A Distributed and Oblivious Heap*

Christian Scheideler and Stefan Schmid

Institut für Informatik
Technische Universität München, D-85748 Garching, Germany

Abstract. This paper shows how to build and maintain a distributed heap which we call SHELL. In contrast to standard heaps, our heap is oblivious in the sense that its structure only depends on the nodes currently in the network but not on the past. This allows for fast join and leave operations which is desirable in open distributed systems with high levels of churn and frequent faults. In fact, a node fault or departure can be fixed in SHELL in a constant number of communication rounds, which significantly improves the best previous bound for distributed heaps. SHELL has interesting applications. First, we describe a robust distributed information system which is resilient to Sybil attacks *of arbitrary scale*. Second, we show how to organize heterogeneous nodes of *arbitrary* non-uniform capabilities in an overlay network such that the paths between any two nodes do not include nodes of lower capacities. This property is useful, e.g., for streaming. All these features can be achieved without sacrificing scalability: our heap has a de Bruijn like topology with node degree $O(\log^2 n)$ and network diameter $O(\log n)$, n being the total number of nodes in the system.

1 Introduction

In recent years, peer-to-peer systems have received a lot of attention both inside and outside of the research community. Major problems for these systems are how to handle a large churn, adversarial behavior, or participants with highly varying availability and resources. This is particularly the case in open peer-to-peer systems, where any user may join and leave at will. In this paper, we argue that many of these challenges can be solved by organizing the nodes in a distributed heap called *SHELL*.[1] SHELL is *oblivious*, which implies that its structure only depends on the nodes currently in the system but not on the past. It has turned out that this is a crucial property for systems with frequent membership changes as recovery and maintenance is simpler and faster. In fact, in SHELL, a join operation can be handled in $O(\log n)$ time and a leave operation in constant time, which is much better than the $O(\log^2 n)$ runtime bound previously known for scalable distributed heaps [3].

* Partly supported by the DFG-Project SCHE 1592/1-1. For the full version see [15].

[1] The name SHELL is due to the fact that nodes are organized in different layers in our network, where nodes "higher" in the heap can be protected and operate independently of nodes "lower" in the heap.

S. Albers et al. (Eds.): ICALP 2009, Part II, LNCS 5556, pp. 571–582, 2009.

SHELL has a number of interesting applications. As a first case study, we construct a fault-tolerant distributed information system called *i-SHELL* which is resilient to churn and Sybil attacks of *any* size. Sybil attacks are a particularly cumbersome problem in open distributed systems: a user may join the system with a large number of identities in order to, e.g., take over responsibility for an unfair amount of the resources in the system, control or isolate certain parts of the overlay network, or flood the system with futile traffic. The key idea of i-SHELL is that nodes only connect to older nodes in the system so that nodes that were already in the system when the Sybil attack takes place are unaffected by it.

As a second case study, we show that SHELL can also be used to organize nodes with arbitrary capacities in an efficient manner. For example, in a scenario where nodes have non-uniform Internet connections, our *h-SHELL* system guarantees that the paths between two nodes with fast Internet connections only include nodes which are also fast while keeping a low congestion. This is a vital property, e.g., for streaming.

1.1 Model and Definitions

In order to present our key ideas in a clean way, we will use a high-level model for the design and analysis of the SHELL system. We assume that time proceeds in rounds, and all messages generated in round i are delivered in round $i+1$ as long as no node sends and receives more than a polylogarithmic amount of information. In other words, we assume the standard synchronous message-passing model with the restriction that a node can only communicate with nodes that it has currently links to. We do not consider the amount of time needed for internal computation as that will be very small in our case. Each node v in the system is associated with a key $key(v) \in \mathbb{N}$. Our heap will organize the nodes according to these keys. We assume the existence of a symmetry breaker (e.g., unique IP addresses) which allows us to order nodes with the same key so that we can assume w.l.o.g. that all keys are distinct. The *order* n_v of a node v is defined as the number of nodes w in the system with $key(w) < key(v)$. Intuitively, n_v represents the number of nodes that are above v in the heap.

