HSkip+: A Self-Stabilizing Overlay Network for Nodes with Heterogeneous Bandwidths

Matthias Feldotto
Heinz Nixdorf Institute &
Department of Computer Science
University of Paderborn, Germany
Email: feldi@mail.upb.de

Christian Scheideler
Theory of Distributed Systems Group
Department of Computer Science
University of Paderborn, Germany
Email: scheideler@upb.de

Kalman Graffi
Technology of Social Networks Group
Department of Computer Science
University of Düsseldorf, Germany
Email: graffi@cs.uni-duesseldorf.de

***Abstract*—In this paper we present and analyze HSkip+, a self-stabilizing overlay network for nodes with arbitrary heterogeneous bandwidths. HSkip+ has the same topology as the Skip+ graph proposed by Jacob et al. [1] but its self-stabilization mechanism significantly outperforms the self-stabilization mechanism proposed for Skip+. Also, the nodes are now ordered according to their bandwidths and not according to their identifiers. Various other solutions have already been proposed for overlay networks with heterogeneous bandwidths, but they are not self-stabilizing. In addition to HSkip+ being self-stabilizing, its performance is on par with the best previous bounds on the time and work for joining or leaving a network of peers of logarithmic diameter and degree and arbitrary bandwidths. Also, the dilation and congestion for routing messages is on par with the best previous bounds for such networks, so that HSkip+ combines the advantages of both worlds. Our theoretical investigations are backed by simulations demonstrating that HSkip+ is indeed performing much better than Skip+ and working correctly under high churn rates.**

I. Introduction

Peer-to-peer systems have become very popular for a variety of reasons. For example, the fact that peer-to-peer systems do not need a central server means that individuals can search for information or cooperate without fees or an investment in additional high-performance hardware. Also, peer-to-peer systems permit the sharing of resources (such as computation and storage) that otherwise sit idle on individual computers. However, the absence of any trusted anchor like a central server also has its disadvantages since churn and adversarial behavior has to be managed by the peers without outside help. One promising approach that has been investigated in recent years is to use topological self-stabilization, i.e., the overlay network of the peer-to-peer system can recover its topology from any state, as long as it is initially weakly connected. Various topologies have already been considered, but so far mostly the case has been studied that the peers have the same resources (concerning speed, storage, and bandwidth) whereas in reality the available resources may differ significantly from peer to peer. An exception is a self-stabilizing system for peers with non-uniform storage [2], but no self-stabilizing peer-to-peer system for peers with non-uniform bandwidth has been proposed yet. Due to the current development, especially with mobile devices, this property becomes more important and is often the bottleneck in the system. This paper is the first to propose a system which considers the bandwidth in its design.

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

A. *Our model*

1) Network model: Similar to Nor et al. [3], we assume that we have a (potentially dynamic) set V of n nodes with unique identifiers. Each node v maintains a set of variables (determined by the protocol) that define the *state* of node v. For each pair of nodes u and v we have a channel $C_{u,v}$ that holds messages that are currently in transit from u to v. We assume that the channel capacity is unbounded, there is no message loss, and the messages are delivered asynchronously to v in FIFO order. The sequence of all messages stored in $C_{u,v}$ constitutes the *state* of $C_{u,v}$.

Whenever a node u stores a reference of node v, we consider that as an *explicit edge* (u, v), and whenever a channel $C_{u,v}$ holds a message with a reference of node w, we consider that as an *implicit edge* (v, w). Node references are assumed to be atomic and read-only, i.e., they cannot be split, encoded, or altered. They can only be deleted or copied to produce new references that can be sent to other nodes. If there is a reference of a node that is not in the system any more, we assume that this can be detected by the nodes, so that without loss of generality we can assume that only references of nodes that are still in the system are present in the nodes and the channels. Whenever a message in $C_{u,v}$ contains a node reference w, it also contains w's bandwidth and identifier so that this information can be corrected in v if needed (though it might initially be wrong). Only point-to-point communication is possible, and the nodes can only send messages along explicit edges (since they are not yet aware of the endpoint of an implicit edge). Whenever node u sends a message along an explicit edge (u, v), it is transferred to $C_{u,v}$. Let E_e denote the set of all explicit edges and E_i denote the set of all implicit edges. The overlay network formed by the system is defined as a directed graph $G = (V, E)$ with $E = E_e \cup E_i$. With $G_e = (V, E_e)$ respectively $G_i = (V, E_i)$ we define the network which only consists of the explicit respectively implicit edges. The degree of a node v in G is equal to its degree in G_e (i.e., the number of explicit edges of the form (v, w)), and the diameter of G is equal to the diameter of G_e. We define the specific state of the network at a time t with $G(t), E(t), E_e(t)$ and $E_i(t)$.

2) Computational model: A *program* is composed of a set of variables and actions. An *action* has the form $\langle label \rangle$:

$\langle guard \rangle \rightarrow \langle command \rangle$. $label$ is a name to differentiate between actions, $guard$ can be an arbitrary predicate based on the state of the node executing the action, and $command$ represents a sequence of commands. A $guard$ of the form $received(m)$ is true whenever a message m has been received (and not yet processed) by the corresponding node. An action is *enabled* if its guard is true and otherwise *disabled*.

The *node state* of the system is the combination of all node states, and the *channel state* of the system is the combination of all channel states. Both states together form the *(program) state* of the system. A *computation* is an infinite fair sequence of states such that for each state s_t, the next state s_{t+1} is obtained by executing the commands of an action that is enabled in s_t. So for simplicity we assume that only one action can be executed at a time, but our results would also hold for the distributed scheduler, i.e., only one action can be executed *per node* at a time. We assume two kinds of fairness of computation: weak fairness for action execution and fair message receipt. *Weak fairness* of action execution means that no action will be enabled without being executed for an infinite number of states (i.e., no action will starve, and actions that are enabled for an infinite number of states will be executed infinitely often). *Fair message receipt* means that every message will eventually be received (and therefore processed due to weak fairness).

3) Topological self-stabilization: Next we define topological self-stabilization, which goes back to the idea by Dijkstra [4] and is summarized by Schneider [5]. In the topological self-stabilization problem we start with an arbitrary state (with a finite number of nodes, channels, and messages) in which G is weakly connected, and a *legal state* is any state where the topology of G_e has the desired form and the information that the nodes have about their neighbors is correct. We assume without loss of generality that G is initially weakly connected, because if not, then we would just focus on any of the weakly connected components of G and would prove topological self-stabilization for that component. In order to show topological self-stabilization, two properties need to be shown:

Definition 1 (Convergence)**.** For any initial state in which G is weakly connected, the system eventually reaches a legal state.

Definition 2 (Closure)**.** Whenever the system is in a legal state, then it is guaranteed to stay in a legal state, provided that no faults or changes in the node properties (in our case, the bandwidth) happen.

B. Related work

Topological self-stabilization has recently attracted a lot of attention. Various topologies have been considered such as simple line and ring networks (e.g., [6], [7]), skip lists and skip graphs (e.g., [3], [1]), expanders [8], the Delaunay graph [9], the hypertree [10], and Chord [11]. Also a universal protocol for topological self-stabilization has been proposed [12]. However, none of these works consider nodes with heterogeneous bandwidths.

