Invited Paper: Robust Architectures for Open Distributed Systems and Topological Self-Stabilization

Stefan Schmid
T-Labs / TU Berlin
Berlin, Germany
stefan@net.t-labs.tu-berlin.de

ABSTRACT

Distributed systems are often dynamic in the sense that there are frequent membership changes (nodes joining and leaving the network), either due to regular churn or due to an attack. Maintaining availability and full functionality of such a system under continuous topological changes hence constitutes an important algorithmic challenge. This paper reports on some of our recent results on robust distributed systems. We review two randomized architectures that build upon the continuous-discrete approach by Naor and Wieder, namely the *SHELL network* which allows for fast joins and leaves and organizes more reliable (or stronger) nodes in a core network where their communication is not affected by malicious (or weak) nodes, and the *Chameleon network* whose replica placement strategy and whose intentional topological updates ensure resiliency against denial-of-service attacks, even from past insiders. To complement our investigations on randomized architectures, we discuss algorithms to maintain hypercubic networks under worst-case churn. Finally, we advocate the design of self-stabilizing topologies—a very appealing and still not well-understood notion of robustness—that converge quickly to a desirable structure from arbitrarily degenerated states. As a use case, graph linearization is examined in more detail. This invited paper complements the WRAS'10 talk and is joint work with Matthias Baumgart, Dominik Gall, Riko Jacob, Fabian Kuhn, Andrea Richa, Stephan Ritscher, Christian Scheideler, Joest Smit, Hanjo Täubig, and Roger Wattenhofer.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Routing and Layout*

General Terms

Algorithms, Reliability, Theory

Keywords

Networking, Self-Stabilization, Churn

1. INTRODUCTION

Every application run on multiple machines needs a mechanism that allows the machines to exchange information. A naive solution is to store at each machine the name (e.g., IP address) of every other machine. While this may work well for a small number of machines, large-scale distributed applications such as file sharing, grid computing, cloud computing, or data center networking systems need a different, more scalable approach: instead of forming a clique (where everybody knows everybody else), each machine should only be required to know some small subset of other machines. The resulting graph of knowledge can be seen as a logical network interconnecting the machines; it is also known as *overlay network*. A prerequisite for an overlay network to be useful is that it has good topological properties. Among the most important are: small node degree, small network diameter, or absence of congestion bottlenecks.

A distinguishing property of many (especially open) distributed systems are the frequent membership changes. Nodes on an overlay network may only join for a certain duration to make use of the services (regular "churn"), or they may be unavailable for a certain time period due to an attack. While static topologies are understood well today, researchers have only started to gain insights into the behavior and maintenance of dynamic distributed systems.

WRAS'10, July 29, 2010, Zurich, Switzerland.
ACM 978-1-60558-888-9/10/07.

2. DYNAMIC OVERLAYS

This section reviews an interesting design principle to construct dynamic overlays: the continuous-discrete approach. Subsequently, we present the distributed heap SHELL and the Chameleon network which build upon the continuous-discrete approach. We conclude the section with a mechanism to maintain networks under worst-case membership changes.

2.1 The Continuous-Discrete Approach

A simple and appealing approach to design dynamic distributed (peer-to-peer like) systems is the *continuous-discrete approach* described by Naor and Wieder [24]. It is based on a "think continuously, act discretely" strategy, and applies to a variety of topologies. The idea is as follows: Let I be a Euclidean space, e.g., a 1-dimensional line or cycle. Let G_c be a (infinite) graph where the vertex set is given by the continuous set I, and where each point in I is connected to some other points in I. The actual network is a discretization of this continuous graph based on a dynamic decomposition of the underlying space I into *cells* where each node (or machine/peer/server) is responsible for a cell. Two cells are connected if they contain adjacent points in the continuous graph. Clearly, the partition of the space into cells should be maintained in a distributed manner. When a join operation is performed an existing cell splits, when a leave operation is performed two cells are merged into one.