The problem to be solved for the SHELL system is to find efficient and scalable algorithms for the following operations:

1. v.*join()*: Node v joins the system.
2. v.*leave()*: Node v leaves the system.
3. v.*rekey(x)*: Node v's key changes to x.

By “scalability” we mean that these operations can also be executed efficiently when performed *concurrently*.

Scalability is an important feature of SHELL. Messages can be routed fast while the traffic is distributed evenly among nodes. We measure the congestion as follows.

Definition 1 (Congestion). *The* congestion at a node v *is the number of packets forwarded by* v *in a scenario where each of the* n *nodes in the system wants to send a message to a random node.*

One application of SHELL is a distributed information system resilient to Sybil attacks. Formally, we will study the following type of attack.

Definition 2 (Sybil Attack). *Starting with time* t_0*, an attacker joins the network with an arbitrary number of nodes.*

Our goal is to ensure that *all nodes that joined the network before* t_0 are safe against that Sybil attack.

1.2 Our Contributions

The main contribution of this paper is the presentation of a scalable and robust overlay network called SHELL. SHELL is a distributed heap with join and leave operations with asymptotically optimal runtime. In contrast to other distributed (as well as many standard sequential) heaps (e.g., *PAGODA* [3]), SHELL is oblivious, which allows it to react much more rapidly to dynamic changes: nodes can join in time $O(\log n)$ and leave in time $O(1)$. Another highlight of SHELL is its robustness. For example, we are not aware of any other structure which allows us to build a distributed information system resilient to Sybil attacks of arbitrary scale. We also show that SHELL has interesting applications, e.g., it can deal very efficiently with arbitrary variations in the capacities of the nodes. In summary, our distributed heap has the following properties.

1. *Scalability*: Nodes have degree $O(\log^2 n)$ and the network diameter is $O(\log n)$, where n is the network size. Congestion is bounded by $O(\log n)$ on expectation and $O(\log^2 n)$ w.h.p.[2], which is on par with well-known peer-to-peer networks like Chord [19].
2. *Dynamics*: Nodes can be integrated in $O(\log n)$ time and removed in $O(1)$ time.
3. *Robustness*: SHELL can be used to build robust distributed information systems, e.g., a system which is resilient to arbitrarily large Sybil attacks.
4. *Heterogeneity*: SHELL can organize arbitrarily heterogeneous nodes in an efficient way (e.g., for streaming).

1.3 Related Work

A heap is a standard data structure that has been extensively studied in computer science (e.g., [4]). There are several types of concurrent heap implementations such as *skip queues* [16] or *funnel trees* [17]. Moreover, distributed heaps have been used in the context of garbage collection in distributed programs. However, none of these constructions can be used to design scalable distributed systems like those considered in this paper.

[2] By "with high probability" or "w.h.p." we mean a probability of at least $1 - 1/poly(n)$.

A prominent way to build scalable distributed systems are distributed hash tables (or DHTs). Since the seminal works by Plaxton et al. [12] on locality-preserving data management in distributed environments and by Karger et al. [8] on consistent hashing, many DHTs have been proposed and implemented, e.g., Chord [19], CAN [13], Pastry [14], Tapestry [21], or D2B [7]. While these systems are highly scalable, they often assume nodes to be homogeneous, and they are vulnerable to various attacks, especially Sybil attacks that, if large enough, can cause network partitions in these DHTs.