Various network topologies have been suggested to interconnect nodes with heterogeneous bandwidths. While [13], [14] do not provide any formal guarantees and just evaluate their constructions via experiments, [15], [16] give formal guarantees, but (like for the experimental papers) no self-stabilizing protocol has been proposed for these networks. There has also been extensive work on networks with heterogeneous bandwidths in the context of streaming applications (see, e.g., [17] and the references therein) but the focus is more on coding schemes and optimization problems, so it does not fit into our context.

Probabilistic approaches like ours have the advantage of better graph properties (e.g., a logarithmic expansion) compared to deterministic variants (e.g., [18]).

C. Our Contribution

We modified the protocol proposed for the self-stabilizing Skip+ graph [1] to organize the nodes in a more effective way using the same topology. However, the nodes are not ordered according to their labels but according to their bandwidths. Due to this we call our graph the HSkip+ graph. Improvements of our construction over previous work are:

- We prove self-stabilization under the asynchronous message passing model whereas in [1] it was only shown for the synchronous message passing model.
- In simulations, our Skip+ protocol has basically the same self-stabilization time as the original Skip+ protocol, but it spends significantly less work (in terms of messages that are exchanged between the nodes) in order to reach a legal state. Furthermore, our overlay is working correctly under a churn rate of nearly 50%.
- When a node joins or leaves in a legal state, then the worst case work in [1] is $O(\log^4 n)$ w.h.p. whereas our protocol just needs a worst case work of $O(\log^2 n)$ w.h.p., and a worst-case time of $O(\log n)$ w.h.p. in the synchronous message passing model, to get back to a legal state. The work and time bounds are on par with the previously best (non-self-stabilizing) network of logarithmic diameter and degree for peers with heterogeneous bandwidths [16].
- Also the competitiveness concerning the congestion of arbitrary routing problems in HSkip+ is on par with the previously best (non-self-stabilizing) network of logarithmic diameter and degree for peers with heterogeneous bandwidths [15].

Hence, our HSkip+ construction combines the best results of both worlds (self-stabilizing networks and scalable networks for heterogeneous nodes).

D. Organization of the Paper

This paper is structured as follows: In Section II we present our topology and the associated self-stabilizing algorithm. We show its convergence and closure. Furthermore, we look at the handling of external dynamics and routing in our network. Section III presents our simulation results, especially the comparison of Skip+ and HSkip+. Finally, we end the paper in Section IV with a conclusion. A long version of this paper with additional proofs can be found in [19].

II. Theoretical Analysis

A. HSkip+ Topology

We now present the desired topology for our problem which we call *HSkip+*. It is the same as the *Skip+* topology introduced by Jacob et al. [1], which is based on skip graphs [20], but the ordering of the nodes is different. Instead of using fixed node labels for the ordering, the bandwidth values are used. Also, new rules are used since they turned out to consume clearly less work than the rules proposed for *Skip+*.

As stated in our network model (cf. Sec. I-A1), the system forms a directed graph $G = (V, E)$. Each node $v \in V$ has several internal variables which define the internal state of the node v:

- $v.id$ is the unique, immutable identifier of node v.
- $v.rs$ is an immutable pseudo-random bit string of node v.
- $v.bw$ is the current bandwidth of node v, which is modifiable during the execution. W.l.o.g., we assume that all bandwidths are unique (which is easy to achieve given that the node identifiers are unique).
- $v.nh$ is the neighborhood of node v, i.e., the set of all nodes whose references are stored in v.

We introduce some auxiliary functions for the internal variables, and especially for the bit string $v.rs$:

- $\text{prefix}(v, i) = v.rs[0 \ldots i-1]$ is the prefix of length i of $v.rs$.
- $\text{commonPrefix}(v, w) = \text{argmax}_i\{\text{prefix}(v, i) = \text{prefix}(w, i)\}$ is the length of the maximal common prefix of $v.rs$ and $w.rs$.
- $\text{level}(v) = \max_{w \in v.nh}\{\text{commonPrefix}(v, w)\}$ is called the current level of node v and will be used later for the grouping of the nodes.
- $\deg(v) = |v.nh|$ is called the current degree of node v, the number of neighbors.

As mentioned before, we are aiming at maintaining the HSkip+ graph among the nodes. For the HSkip+ graph we need a series of definitions.

Definition 3 (Component of HSkip+)**.** Two nodes v and w belong to the same *component* at level i of HSkip+ if their bit values $v.rs$ and $w.rs$ share the same prefix of length i, formally $\text{component}(v, i) = \{w \in V | \text{commonPrefix}(v, w) \geq i\}$. The nodes in each component are ordered according to their bandwidths. A component is called *trivial* if there is only one node in the component.

Components exist at each level as long as they are non-trivial, which means there would be only one node in a component. Therefore, we can define the level of HSkip+ as the number of levels needed to represent all non-trivial components:

Definition 4 (Level of HSkip+)**.** The *level* of the network $G = (V, E)$ is defined by the maximum level i such that two nodes share a common prefix of length i. Formally, $\text{level}(G) = \max_{v,w \in V, v \neq w} \text{commonPrefix}(v, w)$.

In addition to the regular linked list for each component we have further edges to get a more stable neighborhood and to allow local checking of the correctness. Each node v is connected at level i to at least one node w_0 and one node w_1 which share the same prefix of length i and have the next bit as 0 respectively 1:

Definition 5 (Farthest Neighbors of HSkip+)**.** We define the farthest predecessors of node v as

$$\begin{aligned}\text{farthestPred}(v, i, b) &= \text{argmax}_{u \in \{u \in V | u.bw > v.bw\}} \\ &\qquad \{\text{prefix}(v, i) \cdot b = \text{prefix}(u, i+1)\} \\ \text{farthestPred}(v, i) &= \text{argmax}_{u \in \{u \in V | u.bw > v.bw\}} \\ &\qquad \min_{b \in \{0,1\}} \{\text{prefix}(v, i) \cdot b = \text{prefix}(u, i+1)\}\end{aligned}$$

and the farthest successors as

$$\begin{aligned}\text{farthestSucc}(v, i, b) &= \text{argmin}_{w \in \{w \in V | w.bw < v.bw\}} \\ &\qquad \{\text{prefix}(v, i) \cdot b = \text{prefix}(w, i+1)\} \\ \text{farthestSucc}(v, i) &= \text{argmin}_{w \in \{w \in V | w.bw < v.bw\}} \\ &\qquad \max_{b \in \{0,1\}} \{\text{prefix}(v, i) \cdot b = \text{prefix}(w, i+1)\}\end{aligned}$$

Also, all nodes between the farthest predecessor and successor in each level are connected. This property aims at a stable neighborhood which prepares the next higher level as the linked list of level $i+1$ is already available. Formally we can define the neighborhood range at level i with the help of the components:

Definition 6 (Range of HSkip+)**.** The node v is connected at level i to all nodes $w \in \text{component}(v, i)$ with $w.bw \leq \text{farthestPred}(v, i).bw$ and $w.bw \geq \text{farthestSucc}(v, i).bw$, which we call the *range* or *neighbors* of node v at level i.