The recipe to design a dynamic and scalable network is as follows: (1) Choose a proper continuous graph G_c over the continuous

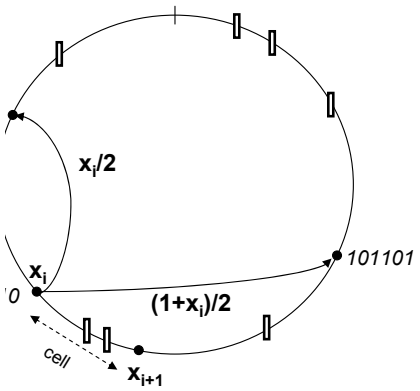

Figure 1: The continuous-discrete approach for the dynamic de Bruijn graph. Nodes are indicated using circles, files using rectangles. In the continuous setting, the node at position $x_i = .011010$ (in binary notation) is connected to positions $x_i/2$ and $(1 + x_i)/2$. In the discrete setting, it is responsible for the cell (i.e., the connections and files which are mapped there) between positions x_i and x_{i+1}.

space I. Design the algorithms in the continuous setting. (This is typically simpler than in the discrete setting: there is no need to deal with scalability issues and standard mathematical tools can be used for proving statements.) (2) Find an efficient way to discretize the continuous graph in a distributed manner, such that the algorithms designed for the continuous graph would perform well in the discrete graph. The discretization is done via a decomposition of I into cells. If the cells which compose I are allowed to overlap then the resulting graph would be fault tolerant.

To give an example, in order to build a dynamic peer-to-peer network—a *distributed hash table* (DHT) where nodes collaboratively store files which are mapped to the $[0, 1)$ interval as well, e.g., by a random hash function—featuring a *de Bruijn* topology of logarithmic diameter and constant node degree, a node at position $x \in [0, 1)$ (in binary form $b_1 b_2 ...$ such that $x = \sum_{i=1}^{\infty} 2^{-b_i}$) connects to positions $l(x) := x/2 \in [0, 1)$ and $r(x) := (1 + x)/2 \in [0, 1)$ in G_c (out-degree two per node). Observe that if position x is written in binary form, then $l(x)$ effectively shifts '0' into the left and $r(x)$ shifts a '1' into the left. Moreover, observe that routing on the corresponding overlay network is straight-forward: based solely on the current position and the destination (without the overhead of maintaining routing tables), a message can be forwarded by fixing one bit after the other.

The set of nodes in the cyclic $[0, 1)$ space then define the *discrete* graph: Let x_i denote the position of the i^{th} node (ordered in increasing order w.r.t. position). Node i is responsible for the cell $[x_i, x_{i+1})$, computed in a modulo manner, that is, this node is responsible to store the data (or files) mapped to this cell plus for the establishment of the corresponding connections defined in G_c. Figure 1 gives an example.

2.2 The SHELL Heap

There are several dynamic distributed systems that are based on the continuous-discrete approach. One example is SHELL [28]. SHELL is motivated by the observation that in many systems with open clientele, nodes that have joined earlier are likely to stay longer also in the future. SHELL organizes the nodes in a *distributed heap*: the topology can be regarded as a redundant continuous-discrete de Bruijn graph where a node v connects to entire intervals in the continuous space, but only to those nodes in these intervals that joined the system before v. By its connection and routing policy, SHELL ensures that whenever possible, communication between older nodes is constrained to the more stable core network. Concretely, it can be shown that a message sent from some node u that joined at time t' to some node v that joined at time t'' only traverses nodes that joined before $\max\{t', t''\}$. Thus, SHELL is resilient to certain types of Sybil attacks: A set of malicious nodes that join at time t trying to flood the network cannot disrupt communication between nodes that arrived before t. Alternatively, SHELL can be used to organize heterogeneous nodes such that more powerful nodes can collaborate directly with each other, i.e., they do not depend on (and hence are slowed down by) contributions from weaker nodes.

The SHELL heap has some interesting properties: it 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 departure or fault does not entail much work and can be dealt with in a constant number of communication rounds.

In summary, SHELL 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., which is on par with well-known peer-to-peer networks like Chord [30].

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 manner (e.g., for streaming applications).

2.3 The Chameleon System

Distributed DoS attacks are believed to be one of the biggest problems in today's open distributed systems such as the Internet. Attackers use the fact that Internet servers are typically accessible to anyone in order to overload them with bogus requests from so-called *bot nets*, which are large groups of machines that are under their control. Some popular information services like Google and Akamai are under constant DoS attacks, and also the Domain Name System has been hit several times by major DoS attacks during the last years.

The predominant approach to deal with the threat of DoS-attacks is the introduction of *redundancy*. Information which is replicated on multiple machines is more likely to remain accessible during a DoS attack. However, replication can entail a large overhead in terms of storage and update costs. In order to preserve scalability, it is therefore vital that the burden on the servers be minimized.