Sybil attacks [6] are an important challenge for open distributed systems. A prominent example is our email system in which tons of spam emails are created by Sybils to evade filtering. A solution to the Sybil attack problem in practice is to have new subscribers solve difficult cryptographic puzzles which limits the rate at which participants can join, or to perform Turing tests to prevent automated subscriptions and to ensure that a new user is indeed a human being. Most of these solutions are based on centralized entities. In fact, a well-known result by Douceur [6] claims that in purely decentralized environments, it is inherently difficult to handle Sybil attacks. Douceur finds that the only means to limit the generation of multiple identities is to have a trusted authority be responsible for generating them. Bazzi et al. propose a Sybil defense based on network coordinates [1,2] in order to differentiate between nodes. Other approaches are based on social networks [5,20] and game theory [9]. All of these approaches are aiming at detecting and/or limiting Sybil attacks. In our paper, we do not aim at preventing Sybil attacks. We assume that an attacker can indeed connect an unbounded number of nodes to the network (by controlling, e.g., a botnet). Nevertheless, SHELL remains efficient at any time for those nodes that have already been in the system before the attack.

As a second application, we demonstrate how SHELL can organize heterogeneous nodes such that stronger nodes can operate independently of weaker nodes. While many peer-to-peer systems assume that nodes have uniform capabilities, there have also been several proposals to construct heterogeneous systems (e.g., [11,18]). These are usually based on multi-tier architectures but can only handle a certain subset of the capacity distributions well. The system closest to ours is PAGODA [3] which also constructs a distributed heap. However, this architecture is not oblivious. The more rigid structure implies that the system is less dynamic and cannot adapt to bandwidth changes nearly as quickly as SHELL. In fact, a join and leave operation take $O(\log^2 n)$ time, and it appears that without major modifications it is not possible to lower the runtime of the operations to something comparable with SHELL.

2 The SHELL Heap

In this section, we present the SHELL heap. Due to space constraints, the proofs were left out and can be found in [15].

2.1 The SHELL Topology

The SHELL topology is based on a dynamic de Bruijn graph and builds upon the continuous-discrete approach introduced by Naor and Wieder [10]. In the classical d-dimensional de Bruijn graph, $\{0,1\}^d$ represents the set of nodes and two nodes $x, y \in \{0,1\}^d$ are connected by an edge if and only if there is a bit $b \in \{0,1\}$ so that $x = (x_1 \ldots x_d)$ and $y = (b\, x_1 \ldots x_{d-1})$ (i.e., y is the result of a right shift of the bits in x with the highest bit position taken by b) or $y = (x_2 \ldots x_d b)$. When viewing every node $x \in \{0,1\}^d$ as a point $\sum_{i=1}^{d} x_i/2^i \in [0,1)$ and letting $d \to \infty$, then the node set of the de Bruijn graph is equal to $[0,1)$ and two points $x, y \in [0,1)$ are connected by an edge if and only if $x = y/2$, $x = (1+y)/2$ or $x = 2y \pmod 1$. This motivates the dynamic variant of the de Bruijn graph described in the following.

For any $i \in \mathbb{N}_0$, a *level i interval* (or simply *i-interval*) is an interval of size $1/2^i$ starting at an integer multiple of $1/2^i$ in $[0,1)$. The *buddy* of an i-interval I is the other half of the $(i-1)$-interval that contains I. We assume that every node in the system is assigned to some fixed (pseudo-)random point in $[0,1)$ (the node set of the continuous de Bruijn graph above) when it joins the system. We also call this point its *identity* or *id*. For now, suppose that every node v knows its order n_v. Later in this section we present a local control strategy that allows the nodes to obtain a good estimate on n_v. We want to maintain the following condition at any point in time for some fixed and sufficiently large constant $c > 1$.