Furthermore, we define different neighborhood shortcuts:

Definition 7 (Neighbors of HSkip+)**.** Let

- $\text{preds}(v, i) = \{u \in \text{neighbors}(v, i) | u.bw > v.bw\}$
- $\text{succs}(v, i) = \{w \in \text{neighbors}(v, i) | w.bw < v.bw\}$
- $\text{closestPred}(v, i) = \text{argmin}_{u \in \text{preds(v,i)}} u.bw$
- $\text{closestSucc}(v, i) = \text{argmax}_{w \in \text{succs(v,i)}} w.bw$

With the three definitions of components, level and range (cf. Def. 3, 4 and 6) in the HSkip+ topology we can now define the desired network. See Fig. 1 for an example.

Definition 8 (HSkip+)**.** The HSkip+ network is defined by $G^{HSkip+} = \left(V, E^{HSkip+}\right)$ with

$$E^{HSkip+} = \{(v, w) | \exists i \in [0, \text{level}] : w \in \text{range}(v, i)\}$$

Figure 1 shows an example network with 8 nodes and 4 levels. The image visualizes the edges caused by the different levels in the HSkip+ topology.

B. HSkip+ Algorithm

To reach the presented topology, we now introduce a self-stabilizing algorithm whose correctness we will prove afterwards. All following operations are executed at node v. The algorithm works in the asynchronous message passing model presented in Section I. We just use two types of guards: $true$ and $received(m)$. $true$ means that the action is continuously enabled, which implies that it is executed infinitely often. In addition to the definitions in Sec. II-A we take use of local variants (e. g. $\text{localFarthestPred}(v, i)$) which represent the cached information at node v.

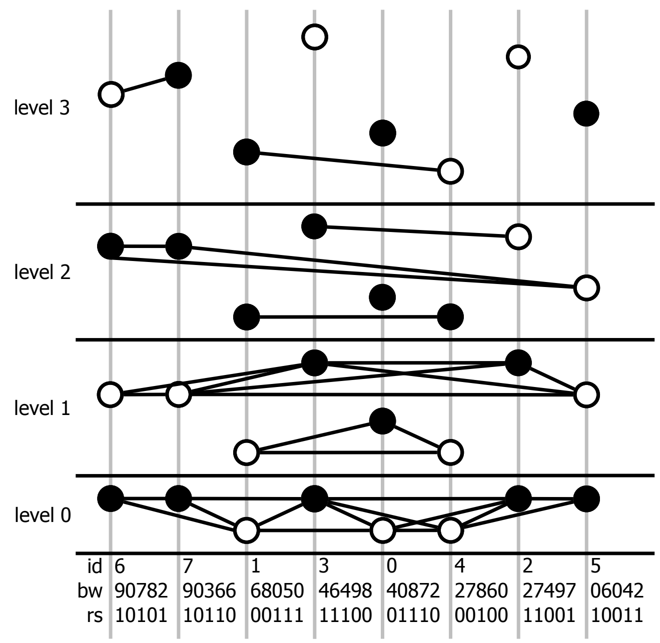
level 3
level 2
level 1
level 0
id 6 7 1 3 0 4 2 5
bw 90782 90366 68050 46498 40872 27860 27497 06042
rs 10101 10110 00111 11100 01110 00100 11001 10011

Fig. 1. Example for the HSkip+ topology with eight nodes.

```
$true \rightarrow$
CheckNeighborhood()
IntroduceNode()
IntroduceClosestNeighbors()
LinearizeNeighbors()
```

Fig. 2. Action called periodically by node v.

The *CheckNeighborhood()* function checks if all nodes which are in the neighborhood of the node v are needed for the topology (cf. Fig. 3). If a node w is no longer needed in any level, it is removed from the neighborhood of v and forwarded to another node x, concretely the node in the neighborhood with the longest common prefix, because it is the most promising node which could include the node in its own neighborhood at any level.

```
**function** CheckNeighborhood
    **for all** $w \in v.nh$ **do**
        **if** CheckNode($w$) = $false$ **then**
            $v.nh = v.nh \backslash \{w\}$
            send $m = (\textit{build}, w)$
                to node $\text{argmax}_{x \in v.nh}\text{commonPrefix}(w, x)$
        **end if**
    **end for**
**end function**
```

Fig. 3. *CheckNeighborhood()* inspects the nodes in $v.nh$.

The test if a node w is really needed for node v is done by the *CheckNode()* function (cf. Fig. 4). For the node w in each level i it is checked if the node w is in the range (cf. Def. 6) of node v. If this is the case for at least one level, the node w is needed in the neighborhood of v.

The other three periodic actions introduce new neighbors to each other to reach the desired topology. In the first function *IntroduceNode()*, node v introduces itself to all of its neighbors to create backward edges (cf. Fig. 5). In the second function *IntroduceClosestNeighbors()*, node v introduces the two direct neighbors, the closest predecessor and the closest successor, in each level to all other neighbors in this level (cf. Fig. 6). The last function *LinearizeNeighbors()* linearizes the neighborhood (cf. Fig. 7), i.e., each neighbor is introduced to the subsequent neighbor.

```
**function** CheckNode(node $w$)
    $needed \leftarrow false$
    **for** $i = 0$ to level($v$) **do**
        **if** prefix$(v, i)$ = prefix$(w, i)$
            $\wedge$(localFarthestPred$(v, i) = \perp$
                $\vee w.bw \leq$ localFarthestPred$(v, i).bw$)
            $\wedge$(localFarthestSucc$(v, i) = \perp$
                $\vee w.bw \geq$ localFarthestSucc$(v, i).bw$)
         **then**
            $needed \leftarrow true$
        **end if**
    **end for**
    **return** $needed$
**end function**
```

Fig. 4. *CheckNode()* tests if a node w is needed in $v.nh$.

```
**function** IntroduceNode
    **for all** $w \in v.nh$ **do**
        send $m = (\textit{build}, v)$ to node $w$
    **end for**
**end function**
```

Fig. 5. *IntroduceNode()* introduces the node v to all neighbors.

In addition to the periodic actions we have different reactive actions triggered by the $received(m)$ guard. Depending on the message type of the message m received by a node v, the *Build*, *Remove* or *Lookup* operation is called (cf. Fig. 8).

The *Build()* function of a node v must handle two cases (cf. Fig. 9): The node x which is given as argument is already in the neighborhood or not. If it is already included, its information (especially its current bandwidth) is updated. Then the *CheckNeighborhood()* operation is called to check if the modified node x and all other nodes are still needed for the node. In the case that the node x is not yet in the neighborhood, it is checked by the *CheckNode()* function (cf. Fig. 4) if the node should be integrated in the neighborhood. If the node is needed in one level, it will be included in the neighborhood and the whole neighborhood will be checked if there is an

```
**function** IntroduceClosestNeighbors
    **for** $i = 0$ to level($v$) **do**
        **if** localClosestPred$(v, i) \neq \perp$ **then**
            **for all** $w \in$ localNeighbors$(v, i)$ **do**
                send $m = (\textit{build}, \text{localClosestPred}(v, i))$
                    to node $w$
            **end for**
        **end if**
        **if** localClosestSucc$(v, i) \neq \perp$ **then**
            **for all** $w \in$ localNeighbors$(v, i)$ **do**
                send $m = (\textit{build}, \text{localClosestSucc}(v, i))$
                    to node $w$
            **end for**
        **end if**
    **end for**
**end function**
```