Chameleon [7] is a distributed information system (over a set of completely connected servers) which is robust to Denial-of-Service (DoS) attacks on the nodes as well as the operations of the system, while using a small degree of redundancy only. It is another example of a system building upon continuous-discrete principles. Chameleon employs a randomized replication scheme whose appearance cannot be predicted by the attacker though, maybe at first

sight paradoxically, the data can still be efficiently located. Interestingly, due to randomization, a polylogarithmic redundancy factor is enough to deal with an adversary blocking a constant fraction of all servers. In addition, Chameleon pro-actively changes its appearance over time (hence the system's name), which renders the network resilient even to past insiders who have full knowledge of the system's internals up to a certain (unknown) time point t_0. Despite ongoing attacks, Chameleon can process put and get requests efficiently at any time.

More specifically, Chameleon ensures (before and after t_0, *without* knowing t_0):

1. *Scalability*: Every node spends at most polylogarithmic time (number of communication rounds) and work (number of messages) in order to serve all requests, and no node will get overloaded over time.

2. *Robustness*: All get requests for data that was inserted or last updated *after* t_0 are served correctly under any adversarial attack within our model.

Achieving these conditions is not an easy task as the system cannot afford to continuously replace all the data in it (recall that the system does *not* know t_0 and we have no bound on the number of data items in the system). Also, no long-term information hiding techniques can be used (as the adversary has *full* knowledge of the system up to phase t_0).

2.4 Worst-Case Churn

The analysis of fault tolerance of dynamic distributed systems usually only considers random faults of some kind, and indeed, many of the guarantees discussed above are subject to some probabilistic assumptions. Contrary to traditional algorithmic research, faults as well as joins and leaves occurring in a worst-case manner are hardly considered in dynamic distributed systems. Moreover, most fault tolerance analyses are static in the sense that only a functionally bounded number of random nodes can be crashed. After removing a few nodes the system is given sufficient time to recover again. The more realistic dynamic case where worst-case faults steadily occur has not found much attention.

What degree of dynamics can be tolerated by a distributed network subject to *continuous* membership changes happening in a *worst-case* manner? In [20], a deterministic framework is developed that allows us to design maintenance algorithms for distributed hash tables (DHTs) on different topologies. For certain networks these algorithms are proved to achieve an "optimal robustness" in the sense that there is no alternative system which can tolerate higher worst-case churn rates without disconnecting. The basic idea is to *simulate* a given network graph, for example, a *hypercube* [19]: Each node is part of a distinct hypercube *vertex*; each hypercube vertex consists of a logarithmic number of nodes. Nodes have connections to other nodes of their hypercube vertex and to nodes of the neighboring hypercube vertices. It is assumed that the data items are mapped to vertices by a hash function, and hence, in order to prevent data loss, there must always be at least one node per vertex left. Therefore, after a number of joins and leaves, some nodes may have to change to another hypercube vertex such that up to constant factors, all hypercube vertices have the same cardinality at all times. If the total number of nodes grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively. The balancing of nodes among the vertices can be seen as a *dynamic token distribution problem* on the hypercube: Each vertex of a graph (hypercube) has a certain number of tokens, and the goal is to distribute the tokens along the edges of the graph such that all vertices end up with the same or almost the same number of tokens.

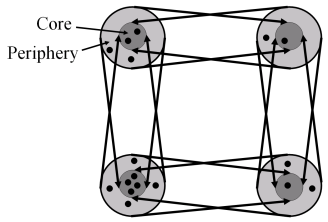

Figure 2: A simulated 2-dimensional hypercube with four vertices, each consisting of a *core* and a *periphery*. All nodes within the same vertex are completely connected to each other, and additionally, all nodes of a vertex are connected to all core nodes of the neighboring vertices.

The system in [20] builds on two basic components: i) an algorithm which performs the described dynamic token distribution and ii) an information aggregation algorithm which is used to estimate the number of nodes in the system and to adapt the hypercube's dimension accordingly. The following result can be derived.

THEOREM 2.1. *Given an adversary who inserts and removes at most a logarithmic number of nodes per communication round, there is an algorithm which ensures that 1) every vertex always has at least one node and hence no data is lost; 2) each node has degree* $\Theta(\log n)$*; and 3) the network diameter is logarithmic as well.*

Figure 2 visualizes the simulated hypercube topology. For efficiency reasons and in order to minimize data moves, the nodes in a vertex are divided into a *core* (storing data) and a *periphery* (used for balancing nodes between vertices). These techniques are applicable to alternative graphs, like pancake graphs [18].