Condition 21. *Each node v has* forward edges *to all nodes w with key$(w) <$ key(v) in the following three intervals:*

- *the $\ell_{v,0}$-interval containing v (v's* home interval*) and its buddy,*
- *the $\ell_{v,1}$-interval containing $v/2$ and the $\ell_{v,2}$-interval containing $(1+v)/2$ (v's* de Bruijn intervals*) and their buddies.*

v also has backward edges *to all nodes that have forward edges to it.*

The level $\ell_{v,0} \in \mathbb{N}_0$ of v is chosen as the largest value such that the $\ell_{v,0}$-interval containing v contains at least $c \log n_v$ nodes w with key$(w) <$ key(v) for some fixed and sufficiently large constant c. If there is no such $\ell_{v,0}$ (i.e., n_v is very small), then $\ell_{v,0}$ is set to 0. The same rule is used for the selection of the levels $\ell_{v,i}$, $i \in \{1,2\}$, using the points $(i+v-1)/2$ instead of v.

The conditions on $\ell_{v,i}$ suffice for our operations to work w.h.p. If we want guarantees, one would have to extend the definition of $\ell_{v,i}$ to lower bounds on the number of nodes in both halves of the $\ell_{v,i}$-interval as well as its buddy, but for simplicity we will not require that here.

The forward edges are related to the upward edges in a standard (min-)heap while the backward edges are related to the downward edges. However, instead of a tree-like structure, we have a de Bruijn-like structure among the nodes. Forward edges to the home interval of a node are called *home edges* and edges to the de Bruijn intervals *de Bruijn edges*. Our construction directly yields the following properties.

Fact 22 (Oblivious Structure). *The SHELL topology only depends on the current set of nodes and their keys but not on the past.*

Fact 23 (Forward Independence). *The forward edges of a node v only depend on nodes u with $key(u) < key(v)$.*

Recall that every node is given a (pseudo-)random point in $[0, 1)$. Then the following property also holds.

Lemma 1 (Level Quasi-Monotonicity). *For any pair of nodes v and w with $key(v) > key(w)$ it holds that $\ell_{v,i} \geq \ell_{w,j} - 1$ for any $i, j \in \{0, 1, 2\}$, w.h.p.*

In this lemma and the rest of the paper, "w.h.p." means with probability at least $1 - 1/poly(n_v)$. Next we bound the degree of the nodes. From the topological conditions we can immediately derive the following property.

Lemma 2. *Every node v has $\Theta(c \log n_v)$ many forward edges to every one of its intervals, w.h.p.*

For the backward edges, we have the following bound, where n is the total number of nodes in the system.

Lemma 3. *The maximal number of backward edges of a node is limited by $O(c \log^2 n)$ w.h.p.*

2.2 Routing

We now present a routing algorithm on top of the described topology. For any pair (u, v) of nodes, the operation $route(u, v)$ routes a message from node u to node v. The routing operation consists of two phases: first, a $forward(v)$ operation is invoked which routes a message from node u to some node w with $key(w) < key(u)$ whose home interval contains v. Subsequently, if necessary (i.e., if $w \neq v$), a $refine(v)$ operation performs a descent or ascent along the levels until (the level of) node v is reached. In the following, we will show how to implement the $route(u, v)$ operation in such a way that a message is only routed along nodes w for which it holds that $key(w) \leq \max\{key(u), key(v)\}$.

