurrent insertions. This ensures that even if a node originally has n insertion requests, they can be handled in $\log n$ passes of this algorithm.

For the deletion requests, only one request per 1-edge with number 0 will be accepted to participate in the coupling in a pass to make sure that the 0-ring does not fall apart.

Once the pairs have been created and executed, we continue with handling the update operations for level 2, and afterwards level 3, etc., until we reach a level above $(\log r_d) + 1$. At that point, a new coupling pass can start for level 0. It is not difficult to show that this algorithm allows to obtain the following result.

Theorem 11.5 *Any distribution of k update requests among n nodes can be handled in $O((\log n + \log k)(\log\log n)^2)$ time when using the deterministic variant and $O((\log n + \log k)(\log\log\log n)^2)$ time when using the randomized variant of the concurrent Hyperring scheme.*

Proof. For the deterministic variant, it takes $O(\log\log n)$ parallel steps to handle level 0 and therefore $O((\log\log n)^2)$ steps to handle $O(\log\log n)$ levels. Each time level 0 is executed, the number of remaining update requests decreases by a constant factor, and it takes only $O(\log n)$ steps after the last request has been processed on level 0 to clear up also the other levels. A standard potential function argument can be used to show this. $\sqcap$

11.4 Domain Name Service

Suppose now that we want to establish a name service between peers. Each peer p has a name $\mathsf{Name}(p)$, and the goal is to interconnect the peers in a way so that requests to locate peers with certain names can be done massively in parallel. If new peers arrive and old peers leave in a sequential way, we can use our sequential Hyperring scheme to maintain the Hyperring and the Contact scheme described below to fulfill concurrent search requests.

The only additional requirement we need is that the peers have to be in sorted order according to the names in the 0-ring. However, the insertion scheme Add can easily be extended to this by first routing a new peer to its correct location using Contact and then inserting it according to Add. The routing for a new peer p is done using the following rule:

Suppose that v is the current location of p. As long as $\mathsf{Name}(p) \notin [\mathsf{Name}(v), \mathsf{Name}(\mathsf{succ}_0(v)))$ (in which case p has reached its destination), p moves to the node w with maximum i so that $w = \mathsf{succ}_i(v)$ and $\mathsf{Name}(w) \leq \mathsf{Name}(p)$ (if there is no such w, choose the w among v's successors with maximum name).

Lemma 11.6 *The path length when using the rule above is $O(\log n)$.*

Proof. Recall that in the highest level each ring only consists of a constant number of edges. Hence, p will only traverse a constant number of edges in this level. Once p reaches an i-edge that bridges its destination, it considers only $< i$-edges for the rest of its path. Since an i-edge bridges at most 3 $(i-1)$-edges, p will traverse at most 2 edges in each level (after the highest one), and since there are only $\log n + 2$ levels, p will reach its destination after traversing $O(\log n)$ edges. $\sqcap$

11.5 Routing arbitrary permutations

Once we have a name service (i.e. every node v has a name $\mathsf{Name}(v)$ and the nodes are sorted on the 0-ring according to their names), we can efficiently route arbitrary permutations in the Hyperring. For this we use the following strategy to send a request to some node with name Name:

Contact(Name)**:**
This operation consists of two phases. In a first phase, the request R is sent to a random intermediate destination. This is done by sending it to a random ring in each level, starting with level 0. In level 0, R decides with probability $1/2$ whether to stay at its source node s or to move to its closest predecessor in level 0 that belongs to the 1-ring s does not belong to. Once R reaches the node it wants to reach in level $i-1$, it enters level i. In level i, R decides with probability $1/2$ whether stay at its current node v or to move to its closest predecessor in level i that belongs to the $(i+1)$-ring v does not belong to. Once R reaches the highest possible level, it chooses a node in the ring belonging to this level uniformly at random as its intermediate destination.
For the second phase, notice that the names of the nodes are in sorted order in the 0-ring. Thus, R moves to its destination with the following strategy: Suppose that v is the current location of R. As long as $\mathsf{Name} \notin [\mathsf{Name}(v), \mathsf{Name}(\mathrm{succ}_0(v)))$ (in which case R has reached its destination), R moves to the node w with maximum i so that $w = \mathrm{succ}_i(v)$ and $\mathsf{Name}(w) \leq \mathsf{Name}$ (if there is no such w, choose the w among v's successors with maximum name).