3. SELF-STABILIZATION

An appealing and strong notion of robustness is *topological self-stabilization*: A system is called self-stabilizing if it guarantees that from any weakly connected initial state (e.g., a degenerated system after an attack), in the absence of further membership changes, it will quickly converge to a desirable network. In this section, as a most simple but already non-trivial use case, self-stabilizing graph linearization is considered in more detail. Subsequently, we briefly report on results for 2-dimensional linearization (namely, self-stabilizing Delaunay graphs) and skip graphs.

3.1 Graph Linearization

Graph linearization can be regarded as the *drosophila melanogaster* of topological self-stabilization: We investigate how to recover a sorted list—i.e., how to *linearize* a graph—from *any* connected state. This section is based on the material presented in [13].

Formally, we are given a system consisting of a fixed set V of $n = |V|$ nodes. Every node has a unique (but otherwise arbitrary) integer *identifier*. In the following, if we compare two nodes u and v using the notation $u < v$ or $u > v$, we mean that the identifier of u is smaller than v or vice versa. For any node v, $\mathrm{pred}(v)$ denotes the predecessor of v (i.e., the node $u \in V$ of largest identifier with $u < v$) and $\mathrm{succ}(v)$ denotes the successor of v according to "$<$". Two nodes u and v are called *consecutive* if and only if $u = \mathrm{succ}(v)$ or $v = \mathrm{succ}(u)$.

Each pair (u, v) of nodes shares a Boolean variable $e(u, v)$ which specifies an undirected adjacency relation: u and v are called

neighbors if and only if this shared variable is true. The set of neighbor relations defines an undirected graph $G = (V, E)$ among the nodes. A variable $e(u, v)$ can only be changed by u and v, and both u and v have to be involved in order to change $e(u, v)$. (E.g., node u sends a change request message to u.) For any node $u \in V$, let $u.L$ denote the set of left neighbors of u—the neighbors which have smaller identifiers than u—and $u.R$ the set of right neighbors (with larger IDs) of u. $\deg(u)$ will denote the degree of a node u and is defined as $\deg(u) = |u.L \cup u.R|$. Moreover, the distance between two nodes $\text{dist}(u, v)$ is defined as $\text{dist}(u, v) = |\{w : u < w \leq v\}|$ if $u < v$ and $\text{dist}(u, v) = |\{w : v < w \leq u\}|$ otherwise. The length of an edge $e = \{u, v\} \in E$ is defined as $\text{len}(e) = \text{dist}(u, v)$.

We consider distributed algorithms which are run by each node in the network. The program executed by each node consists of a set of *variables* and *actions*. An action has the form

$$< \text{name} > \quad : \quad < \text{guard} > \quad \rightarrow \quad < \text{commands} >$$

where $<$ name $>$ is an *action label*, $<$ guard $>$ is a Boolean predicate over the (local and shared) variables of the executing node and $<$ commands $>$ is a sequence of commands that may involve any local or shared variables of the node itself or its neighbors. Given an action A, the set of all nodes involved in the commands is denoted by $V(A)$. Every node that either owns a local variable or is part of a shared variable $e(u, v)$ accessed by one of the commands in A is part of $V(A)$. Two actions A and B are said to be *independent* if $V(A) \cap V(B) = \emptyset$. For an action execution to be scalable we require that the number of interactions a node is involved in (and therefore $|V(A)|$) is *independent* of n. An action is called *enabled* if and only if its guard is true. Every enabled action is passed to some underlying scheduling layer (to be specified below). The scheduling layer decides whether to accept or reject an enabled action. If it is accepted, then the action is executed by the nodes involved in its commands.

We model distributed computation as follows. The assignments of all local and shared variables define a *system state*. Time proceeds in *rounds*. In each round, the scheduling layer may select any set of independent actions to be executed by the nodes. The *work* performed in a round is equal to the number of actions selected by the scheduling layer in that round. A *computation* is a sequence of states such that for each state s_i at the beginning of round i, the next state s_{i+1} is obtained after executing all actions that were selected by the scheduling layer in round i. A distributed algorithm is called *self-stabilizing* w.r.t. a set of system states S and a set of *legal* states $L \subseteq S$ if for any initial state $s_1 \in S$ and any fair scheduling layer, the algorithm eventually arrives (and stays) at a state $s \in L$.