Forward(v). We first consider the $forward(v)$ algorithm, where node u sends a message along forward edges to a node whose home interval includes node v. Let $(x_1, x_2, x_3, \ldots)$ be the binary representation of u and $(y_1, y_2, y_3, \ldots)$ be the binary representation of v (i.e., $u = \sum_{i \geq 1} x_i/2^i$). Focus on the first $k = \log n_u$ bits of these representations. Ideally, we would like to send the message along the points $z_0 = (x_1, x_2, x_3, \ldots)$, $z_1 = (y_k, x_1, x_2, x_3, \ldots)$, $z_2 = (y_{k-1}, y_k, x_1, x_2, x_3, \ldots)$, $\ldots, z_k = (y_1, \ldots, y_k, x_1, x_2, x_3, \ldots)$. We emulate that in SHELL by first sending the message from node u along a forward edge to a node u_1 with largest key whose home interval contains z_1. u can indeed identify such a node since $z_1 = z_0/2$ or $z_1 = (1 + z_0)/2$, i.e., z_1 is contained in one of u's de Bruijn intervals, say, I. Furthermore, u has $\Theta(c \log n_u)$ forward edges to each of the two halves of its intervals, w.h.p., and from Lemma 1 it follows that every node w that u has a forward edge to, $\ell_{w,0} \leq \ell_{u,i} + 1$, i.e., w's home interval has at least half the size of I. Hence, u has a forward connection to a node u_1 whose home interval contains z_1 w.h.p. From u_1, we forward the message along a forward edge to a node u_2 with largest key whose home interval contains z_2. Again, u_1 can identify such a node since $z_2 = z_1/2$ or $z_2 = (1 + z_1)/2$ and z_1 belongs to the home interval of u_1, which implies that z_2 belongs to one of the de Bruijn intervals of u_1. We continue that way until a node u_k is reached whose home interval contains z_k, as desired. Observe that according to Lemma 1, u_k contains v in its home interval as the first k bits of u_k and v match and $\ell_{u_k,0} < k$, w.h.p., so the forward operation can terminate at u_k. We summarize the central properties of the forward phase in three lemmas. The first lemma bounds the dilation.

Lemma 4. *For any starting node u and any destination v, forward(v) has a dilation of $\log n_u$, w.h.p.*

The next lemma is crucial for the routing to be scalable.

Lemma 5. *In the forward phase, a packet issued by a node u of order n_u will terminate at a node of order at least $n_u/2$ w.h.p.*

As a consequence, we can bound the congestion.

Lemma 6. *For a random routing problem, the congestion at any node v is $O(\log n_v)$ on expectation and $O(\log^2 n_v)$ w.h.p.*

Refine(v). Recall that once the *forward(v)* operation terminates, the packet has been sent to a node w that contains the location of v in its home interval. In a second *refining* phase *refine(v)*, the packet is forwarded to the level of v in order to deliver it to v. First, suppose that the packet reaches a node w with $key(w) > key(v)$. According to Condition 21, w has forward connections to all nodes x in its home interval with $key(x) < key(w)$. Hence, w has a forward edge to v and can therefore directly send the packet to v.

So suppose that the packet reached a node w with $key(w) < key(v)$. In this case, w may not be directly connected to v since there will only be a forward edge from v to w (and therefore a backward edge from w to v) if w is in v's home interval, which might be much smaller than w's home interval. Therefore, the packet has to be sent downwards level by level until node v is reached. Suppose that w is at level ℓ in its home interval. We distinguish between two cases.

Case 1: $n_v \leq (3/2)n_w$. Then v and w can be at most one level apart w.h.p.: Since the interval size of w can be at most $2(1+\delta)c \log n_w/n_w$ for some constant $\delta > 0$ (that can be made arbitrarily small depending on c) w.h.p., two levels downwards there can be at most $(1+\delta)^2 c \log n_w/2$ nodes left in a home interval of that level that w has forward edges to, w.h.p. Moreover, there can be at most an additional $(1+\delta)(n_w/2)(1+\delta)c \log n_w/(2n_w) = (1+\delta)^2 c \log n_w/4$ nodes that v has forward edges to, which implies that the level of v must be larger than $\ell + 2$ w.h.p. Thus, w is either in v's home interval or its buddy, which implies that v has a forward edge to w (resp. w has a backward edge to v), so w can deliver the packet directly to v.

Case 2: $n_v > (3/2)n_w$. Then there must be at least one node x with $key(w) \leq key(x) < key(v)$ that is in the $\ell+1$-interval containing v (which might be w itself) w.h.p. Take the node with largest such key. This node must satisfy $n_x \leq (3/2)n_w$ w.h.p., which implies that it is at level ℓ or $\ell+1$ by Case 1, so w has a backward edge to that node and therefore can send the packet to it. The forwarding of the packet from x is continued in the same way as for w so that it reaches node v in at most $\log n_v$ hops.