Theorem 11.7 *For every permutation routing problem π, the time required for routing π with the help of* Contact *is $O(\log n)$ with high probability (where the probability only depends on the random choices of the packets when using the deterministic variant of the Hyperring).*

Proof. First we consider phase 1. It is easy to check by induction on the level i that every node v can only receive packets from nodes that are bridged by the i-edge leaving v. Since every i-edge bridges at most $3 \cdot 2^i$ nodes and every packet that is able to reach v at level i does so with probability exactly $1/2^i$, the expected congestion in level i is at most $3 \cdot 2^i \cdot 1/2^i = 3$. Hence, the expected edge congestion of phase 1 is at most 3. Also, since Proposition 11.3 implies that every node has a degree of at least $\log n - 2$, every node will be the intermediate destination of a packet with probability at most $1/2^{\log n - 2}$, which is $O(1/n)$.

Next we consider phase 2. Here, we use the crucial fact that every i-ring R can only receive packets from rings on top of it. Thus, it can only receive packets from its own nodes. Consider now an arbitrary node v in R. It is easy to check that only those packets will be sent to v whose destination is bridged by the edge leaving v. From Proposition 11.3 we know the i-edge e leaving v bridges at most $3 \cdot 2^i$ nodes and that R consists of at most $3 \cdot n/2^i$ nodes. Since we know from the first phase that the probability of a packet with destination bridged by e to choose a node in R as intermediate destination is $O(1/n)$ for each node, the expected edge congestion in level i is bounded by $(3 \cdot 2^i) \cdot (3 \cdot n/2^i) \cdot O(1/n) = O(1)$. Hence, also in the phase 2 the expected edge congestion is bounded by a constant.

Since the random choices made by a packet are independent from other packets, it follows from the Chernoff bounds that the node congestion caused by our permutation routing strategy is $O(\log n)$ in the expected case and also with high probability.

The dilation of the permutation routing strategy is certainly at most $O(\log n)$. Hence, using a slight variant of the growing rank protocol (see Section 3), the runtime of routing a permutation can be bounded by $O(\log n)$ with high probability. $\sqcap$

References

[1] B. Awerbuch and C. Scheideler. Self-repairing and searchable distributed data structures. Dept. of Computer Science, Johns Hopkins University, November 2002.

Add(u):
{v: node executing Add()}
    replace the edge $(v, \text{succ}_0(v))$ by (v, u) and $(u, \text{succ}_0(v))$
    ask u to call Integrate$(0, \deg(v) + 2)$