Fig. 6. *IntroduceClosestNeighbors()* introduces the closest neighbors.

```
function LinearizeNeighbors
    for $i = 0$ to level($v$) do
        for $j = 0$ to $|$localPredecessors$(v, i)| - 1$ do
            send $m = ($build$,$ localPredecessors$(v, i)[j + 1])$
                to localPredecessors$(v, i)[j]$
        end for
        for $j = 0$ to $|$localSuccessors$(v, i)| - 1$ do
            send $m = ($build$,$ localSuccessors$(v, i)[j + 1])$
                to localSuccessors$(v, i)[j]$
        end for
    end for
end function
```

Fig. 7. *LinearizeNeighbors()* introduces the neighbors to each other.

```
$received(m) \rightarrow$
if $m = ($build$, x)$ then
    $Build(x)$
else if $m = ($remove$, x)$ then
    $Remove(x)$
else if $m = ($lookup$, x)$ then
    $Lookup(x)$
end if
```

Fig. 8. The incoming messages at node v are handled.

unnecessary node now. If the new node is not needed by the node, it will be forwarded to the next node w with the longest common prefix.

```
function Build(node $x$)
    if $x \in v.nh$ then
        update neighbor information
        CheckNeighborhood()
    else
        $v.nh = v.nh \cup \{x\}$
        if CheckNode($x$) = $true$ then
            CheckNeighborhood()
        else
            $v.nh = v.nh \backslash \{x\}$
            send $m = ($build$, x)$
                to node $\text{argmax}_{w \in v.nh}$ commonPrefix$(x, w)$
        end if
    end if
end function
```

Fig. 9. *Build* handles an incoming *build* message.

With these self-stabilizing rules the desired topology can be reached and maintained in a finite number of steps.

C. Correctness

Next we show the convergence and the closure of the algorithm, which implies that it is a self-stabilizing algorithm for the HSkip+ topology. Most of the proofs are skipped due to space constraints, but we dissected the self-stabilization process into sufficiently small pieces so that it is not too difficult to verify them with the help of the protocols.

1) Convergence: Here, we prove the following theorem:

Theorem 9 (Convergence). *If $G(t) = (V, E)$ is weakly connected at time t, then $G_e(t') = G^{HSkip+}$ for some time $t' > t$ and all node information is correct.*

Proof: To prove this theorem we will show different lemmas which lead to the convergence. But first, we give an overview of the whole proof: Starting with any weakly connected graph we first show that the network stays connected over time (cf. Lem. 10). Furthermore, we show that all wrong information about nodes in the network will be removed over time (cf. Lem. 11). Having reached this state, we will show the creation of the correct linked list at the bottom level (cf. Lem. 12) and the maintenance of it (cf. Lem. 13). The next step is the proof of the creation of the HSkip+ topology at the bottom level (cf. Lem. 14) and its maintenance (cf. Lem. 15). By showing the inductive creation of the HSkip+ topology at all levels (cf. Lem. 16) we can finish the convergence proof. Due to space constraints the formal proofs of the following lemmas appear in [19].

Note that edges are only deleted if there already exist other edges concerning the desired topology. Otherwise, edges are only added or delegated in a sense that a node u holding an identifier of v may forward that identifier to one of its neighbors w, which also preserves connectivity. Hence, we get:

Lemma 10 (Weakly Connected). *If $G(t) = (V, E)$ is weakly connected at time t, then $G(t')$ is weakly connected at any time $t' > t$.*

Due to our FIFO delivery and fair message receipt assumption, all messages initially in the system will eventually be processed. Moreover, the distance (w.r.t. the sorted ordering of the nodes) of wrong information in the network can be shown to monotonically decrease. Once it is equal to one, the periodic self-introduction (cf. Figure 5) ensures that the node information is corrected, which implies the following lemma.

Lemma 11 (Wrong Information). *If $G(t) = (V, E)$ is weakly connected at time t, then there is a time point t' at which all node information in $G(t'')$ is correct for any $t'' > t'$.*

From now on, let us only consider time steps in which all node information is correct. Next to the edge set of the desired topology E^{HSkip+} we define the edge set E^{HSkip+_i} which contains all edges belonging to the topology at level i and the edge set E^{list_i} which contains all edges needed for a linked list at level i. Then it follows from arguments in [1] for the edge set E^{list_0} at level 0:

Lemma 12 (List Creation). *If $G(t) = (V, E)$ is weakly connected, then $E_e(t') \supseteq E^{list_0}$ at some time $t' > t$.*

Moreover, it follows from the algorithm that the set of linked lists at level i, E^{list_i}, is maintained over time.

Lemma 13 (List Maintenance). *If $E_e(t) \supseteq E^{list_i}$ at time t, then $E_e(t') \supseteq E^{list_i}$ at any time $t' > t$.*

Starting with a linked list at a level i we can show the creation of the HSkip+ links at the same level, E^{HSkip+_i}:

Lemma 14 (HSkip+ Creation). *If $E_e(t) \supseteq E^{list_i}$ at time t, then eventually $E_e(t') \supseteq E^{HSkip+_i}$ at time $t' > t$.*

Additionally, it follows from our rules that the HSkip+ topology at a level i is maintained over time.

Lemma 15 (HSkip+ Maintenance). *If $E_e(t) \supseteq E^{HSkip+_i}$ at time t, then $E_e(t') \supseteq E^{HSkip+_i}$ at any time $t' > t$.*

Using the simple observation that $E^{list_{i+1}} \subseteq E^{HSkip+_i}$, we can then conclude:

Lemma 16 (HSkip+ Induction)**.** *If $E_e(t) \supseteq E^{HSkip+_i}$ at time t, then $E_e(t') \supseteq E^{HSkip+_{i+1}}$ at some time $t' > t$.*

The previous lemmas immediately imply Theorem 9. ■

2) Closure: After proving the convergence of our algorithm we need to show its closure. This means that the network stays in a legal state once it has reached one. Formally, we show (cf. Def. 2):

Theorem 17 (Closure)**.** *If $G_e(t) = G^{HSkip+}$ at time t, then $G_e(t') = G^{HSkip+}$ at any time $t' > t$.*

Proof: Suppose that at time t we have a network which forms the desired topology with its explicit edges, i.e., $E_e(t) = E^{HSkip+}$. If $E_e(t+1) \neq E_e(t)$, then at least one explicit edge is added or removed. Edges are only removed in the *CheckNeighborhood()* operation if they are not needed and edges are only added in the *Build()* operation if they are needed for the neighborhood. However, $G_e(t)$ already forms the correct neighborhood at time t at all levels (cf. Def. 8). As there are no external changes and faults, the neighborhood is still correct at time $t+1$. Therefore, no edge is removed or added, so $G_e(t+1) = G_e(t) = G^{HSkip+}$. ■

The convergence and closure together show the correctness of the self-stabilizing algorithm. In other words we have designed an algorithm which creates the HSkip+ topology and ensures that it stays correct.

D. External Dynamics

As external dynamics of a network we consider all events which can have an influence on the network. Two typical events for a peer-to-peer system are arrivals and departures of nodes. The network has to adapt the topology in this case. In our HSkip+ network we will also consider changes in the bandwidths of the nodes because also this will have an influence on the network topology. To handle these events we need a Join, Leave, and Change operation.