For the refine operation, the following lemma holds.

Lemma 7. *For any starting node w and any node v, the refine(v) operation has a dilation of $O(\log n_v)$. Furthermore, the congestion at any node u is at most $O(c \log n_u)$ w.h.p.*

2.3 Join and Leave

Open distributed systems need to incorporate mechanisms for nodes to join and leave. Through these membership changes, the size of the network can vary significantly over time. A highlight of SHELL is its flexibility which facilitates very fast joins and leaves.

Join. We first describe the join operation. Recall that each node v is assigned to a (pseudo-)random point in $[0, 1)$ when it joins the system. For the bootstrap problem, we assume that node v knows an arbitrary existing node u in the system which can be contacted initially. Then the following operations have to be performed for $x \in \{v, v/2, (1+v)/2\}$:

1. *forward(x)*: Route v's join request along forward edges to a node w with $key(w) \leq key(u)$ whose home interval contains x.
2. *refine(x)*: Route v's join request along forward or backward edges to a node w' with *maximum key* $\leq key(v)$ that contains x in its home interval.
3. *integrate(x)*: Copy the forward edges that w' has in its home interval and buddy to v (and check Condition 21 for the involved nodes).

Here, we use a slight extension of the refine operation proposed for routing. If $key(v) > key(w)$, there is no change compared to the old refine operation. However, if $key(v) < key(w)$, then we have to send v's join request upwards along the levels till it reaches a node w' with maximum key $\leq key(v)$ that contains v in its home interval. This is necessary because w may not have a forward edge to w'.

Observe that a membership change can trigger other nodes to upgrade or downgrade. To capture these effects formally, we define the *update cost* as follows.

Definition 3 (Update Cost). *The total number of links which need to be changed after a given operation (e.g., a single membership change) is referred to as the* update cost *induced by the operation.*

Theorem 24. *A join operation terminates in time $O(\log n)$ w.h.p. The update cost of a join operation is bounded by $O(c \log^2 n)$ w.h.p.*

Leave. If a node v leaves in SHELL, it suffices for its neighbors to drop v from their neighbor list, which can be done in constant time. Some of the nodes may then have to upgrade to a higher level, which also takes constant time w.h.p. as it suffices for a node to learn about its 3-hop neighborhood w.h.p. (due to the use of buddy intervals). Thus, we have the following result.

Theorem 25. *The leave operation takes a constant number of communication rounds w.h.p. Moreover, the update cost induced at other nodes (cf Definition 3) is bounded by* $O(c \log^2 n)$ *w.h.p.*

2.4 Rekey

There are applications where node keys are not static and change over time. For instance, in the heterogeneous peer-to-peer system described in Section 3, the available bandwidth at a node can decrease or increase dynamically. Our distributed heap takes this into account and allows for frequent rekey operations. Observe that we can regard a rekey operation as a leave operation which is immediately followed by a join operation at the same location in the ID space but maybe on a different partition level. While a node can downgrade in constant time, decrease key operations require collecting additional contact information, which takes logarithmic time. From our analysis of join and leave operations, the following corollary results.

Corollary 26. *In SHELL, a node can perform a rekey operation in time* $O(\log n)$*, where* n *is the total number of nodes currently in the system. The update cost induced at other nodes (cf Definition 3) is at most* $O(c \log^2 n)$*.*

2.5 Estimation of the Order

So far, we have assumed that nodes know their order in order to determine their level. Of course, an exact computation takes time and may even be impossible in dynamic environments. In the following, we will show that sufficiently good approximations of the correct partition level i can be obtained by *sampling*.

In order to find the best home interval, adjacent intervals, and de Bruijn interval sizes, a node v counts the number $B(j)$ of nodes in a given j-level interval it observes. Ideally, the smallest j with the property that the home interval contains at least $c \log n_v$ nodes of lower keys defines the forward edges. We now prove that if decisions are made with respect to these $B(j)$, errors are small in the sense that the estimated level is not far from the ideal level i.