A distributed algorithm is called *self-stabilizing* in this context if for any initial state that forms a connected graph, it eventually arrives at a state in which for all node pairs (u, v),

$$e(u, v) = 1 \quad \Leftrightarrow \quad u = \text{succ}(v) \ \vee \ v = \text{succ}(u)$$

i.e., the nodes indeed form a sorted list. Once it arrives at this state, it should stay there, i.e., the state is the (only) *fixpoint* of the algorithm.

In the algorithms we propose, each node $u \in V$ repeatedly performs simple linearization steps in order to arrive at that fixpoint: A linearization step involves three nodes u, v, and v' with the property that u is connected to v and v' and either $u < v < v'$ or $v' < v < u$. In both cases, u may command the nodes to move the edge $\{u, v'\}$ to $\{v, v'\}$. If $u < v < v'$, this is called a *right* linearization and otherwise a *left* linearization (see also Figure 3). Since only three nodes are involved in such a linearization, this can be formulated by a scalable action. Henceforth, we will refer to u, v, and v' as a *linearization triple*.

Figure 3: Left and right linearization step.

We study two most simple distributed and self-stabilizing linearization algorithms: LIN_{all} and LIN_{max}. In algorithm LIN_{all} each node constantly tries to linearize its neighbors according to the *linearize left* and *linearize right* rules in Figure 3. In doing so, *all* possible triples on both sides are proposed to a (hypothetical) scheduler. More formally, in LIN_{all} every node u checks the following actions for every pair of neighbors v and w:

linearize left(v, w):
$(v, w \in u.L \ \wedge \ w < v < u) \ \rightarrow \ e(u, w) := 0, e(v, w) := 1$

linearize right(v, w):
$(v, w \in u.R \ \wedge \ u < v < w) \ \rightarrow \ e(u, w) := 0, e(v, w) := 1$

LIN_{max} is similar to LIN_{all} but instead of proposing all possible triples on each side, LIN_{max} only proposes the triple which is the *furthest* (w.r.t. IDs) on the corresponding side. Concretely, every node $u \in V$ checks the following actions for every pair of neighbors v and w:

linearize left(v, w):
$(v, w \in u.L) \ \wedge \ w < v < u \ \wedge \nexists x \in u.L \setminus \{w\} : x < v)$
$\rightarrow \ e(u, w) := 0, e(v, w) := 1$

linearize right(v, w):
$(v, w \in u.R) \ \wedge \ u < v < w \ \wedge \nexists x \in u.R \setminus \{w\} : x > v)$
$\rightarrow \ e(u, w) := 0, e(v, w) := 1$

Note that both algorithms ensure that in the absence of external changes, connectivity is preserved at all times. Moreover, it can be seen that both algorithms eventually converge to a sorted linear network. In order to analyze the actual convergence time, a model for the parallel execution is needed. Unfortunately, many parallel runtime models studied in the literature are either overly pessimistic in the sense that they can force the algorithm to work serially, or they are too optimistic in the sense that contention or congestion issues are neglected. In [13], a new family of execution models is proposed that distinguishes actions proposed by an algorithm and a more or less adversarial scheduler that selects some of them for parallel execution. Concretely, different "scalable schedulers" can be considered, e.g., a *worst-case scheduler* $\mathcal{S}_{\text{wc}}$: This scheduler must select a maximal *independent set* of enabled actions in each round, but it may do so to enforce a runtime (or work) that is as large as possible. For instance, for the worst case scheduler $\mathcal{S}_{\text{wc}}$ we have the following result.

THEOREM 3.1. *Under a worst-case scheduler $\mathcal{S}_{wc}$, LIN_{max} terminates after $O(n^2)$ work (single linearization steps), where n is the total number of nodes in the system. This is tight in the sense that there are situations where under a worst-case scheduler $\mathcal{S}_{wc}$, LIN_{max} requires $\Omega(n^2)$ rounds. LIN_{all} terminates after $O(n^2 \log n)$ many rounds.*

3.2 Delaunay and Skip Graphs

Motivated by the insights from graph linearization, one may wonder how to design more sophisticated self-stabilizing topologies. A natural idea is to reason about a 2-dimensional linearization variant, e.g., about self-stabilizing *Delaunay graphs*. In [17], it is shown that such a generalization is possible indeed, although the construction and analysis is much more complicated than in the 1-dimensional case, and requires geometric arguments.