Integrate(i, d):
{u: node currently executing Integrate(i, d)}
if $\deg(\text{succ}_i(u)) > i$
    if there is a bridge in the $(r_d + 3)$-neighborhood of u in level i
        if u is inside of the bridge
            if u is inside the inner edge of the bridge
                set $v = \text{pred}_i(\text{pred}_i(u))$
                replace the edge $(v, \text{succ}_{i+1}(v))$ by (v, u) and $(u, \text{succ}_{i+1}(v))$
                call Integrate$(i + 1, d)$
            else
                exchange $u^{>R}$ and $\text{succ}_i(u)^{>R}$ between u and $\text{succ}_i(u)$
                set $w = \text{pred}_i(u)$ and $v = \text{succ}_i(u)$
                replace the edge $(w, \text{succ}_{i+1}(w))$ by (w, v) and $(v, \text{succ}_{i+1}(w))$
                call Integrate$(i + 1, d)$ in v
        else {u not inside a bridge}
            set w = node of the bridge closest to u
            permute the endpoints $v^{>R}$ of nodes v between u and w as shown in Figure 3
            if distance between u and w is even
                set $w = \text{pred}_i(u)$ and $v = \text{succ}_i(u)$
                replace the edge $(w, \text{succ}_{i+1}(w))$ by (w, v) and $(v, \text{succ}_{i+1}(w))$
                call Integrate$(i + 1, d)$ in v
            else
                set $v = \text{pred}_i(\text{pred}_i(u))$
                replace the edge $(v, \text{succ}_{i+1}(v))$ by (v, u) and $(u, \text{succ}_{i+1}(v))$
                call Integrate$(i + 1, d)$
    else {no bridge in the neighborhood}
        set w = **Select**$(\text{pred}_i(u), \text{succ}_i(u))$ {specified later}
        if $w = \text{pred}_i(u)$
            replace the edge $(w, \text{succ}_{i+1}(w))$ by (w, u) and $(u, \text{succ}_{i+1}(w))$
        else
            replace the edge $(\text{pred}_{i+1}(w), w)$ by $(\text{pred}_{i+1}(w), u)$ and (u, w)
else
    if the i-edges form a ring of size ≥ 8
        create two intertwined $i + 1$-rings with at most one bridge on top of u

Figure 2: The algorithm for inserting a node.