Concretely, at join time, nodes do binary search to determine the level i according to the following rule: if $j > B(j)/c - \log B(j)$ then level j is increased, and otherwise, if $j < B(j)/c - \log B(j)$, j is decreased, until (in the ideal case) a level i with $i = B(i)/c - \log B(i)$ is found (or the level i closest to that).

The following lemma shows that this process converges and that the search algorithm efficiently determines a level which is at most one level off the ideal level with high probability.

Theorem 27. *Let* $\widehat{i}$ *be the level chosen by the sampling method and let* i *be the ideal level. It holds that* $|\widehat{i} - i| \leq 1$ *w.h.p.*

3 Applications

A distributed and oblivious heap structure turns out to be very useful in various application domains. In the following, we sketch two applications. For more information, we refer the reader to the technical report [15]. We first describe a fault-tolerant information system called *i-SHELL* which is resilient to Sybil attacks of arbitrary scale. Second, we show how our heap can be used to build a heterogeneous peer-to-peer network called *h-SHELL*.

3.1 i-SHELL

In order to obtain a robust distributed information system, we order the nodes with respect to their *join times* (i-order). Concretely, $key(v)$ is equal to the time step when v joined the system. For the bootstrap problem, we assume the existence of a *network entry point* assigning time-stamps to the nodes in a verifiable way. Recall that for this choice of the keys the forward connections of a node only depend on older nodes. Moreover, whenever two nodes u and v want to exchange messages, our routing protocol makes sure that these messages are only forwarded along nodes w that are at least as old as u or v. This has the following nice properties:

Churn. Suppose that there are some nodes frequently joining and leaving the system. Then the more reliable nodes can decide to reject re-establishing forward edges to such a node each time the node is back up, forcing it to obtain a new join time stamp from the entry point so that it can connect back to the system. In this way, the unreliable nodes are forced to the bottom of the SHELL system so that communication between reliable nodes (higher up in SHELL) will not be affected by them.

Sybil Attacks. Suppose that at some time t_0 the adversary enters the system with a huge number of Sybil nodes. Then any two nodes u and v that joined the SHELL system before t_0 can still communicate without being affected by the Sybils. The Sybils may try to create a huge number of backward edges to the honest nodes, but an honest node can easily counter that by only keeping backward edges to the T oldest nodes, for some sufficiently large threshold T. Moreover, Sybils could try to overwhelm the honest nodes with traffic. But also here the honest nodes can easily counter that by preferentially filtering out packets from the youngest nodes in case of congestion, so that packets from nodes that joined the system before t_0 can still be served in a timely manner.

Putting it together. We can combine SHELL with consistent hashing in order to convert it into a DHT that is robust to the Sybil attacks described above. One can show the following result:

Theorem 31. *For the nodes injected before t_0, insert, delete and find operations have a runtime of $O(\log n)$, where n is the number of nodes currently in the system, and their congestion is bounded by $O(\log^2 n)$, irrespective of a Sybil attack taking place after t_0.*

3.2 h-SHELL

As a second example, we sketch how to use our heap structure to build a peer-to-peer overlay called h-SHELL. h-SHELL takes into account that nodes can have heterogeneous bandwidths. The system can be used, e.g., for streaming. In h-SHELL, $key(v)$ is defined as the inverse of the bandwidth of v, i.e., the higher its bandwidth, the lower its key and therefore the higher its place in h-SHELL. Nodes may propose a certain bandwidth, or (in order to avoid churn) its neighbors in h-SHELL are measuring its bandwidth over a certain time period and propose an average bandwidth value for a node v that may be used for its key. When the bandwidth at a node changes, a fast rekey operation will reestablish the heap condition.