1) Join: The *Join* operation in our network is very simple. If a node v wants to join the network, it just has to introduce itself to some node w. The rest will be handled by the self-stabilization. Hence, it suffices to execute the code in Fig. 10.

```
function JOIN
    send $m = (build, v)$ to a known node $w$
end function
```

Fig. 10. The *Join* operation of node v sends a *build* message to w.

2) Leave: We distinguish between two cases: A scheduled *Leave* and a *Leave* caused by a failure. In the first case (cf. Fig. 11) the node v which wants to leave the network simply sends a *remove* message to all of its neighbors. The receivers of these messages remove the node v from their neighborhood (cf. Fig. 12). The leaving node also deletes its entire neighborhood.

The *Remove()* operation removes a given node from the neighborhood if it is present (cf. Fig. 12). It is executed as reaction to a *remove* message. A neighborhood check is

```
function LEAVE
    for all $w \in v.nh$ do
        send $m = (remove, v)$ to node $w$
    end for
    $v.nh = \emptyset$
end function
```

Fig. 11. The *Leave()* operation of node v sends *remove* messages.

```
function REMOVE(node $x$)
    if $x \in u.nh$ then
        $u.nh = u.nh \backslash \{x\}$
    end if
end function
```

Fig. 12. The *Remove()* operation removes the node x from $v.nh$.

not needed, as there cannot be too much information after removing some information.

Additionally, we can have a *Leave* in the network caused by a node failure. In this case, we assume the existence of a failure detector at the nodes which checks periodically the existence of the neighbor nodes. Therefore, the outcome of a failed node is the same as for a scheduled *Leave*: all neighboring nodes invalidate their links to the failed node.

3) Change: The *Change* operation updates the bandwidth value of a node v and therefore the order in our topology has to be updated. The operation itself only needs to update its internal variable of the bandwidth (cf. Fig. 13).

```
function CHANGE(new bandwidth $bw$)
    $v.bw = bw$
end function
```

Fig. 13. The *Change()* operation updates the bandwidth value of v.

The correct recovery of the topology after no more external dynamics are happening follows directly from the convergence, the formal proofs for the *Join* operation can be found in [19].

Theorem 18 (Recovery from External Dynamics)**.** *If $G_e(t) = G^{HSkip+}$ at time t and a node v joins or a node u leaves or a node u changes its bandwidth, then eventually $G_e(t') = G^{HSkip+}$ at time $t' > t$.*

It is easy to see that the worst case number of structural changes in a level does not exceed $O(\log^2 n)$ w.h.p., but with more refined arguments one can also show the following result:

Theorem 19 (Structural Changes after External Dynamics)**.** *If $G_e(t) = G^{HSkip+}$ at time t and a node v joins or a node u leaves or a node u changes its bandwidth, then $G_e(t') = G^{HSkip+}$ after $O(\log^2 n)$ structural changes, w.h.p.*

With some effort, one can also show that this is an upper bound for the number of additional messages (i.e., messages beyond those periodically created by the $true$ guard).

Theorem 20 (Workload of External Dynamics)**.** *If $G_e(t) = G^{HSkip+}$ at time t and a node v joins or a node u leaves or a node u changes its bandwidth, then $G_e(t') = G^{HSkip+}$ after $O(\log^2 n)$ additional messages, w.h.p.*

E. Routing

The routing of a message to a target node x is handled by the message $m = (\textit{lookup}, x)$. The *Lookup()* operation handles the forwarding of *lookup* messages (cf. Alg. 14). If the lookup target is equal to the current node, the lookup has finished. This is checked with the help of the identifiers of the nodes. If this is not the case, the lookup is forwarded to the next better node.

```
function LOOKUP(node $x$)
    if $v.id = x.id$ then
        done
    else
        $length$
            $\leftarrow \min(\text{level}(v), \text{commonPrefix}(v, x))$
        if localFarthestPred
                $(v, length, x.rs[length]) \neq nil$ then
            $w \leftarrow$ localFarthestPred
                    $(v, length, x.rs[length])$
        else
            $w \leftarrow$ localFarthestSucc
                    $(v, length, x.rs[length])$
        end if
        send $m = (\textit{lookup}, x)$ to node $w$
    end if
end function
```

Fig. 14. The *Lookup()* operation of v handles the *build* messages.

The next hop is determined by the random bit strings of the involved nodes: In each step we want to have at least one bit more in common with the target node x. Formally, if at the current node v the bit string has i bits in common with the target ($\text{commonPrefix}(v, x) = i$), at the next step at node w we want at least $i+1$ bits in common ($\text{commonPrefix}(w, x) \geq i+1$). In our topology such a node is always present in level i of a node v in both the successors and the predecessors (unless there is no such higher level) as every node is connected at every level i to at least one predecessor and successor with 0 and one with 1 as next bit. First, the routing is done along predecessors so that the message is guaranteed to follow a sequence of nodes of monotonically increasing bandwidth. Once the node with the highest bandwidth defined by this rule has been reached, we use the successor nodes until the target node is reached. From the routing protocol it follows:

Theorem 21 (Correctness of Routing)**.** *If $G_e = G^{HSkip+}$, then a message $m = (\text{lookup}, w)$ sent by a node u eventually reaches node w.*

We can also guarantee the following critical property, this is important to keep the congestion low.

Lemma 22 (Involved Nodes in Routing)**.** *If $G_e = G^{HSkip+}$, the routing from some node u to w only uses nodes v with $v.bw \geq \min\{u.bw, w.bw\}$.*

The *dilation* is defined as the longest path which is needed to route a packet from an arbitrary source to an arbitrary target node. Since it just takes a single hop to climb up one level and there are at most $O(\log n)$ levels w.h.p., we get:

Theorem 23 (Dilation of Routing)**.** *In HSkip+ the dilation is at most $O(\log n)$ w.h.p.*

Formal proofs for the correctness and the dilation appear in [19]. Finally, we analyze the congestion of our routing strategy. The *congestion* of a node v is equal to the total volume of the messages passing it divided by its bandwidth. Let us consider the following routing problem: Each node u selects a target bitstring x independently and uniformly at random and sends a message of volume $\min\{u.bw, w.bw\}$ to the node w with the longest prefix match with x. Of course, u may not know that volume in advance. Our goal will just be to show that the expected congestion of this routing problem is $O(\log n)$. If this is the case, then one can show similar to [15] that the expected congestion of routing any routing problem in HSkip+ is by a factor of at most $O(\log n)$ worse than the congestion of routing that routing problem in any other topology of logarithmic degree, which matches the result in [15].

Theorem 24 (Congestion of Routing)**.** *In the HSkip+ topology, we can route messages with the presented routing algorithm (cf. Alg. 14) and the defined routing problem with an expected congestion of $O(\log n)$.*

To prove the congestion we divide the routing process of a message from an arbitrary source node u to an arbitrary target node w into two parts: On the one hand the routing from u along predecessor edges to an intermediate node v which has no better predecessors and on the other hand the routing back over successors from node v to the target node w. We will start with the first part of the routing process, and initially we separately look at each level i in the network at node v. Later, we will sum it up for all levels.