Another interesting class of topologies to study from a self-stabilization point of view are skip graphs. Due to their hypercubic structure, skip graphs are an interesting candidate for the design of scalable overlays [2, 14]. In [16], a self-stabilizing variant of a skip graph called SKIP$^+$ is described. In contrast to traditional skip graphs, SKIP$^+$ can be locally checked for the correct structure. Interestingly, one can also show that a single join event (i.e., a new node connects to an arbitrary node in the system) or a single leave event (i.e., a node just leaves without prior notice) can be handled "locally" and with polylogarithmic work, demonstrating that the self-stabilizing algorithm is also useful for the case where the overlay network is already forming the desired topology (which is the standard case in the literature).

4. FURTHER READING

There is a large body of related and relevant literature, and only a small subset can be discussed here. The interested reader can find more complete overviews in the full articles on the different systems.

The classic example of a dynamic distributed system are peer-to-peer networks with open membership, and indeed most of the architectures presented in this talk are designed with peer-to-peer in mind. Protocols such as Pastry [11] and CAN [26] allow for unexpected failures, and it is shown that they remain well-structured after failures occur in certain valid initial states; it is typically not shown how a system can return to the initial state after the membership changes [23]. Moreover, maintenance costs can be high. For example, in CAN, a background stabilization process is used which introduces a constant overhead [1].

Awerbuch and Scheideler have proposed several interesting algorithms to render today's peer-to-peer systems more robust. For example, in [4], searchable concurrent data structures are studied where data elements can be stored on a dynamic set of nodes, e.g., in a peer-to-peer network. Their *Hyperring* data structure has degree $O(\log n)$ and requires $O(\log^3 n)$ work for insert and delete operations; search time and congestion is bounded by $O(\log n)$ with high probability, which improves on alternative structures, e.g., the deterministic *Skipnet* by Harvey and Munro [15]. The PAGODA [8] system allows heterogeneous nodes to join and leave in polylogarithmic work; in some sense, PAGODA can be regraded as the predecessor of SHELL [28]. Finally, the Chameleon system builds upon the DoS-resilient archival system by Awerbuch and Scheideler [5].

Liben-Nowell, Balakrishnan, and Karger [23] have analyzed the evolution of distributed systems in the face of concurrent joins and unexpected departures. They give a lower bound for the rate at which nodes in the *Chord* peer-to-peer system must participate to maintain the system's distributed state. For instance, they show that if churn can be described by a Poisson distribution, a peer which receives fewer than k notifications per *half-life* will be disconnected from the network with probability at least $(1 - 1/(e-1))^k$. The half-life time period is defined as the time which elapses in a network of n live nodes before n additional nodes arrive, or before half of the nodes depart. Their result implies that a successor list of length $\Theta(\log n)$ per peer is sufficient to ensure that a graph stays connected with high probability, as long as $\Omega(\log n)$ rounds pass before $n/2$ peers fail. It is also shown in [23] that a modified version of Chord is within a logarithmic factor of the optimal rate. The authors assume that the half-life is known, and the question of how to learn the correct maintenance rate of the behavior of neighbors is left for future research.

Resilience to *worst-case failures* (for the more challenging Byzantine fault model) has been studied by Fiat, Saia et al. in [12, 27]. The authors introduce a system where a $(1-\varepsilon)$-fraction of peers and data survives the adversarial removal of up to half of all nodes with high probability. However, the failure model is static. Abraham et al. [1] address scalability and resilience to worst-case joins and leaves, and propose a generic overlay emulation approach for graph families such as *hypercubes*, *butterflies*, or *de Bruijn* networks. They focus on maintaining a balanced network rather than on fault-tolerance in the presence of concurrent faults; moreover, whenever a join or leave takes place, the network is given some time to adapt. Concurrent (and asynchronous) worst-case joins and leaves are also considered by Li et al. [22]. The leaving nodes execute an "exit" protocol which does not allow for sudden crashes.