From the description of the SHELL topology it follows that whenever two nodes u and v communicate with each other, only nodes w with a bandwidth that is at least the bandwidth of u or v are used for that. Thus, in the absence of other traffic, the rate at which u and v can exchange information is essentially limited by the one with the smaller bandwidth. But even for arbitrary traffic patterns h-SHELL has a good performance. Using Valiant's random intermediate destination trick, the following property can be shown using the analytical techniques in PAGODA [3].

Theorem 32. *For any communication pattern, the congestion in h-SHELL in order to serve it is at most by a factor of $O(\log^2 n)$ higher w.h.p. compared to a best possible network of bounded degree for that communication pattern.*

4 Conclusion

The runtime bounds obtained for SHELL are optimal in the sense that scalable distributed heaps cannot be maintained at asymptotic lower cost. In future research, it would be interesting to study the average case performance and robustness of SHELL "in the wild".

References

1. Bazzi, R., Choi, Y., Gouda, M.: Hop Chains: Secure Routing and the Establishment of Distinct Identities. In: Proc. 10th Intl. Conf. on Principles of Distributed Systems, pp. 365–379 (2006)
2. Bazzi, R., Konjevod, G.: On the Establishment of Distinct Identities in Overlay Networks. In: Proc. 24th Symp. on Principles of Distributed Computing (PODC), pp. 312–320 (2005)
3. Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C., Thober, M.: Pagoda: A Dynamic Overlay Network for Routing, Data Management, and Multicasting. In: Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 170–179 (2004)
4. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (2001)

5. Danezis, G., Lesniewski-Laas, C., Kaashoek, F., Anderson, R.: Sybil-resistant DHT Routing. In: Proc. 10th European Symp. on Research in Computer Security, pp. 305–318 (2005)
6. Douceur, J.R.: The Sybil Attack. In: Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS), pp. 251–260 (2002)
7. Fraigniaud, P., Gauron, P.: D2B: A de Bruijn Based Content-Addressable Network. Elsevier Theoretical Computer Science 355(1) (2006)
8. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In: Proc. 29th ACM Symposium on Theory of Computing (STOC), pp. 654–663 (1997)
9. Margolin, N., Levine, B.: Informant: Detecting Sybils Using Incentives. In: Proc. 11th Intl. Conf. on Financial Cryptography and Data Security, pp. 192–207 (2007)
10. Naor, M., Wieder, U.: Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In: Proc. 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 50–59 (2003)
11. Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Löser, A.: Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-to-Peer Networks. In: Proc. 12th International Conference on World Wide Web (WWW), pp. 536–543 (2003)
12. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In: Proc. 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 311–320 (1997)
13. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A Scalable Content-Addressable Network. In: Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 161–172 (2001)
14. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
15. Scheideler, C., Schmid, S.: A Distributed and Oblivious Heap. In: Technical University Munich, Tech Report TUM-I0906 (2009)
16. Shavit, N., Zemach, A.: Scalable Concurrent Priority Queue Algorithms. In: Proc. 18th Annual ACM Symposium on Principals of Distributed Computing (PODC), pp. 113–122 (1999)
17. Shavit, N., Zemach, A.: Combining Funnels: A Dynamic Approach to Software Combining. Journal of Parallel and Distributed Computing 60 (2000)
18. Srivatsa, M., Gedik, B., Liu, L.: Large Scaling Unstructured Peer-to-Peer Networks with Heterogeneity-Aware Topology and Routing. IEEE Trans. Parallel Distrib. Syst. 17(11), 1277–1293 (2006)
19. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (2001)
20. Yu, H., Kaminsky, M., Gibbons, P., Flaxman, A.: SybilGuard: Defending Against Sybil Attacks via Social Networks. In: Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (2006)
21. Zhao, B., Kubiatowicz, J.D., Joseph, A.: Tapestry: An Infrastructure for Fault-Tolerant Widearea Location and Routing. Technical report, UC Berkeley, Computer Science Division Tecnical Report UCB/CSD-01-1141 (2001)