Lemma 25 (Congestion of node v at level i)**.** *In our routing problem (cf. The. 24), every node v has an expected congestion of $O(1)$ at every level $i \geq 1$.*

Proof: We assume w.l.o.g. that the first i bits of $v.rs$ are 1. So messages with the target bitstring starting with $1 \ldots 1$ as the first i bits will be sent through the node v as long as they start at a node w that, at level 0, is between v and the closest successor w' of v whose first i bits are also 1. We calculate the probability that there are m nodes between v and w'. For each node in between we have the probability $(1-(\frac{1}{2})^i)$ that one of its first i bits differs from 1, and for w' we get a probability of $(\frac{1}{2})^i$ that its first i bits are 1. Altogether, we get a probability that there are m sources that may send their messages to v of $(1-(\frac{1}{2})^i)^m \cdot (\frac{1}{2})^i = \frac{(2^i-1)^m}{(2^i)^{m+1}}$. Furthermore, we have to estimate the fraction of messages which will be sent through v. Each message is sent through v if the first i bits are 1, which has a probability of $\frac{1}{2^i}$. Since we have m possible sources (excluding v), a fraction of $(m+1) \cdot \frac{1}{2^i}$ messages will be sent through v. Lastly, we have to look at the volume sent by a single source. We know that a node s can send at most a volume of $s.bw$. As we have only successors of v as sources and for all successors s it holds that $s.bw < v.bw$, we can upper bound the volume sent by each source by $v.bw$. Altogether, if we have m nodes between v and w', the total expected volume through v at level i is at most $(m+1) \cdot \frac{1}{2^i} \cdot v.bw$. Let the random variable X_v^i be the total volume sent through v at level i. Then

$$E\left[X_v^i\right] = \sum_{m=0}^{\infty} \frac{(2^i-1)^m}{(2^i)^{m+1}} \cdot (m+1) \cdot \frac{1}{2^i} \cdot v.bw = O(v.bw)$$

■

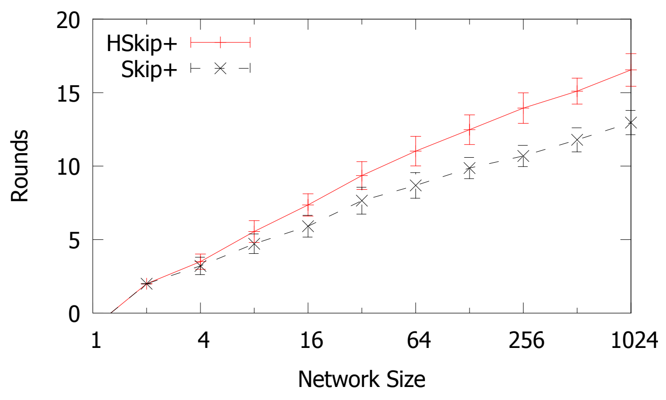

Fig. 15. Stabilization Time.

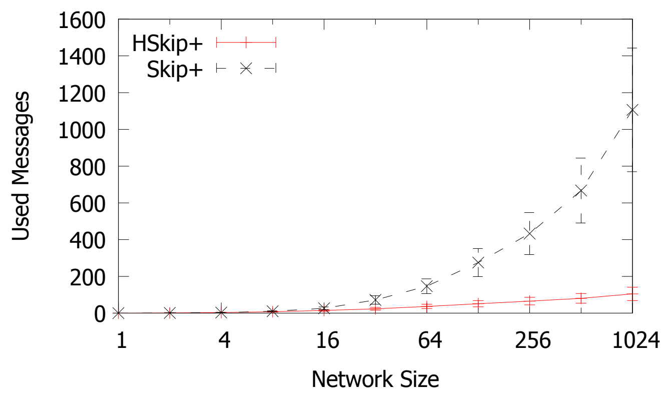

Fig. 16. Overall Used Messages Total.

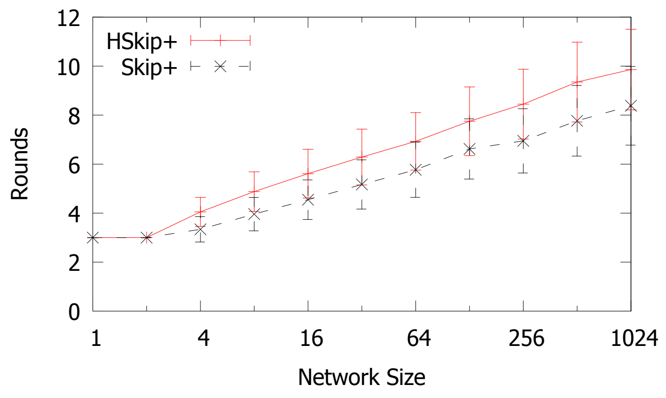

Fig. 17. Stabilization Time after Join.

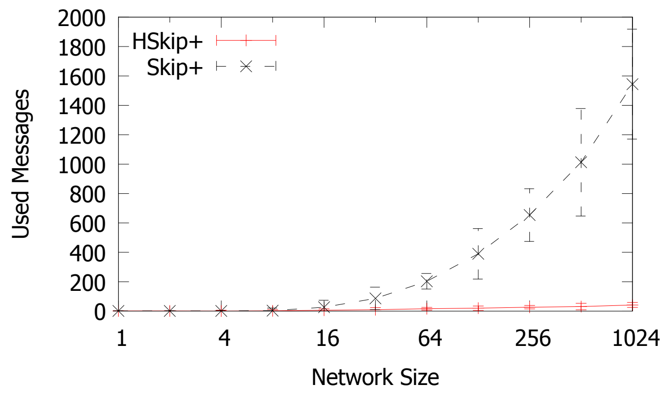

Fig. 18. Used Messages after Join.

Summing up the congestion over all levels, we get:

Lemma 26 (Congestion of node v). *In our routing problem (cf. The. 24), an arbitrary node v has an expected congestion of $O(\log n)$.*

Proof of Theorem 24: With the previous lemmas (cf. Lem. 25 and Lem. 26) we have calculated the congestion caused by the first part of the routing process. For the second part, we can argue in a similar way by looking at the routing path backwards from the target node x to the intermediate node v and we also get an expected volume of $O(\log n) \cdot v.bw$ at node v. Both routing parts together yield an expected congestion of $O(\log n)$ at every node v. ■

III. Simulations

In this section we present simulation results for the HSkip+ network in comparison to the Skip+ graph. We simulated both protocols starting from randomly generated trees. The bandwidths were assigned randomly to the peers according to measurements of the connections in Germany [21]. All simulations are run 100 times with different seeds for network configurations with up to 1024 participating nodes. Since both networks are self-stabilizing, we focus on the stabilization costs in terms of rounds (in each round every node is allowed to process all received messages) until the desired topology is constructed from an initial weakly connected network and the number of messages which were used in this process. The simulator can be found together with the long version of this paper in [19].

A. Self-Stabilization

If we look at the self-stabilization time for Skip+ and HSkip+ (cf. Fig. 15), we see asymptotically similar results. Both networks need $O(\log n)$ rounds for the self-stabilization, whereas the Skip+ has slightly better results. In contrast, if we look at the used messages (cf. Fig. 16), we see a big difference between both networks. While Skip+ uses $O(\log^4 n)$ messages for the self-stabilization process, the HSkip+ topology requires only $O(\log^2 n)$ messages.