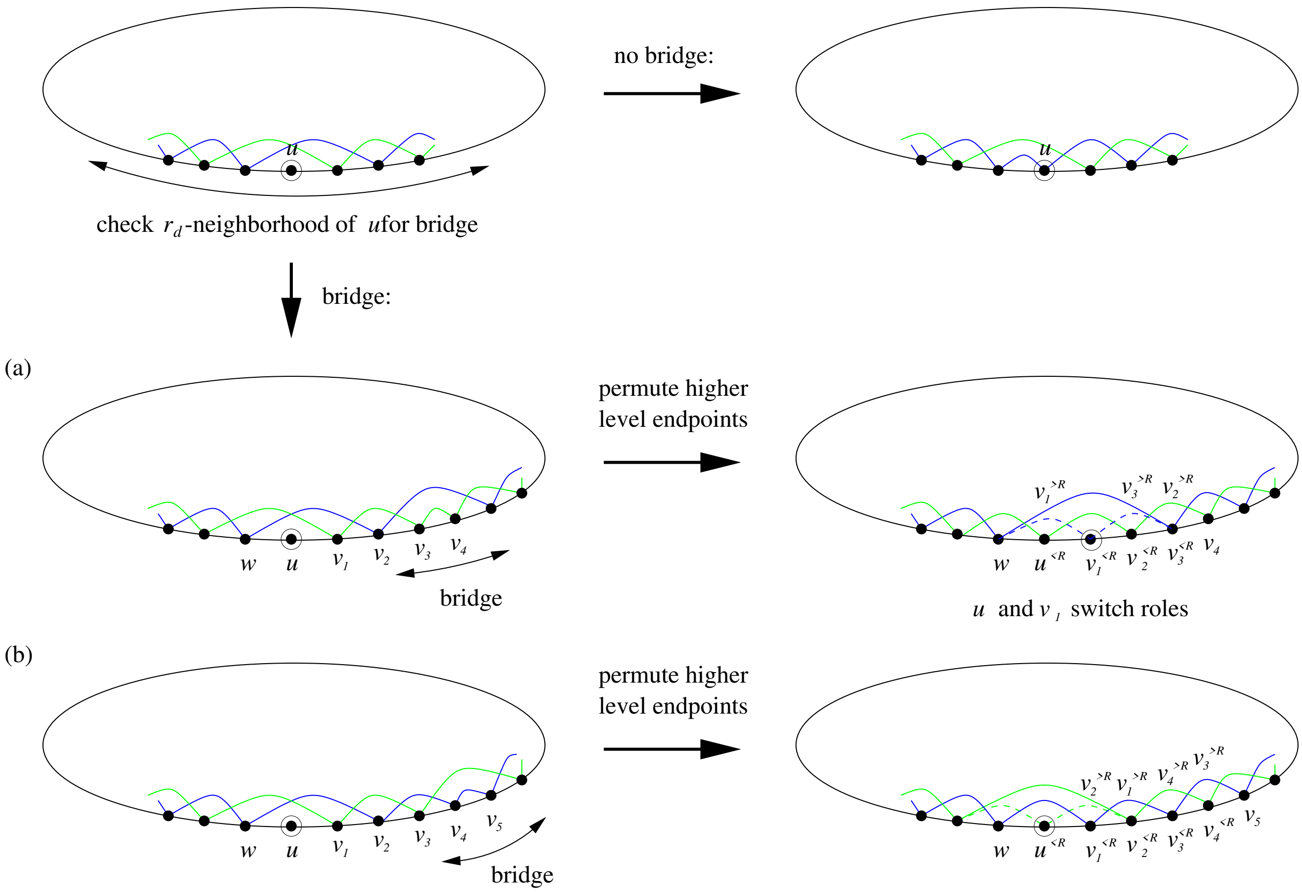

Figure 3: The figure shows the insertion of a node u for ring R. If there is no bridge in the r_d-neighborhood of u (r_d will be determined later), then a bridge will be constructed on top of u and u is continued to be inserted in the next higher ring containing u. If the r_d-neighborhood of u contains a bridge, then the closest of these is moved towards u by permuting the endpoints in $v_{>R}$ for the nodes v between u and the bridge. The outer edge of the bridge is then replaced by the two dashed edges meeting at u. The circled node is continued to be inserted in the next higher ring containing that node. Cases (a) and (b) show the effect of this scheme when the distance of the (closest node of the) bridge to u is even or odd. Notice the crucial property that the insertion scheme only has to continue the insertion for *one* of the two $(i + 1)$-rings on top of R. Otherwise an insertion would be extremely expensive.

Remove(u):
{we only allow a node u to remove itself}
set $d = \deg(u) + 2$
for $i = 0$ **to** $\deg(u) - 1$
  if $i = d - 1$ and the i-edges form a ring of size < 8
    remove the two intertwined d-rings on top of u
  else
    if there is a bridge in the $(r_d + 3)$-neighborhood of u in level i
      set $w =$ node of the bridge closest to u
      permute the pointers of nodes v between u and w as shown in Figure 5
  replace the edges $(\text{pred}_i(u), u)$ and $(u, \text{succ}_i(u))$ by $(\text{pred}_i(u), \text{succ}_i(u))$
if u still has d-edges
  replace the edges $(\text{pred}_d(u), u)$ and $(u, \text{succ}_d(u))$ by $(\text{pred}_d(u), \text{succ}_d(u))$

Figure 4: The algorithm for deleting a node.