B. External Dynamics

As the next step, we look at the external dynamics in the network, namely joining and leaving nodes, as well as bandwidth changes. We start with the needed rounds after a new node joined the network (cf. Fig. 17). The result is similar to the self-stabilization we have seen before. Both networks are stabilized again after $O(\log n)$ rounds, where the Skip+ network is slightly faster. Also the overall used messages for this process reflect the already seen result (cf. Fig. 18). While the Skip+ network consumes $O(\log^4 n)$ messages, the rules of HSkip+ need only $O(\log^2 n)$ messages. The concrete number of messages are not directly comparable to the numbers in Fig. 16 as there are further messages in the system which cannot be excluded from counting.

Also the other two operations show similar results. The self-stabilization after a node left the network or changed its bandwidth can be finished in $O(\log n)$ rounds and nearly in constant time (cf. Fig. 19 and 21). This tendency yields also for the used messages. The work consumed by HSkip+ is clearly less than in the case of Skip+ (cf. Fig. 20 and 22).

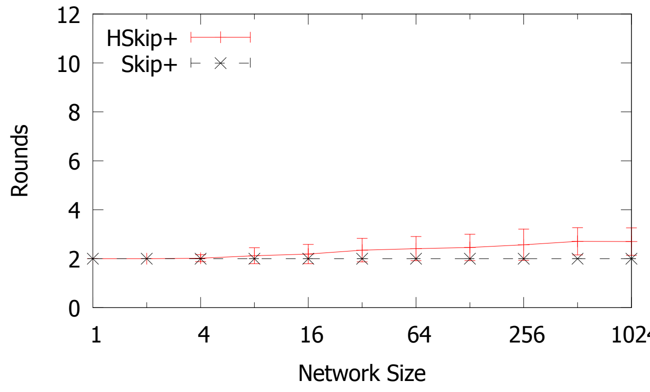

Fig. 19. Stabilization Time after Leave.

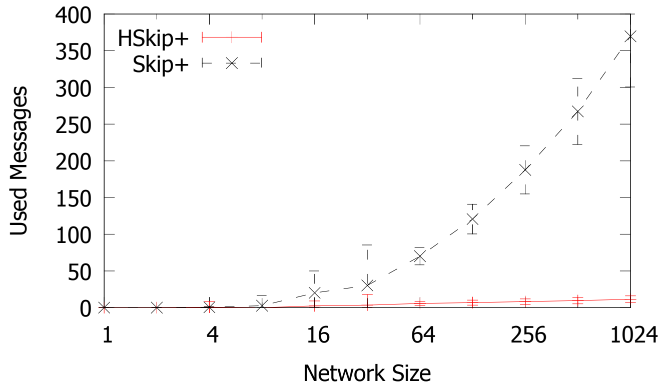

Fig. 20. Used Messages after Leave.

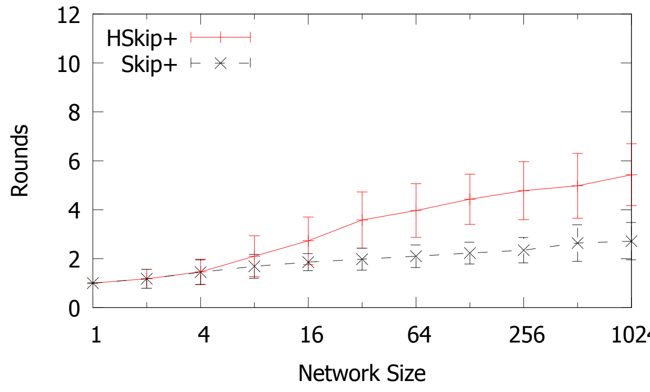

Fig. 21. Stabilization Time after Change.

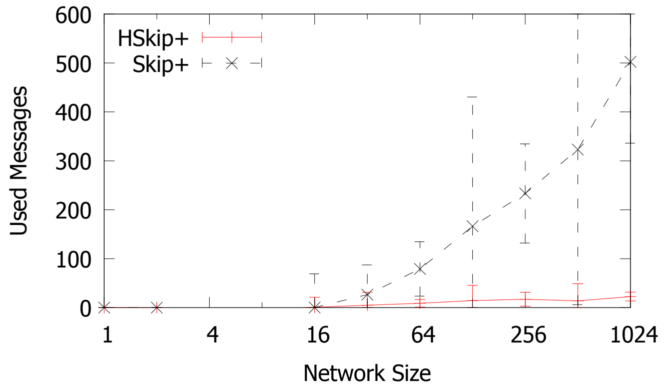

Fig. 22. Used Messages after Change.

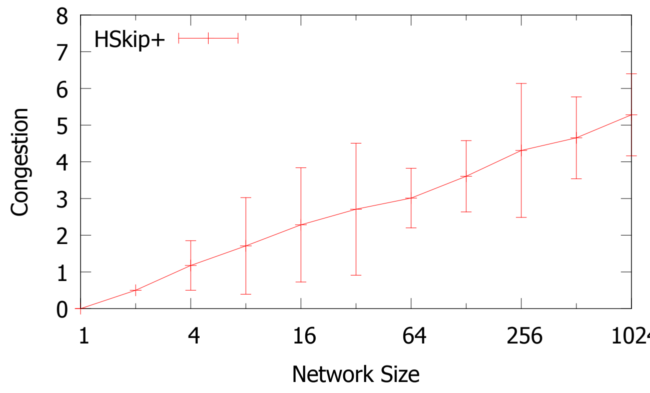

Fig. 23. Routing Congestion of Flow Problem.

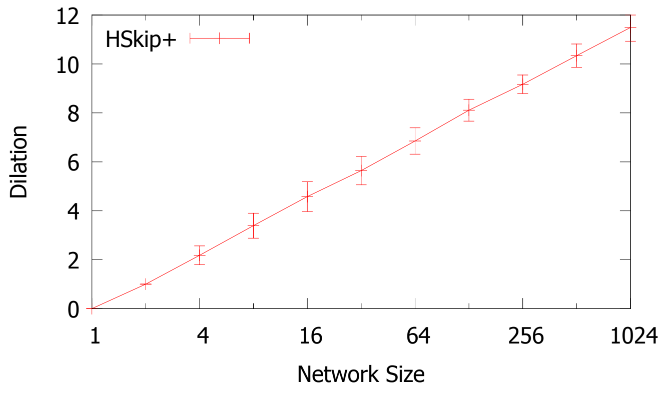

Fig. 24. Routing Dilation of Flow Problem.

C. Routing

For evaluating the routing performance of the topology we look at a flow problem: For each node pair $u, v \in V$ node u sends an amount of data of $\frac{u.bw \cdot v.bw}{\sum_{w \in V} w.bw}$ to node v. The average normalized congestion (according to the nodes' bandwidth) is logarithmic in the network size (cf. Fig. 23). The same yields for the dilation of the routing process (cf. Fig. 24). Both results agree with the theoretical findings as proved in Sec. II-E.

D. Behavior under Churn

As last aspect we look at the network behavior under churn. In a practical usage scenario of a peer-to-peer system nodes can arbitrary join and leave the network. Usual churn behavior as it was studied in several papers (i.e. [22], [23]) has no influence on the topology as the stabilization times of our network are significantly smaller than the session and intersession times of peers in a network. Therefore, we focus on two exceptional scenarios: On the one hand we examined a crash and on the other hand an adversarial attack. In the first scenario x% randomly chosen nodes are leaving from a network with 1024 nodes at the same time and the same number of new nodes join the network (to have a comparable size of the network). In the second scenario the leaving nodes are no longer randomly chosen, but all from a neighboring area.