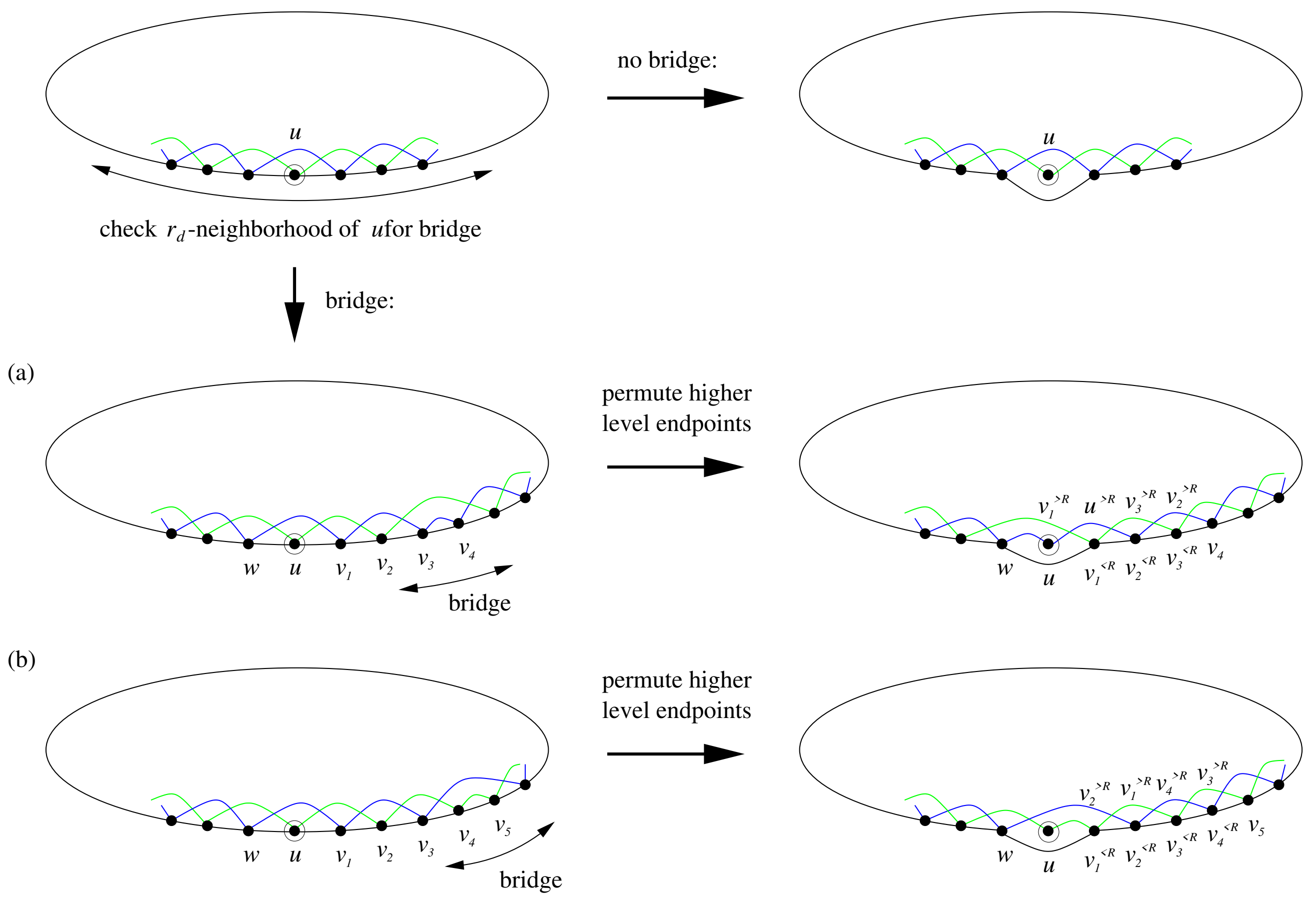

Figure 5: The figure shows the removal of a node u from ring R. If there is no bridge in the r_d-neighborhood of u (r_d will be determined later), then u is simply taken out of R, and u is continued to be deleted from the next higher ring containing u. If the r_d-neighborhood of u contains a bridge, then the closest of these is moved towards u by permuting the endpoints in $v_{>R}$ for the nodes v between u and the bridge. Afterwards, u is taken out of R, and u is continued to be deleted from the next higher ring containing u. Cases (a) and (b) show the effect of this scheme when the distance of the bridge to u is even or odd. Notice that the removal of u from the next higher ring will create a bridge on R for the case that there has not been one in the r_d-neighborhood of R before and otherwise remove the bridge on R that has been moved to u's position. Again, notice the crucial property that the removal scheme only has to continue the removal for *one* of the two $(i+1)$-rings on top of R.