The number of leaving nodes has no remarkable influence on the stabilization time as we have seen it in Fig. 17, Fig. 19 and Fig. 21. Hence, we concentrate on the topology after stabilization and especially how many nodes are still connected correctly (cf. Fig. 25). Up to a churn rate of about 35% all nodes stay in the topology, under a random crash even up to a rate of 60%. Over this limit, the topology looses nodes which are no longer reachable. As for the second evaluation,

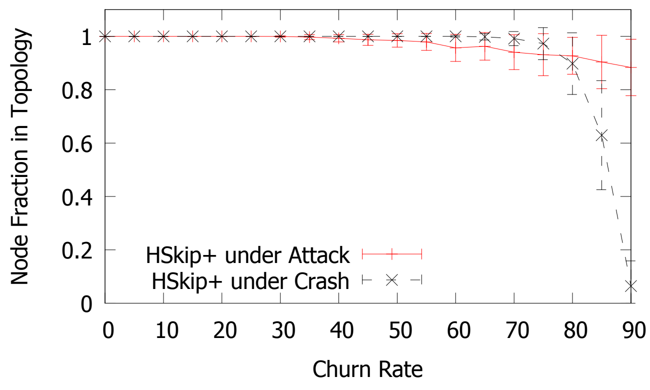

Fig. 25. Stabilization after Crash and Attack.

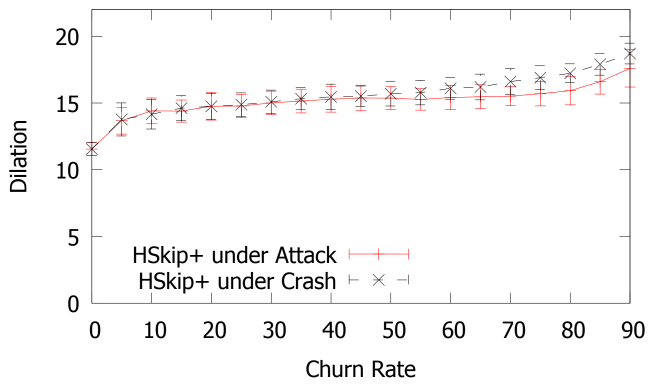

Fig. 26. Routing Dilation during Crash and Attack.

we concentrate on the dilation. We have used the same flow problem as in Sec. III-C starting at the same round as the attack and crash. In a stable network with 1024 nodes, it is about 11 hops (cf. Fig. 24). During the attack or crash it is just a little increased to about 14 or 15 hops; only at a very high churn rate (when we also have lost many nodes) the dilation increases noticeable.

IV. Conclusion

In this paper, we presented HSkip+, a self-stabilizing overlay network based on Skip+. We showed by simulations that the self-stabilization time is nearly identical in $O(\log n)$ while the improved version uses only $O(\log^2 n)$ messages for the process compared to the originally used $O(\log^4 n)$ messages. Also the dealing with external dynamics can be managed in the same time bounds and with the same work. Furthermore, we have extended the network to deal with heterogeneous bandwidths where we reach a logarithmic congestion and dilation in the routing process. Finally, the practical usage was shown by simulations under churn behavior.

References

[1] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig, "A distributed polylogarithmic time algorithm for self-stabilizing skip graphs," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '09)*, 2009, pp. 131–140.

[2] S. Kniesburges, A. Koutsopoulos, and C. Scheideler, "CONE-DHT: A Distributed self-stabilizing algorithm for a heterogeneous storage system," in *Proceedings of the International Symposium on Distributed Computing (DISC'13)*, 2013.

[3] R. M. Nor, M. Nesterenko, and C. Scheideler, "Corona: a stabilizing deterministic message-passing skip list," in *Proceedings of the International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*, 2011, pp. 356–370.

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974.

[5] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1, pp. 45–67, Mar. 1993.

[6] A. Shaker and D. S. Reeves, "Self-Stabilizing Structured Ring Topology P2P Systems," in *Proceedings of the IEEE International Conference on Peer-to-Peer Computing (P2P '05)*, 2005, pp. 39–46.

[7] D. Gall, R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig, "Time complexity of distributed topological self-stabilization: the case of graph linearization," in *Proceedings of the Latin American Conference on Theoretical Informatics (LATIN '10)*, 2010, pp. 294–305.

[8] S. Dolev and N. Tzachar, "Spanders: Distributed spanning expanders," *Sci. Comput. Program.*, vol. 78, no. 5, pp. 544–555, May 2013.

[9] R. Jacob, S. Ritscher, C. Scheideler, and S. Schmid, "Towards higher-dimensional topological self-stabilization: A distributed algorithm for Delaunay graphs," *Theor. Comput. Sci.*, vol. 457, pp. 137–148, Oct. 2012.

[10] S. Dolev and R. I. Kat, "HyperTree for Self-Stabilizing Peer-to-Peer Systems," in *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA '04)*, 2004, pp. 25–32.

[11] S. Kniesburges, A. Koutsopoulos, and C. Scheideler, "Re-Chord: a self-stabilizing chord overlay network," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*, 2011, pp. 235–244.

[12] A. Berns, S. Ghosh, and S. V. Pemmaraju, "Building self-stabilizing overlay networks with the transitive closure framework," in *Proceedings of the International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*, 2011, pp. 62–76.

[13] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Löser, "Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks," in *Proceedings of the ACM International Conference on World Wide Web (WWW '03)*, 2003, pp. 536–543.

[14] M. Srivatsa, B. Gedik, and L. Liu, "Scaling Unstructured Peer-to-Peer Networks With Multi-Tier Capacity-Aware Overlay Topologies," in *Proceedings of the IEEE Parallel and Distributed Systems, Tenth International Conference (ICPADS '04)*, 2004.

[15] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober, "Pagoda: a dynamic overlay network for routing, data management, and multicasting," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, 2004, pp. 170–179.

[16] C. Scheideler and S. Schmid, "A Distributed and Oblivious Heap," in *Proceedings of the Internatilonal Collogquium on Automata, Languages and Programming (ICALP '09)*, 2009, pp. 571–582.

[17] R. Meier and R. Wattenhofer, "Peer-to-Peer Streaming in Heterogeneous Environments," *Signal Processing: Image Communication*, vol. 27, no. 5, pp. 457–469, March 2012.

[18] B. Awerbuch and C. Scheideler, "The hyperring: a low-congestion deterministic data structure for distributed environments," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, pp. 318–327.

[19] M. Feldotto, C. Scheideler, and K. Graffi, "HSkip+: A Self-Stabilizing Overlay Network for Nodes with Heterogeneous Bandwidths," *CoRR*, vol. arXiv:1408.0395 [cs.DC], 2014.

[20] J. Aspnes and G. Shah, "Skip graphs," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*. Society for Industrial and Applied Mathematics, 2003, pp. 384–393.

[21] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen, "Tätigkeitsbericht 2012/2013," 2013.

[22] K. Pussep, C. Leng, and S. Kaune, "Modeling User Behavior in P2P Systems," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 447–461.

[23] M. Steiner, T. En-Najjary, and E. W. Biersack, "Long term study of peer behavior in the KAD DHT," *IEEE/ACM Trans. Netw.*, vol. 17, no. 5, pp. 1371–1384, 2009.