Topological self-stabilization is a relatively young field, and researchers have only started to examine the most simple networks such as line or ring graphs, e.g., [9]. While interesting compilers [6] have been proposed many years ago that allow to render any local algorithm self-stabilizing (see also the recent survey [21]), the overhead when applied to dynamically changing topologies and the implications on randomized algorithms are not well understood yet. The *Iterative Successor Pointer Rewiring Protocol* described in [10] and the *Ring Network* described in [29] organize the nodes in a sorted ring. Aspnes et al. [3] present a self-stabilizing algorithm for overlays for states where the network nodes initially have out-degree 1. In [25], a local-control strategy called *linearization* is presented for converting an arbitrary connected graph into a sorted list; time complexity results for convergence are derived as well, for a simplified model without congestion.

5. CONCLUDING REMARKS

This paper reviewed different notions of robustness as well as mechanisms to ensure availability and functionality in distributed systems with dynamic membership. It has been shown that randomization can help to design dynamic networks with attractive properties or to redundantly store data such that it is hard to block access. We then discussed *deterministic* algorithms to maintain networks under worst-case changes. Unfortunately, these algorithms can only be used from certain configurations and if the degree of dynamics is bounded. This motivated us to study topologies that can be repaired from any connected state.

The material presented in this WRAS talk concentrates on our own work and the selection is highly biased. For a thorough review of related literature, in order to put the contributions into perspective, and for a discussion of the numerous exciting open questions, we refer the reader to the related work sections of the corresponding articles. Similarly, while we have focused on the main results, the detailed algorithms and the analysis can only be covered by the original full papers.

6. REFERENCES

[1] Ittai Abraham, Baruch Awerbuch, Yossi Azar, Yair Bartal, Dahlia Malkhi, and Elan Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proc. 17th Int. Symposium on Parallel and Distributed Processing (IPDPS)*, 2003.

[2] James Aspnes and Gauri Shah. Skip graphs. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[3] James Aspnes and Yinghua Wu. $O(\log n)$-time overlay network construction from graphs with out-degree 1. In *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *LNCS*, pages 286–300, 2007.

[4] Baruch Awerbuch and Christian Scheideler. The hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. 15th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 318–327, 2004.

[5] Baruch Awerbuch and Christian Scheideler. A denial-of-service resistant dht. In *Proc. 21st International Symposium on Distributed Computing (DISC)*, 2007.

[6] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 258–267, 1991.

[7] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A dos-resilient information system for dynamic data management. In *Proc. 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[8] Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark Thober. Pagoda: A dynamic overlay network for routing, data management, and multicasting. In *Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 170–179, 2004.

[9] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list. In *Proc. 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2008.

[10] Curt Cramer and Thomas Fuhrmann. Self-stabilizing ring networks on connected graphs. Technical Report 2005-5, System Architecture Group, University of Karlsruhe, 2005.

[11] Peter Druschel and Antony Rowstron. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[12] A. Fiat and J. Saia. Censorship resistant peer-to-peer content addressable networks. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, 2002.

[13] Dominik Gall, Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *Proc. 9th Latin American Theoretical Informatics Symposium (LATIN)*, 2010.

[14] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 113–126, 2003.

[15] N.J.A. Harvey and J.I. Munro. Deterministic SkipNet. *Inf. Process. Lett.*, 90(4):205–208, 2004.

[16] Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. ACM Symp. on Principles of Distributed Computing (PODC)*, 2009.

[17] Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. A self-stabilizing and local delaunay graph construction. In *Proc. 20th International Symposium on Algorithms and Computation (ISAAC)*, 2009.

[18] Fabian Kuhn, Stefan Schmid, Joest Smit, and Roger Wattenhofer. A blueprint for constructing peer-to-peer systems robust to dynamic worst-case joins and leaves. In *Proc. 14th IEEE International Workshop on Quality of Service (IWQoS)*, 2006.

[19] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *Proc. 4th International Workshop on Peer-To-Peer Systems (IPTPS)*, 2005.

[20] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. Towards worst-case churn resistant peer-to-peer systems. *Journal Distributed Computing (DIST)*, 22(4), 2010.

[21] Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: Self-stabilization on speed. In *Proc. 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2009.

[22] Xiaozhou Li, Jayadev Misra, and C. Greg Plaxton. Active and concurrent topology maintenance. In *Proc. 18th Ann. Conference on Distributed Computing (DISC)*, 2004.

[23] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proc. 21st Annual Symposium on Principles of Distributed Computing (PODC)*, pages 233–242, 2002.

[24] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proc. 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[25] Melih Onus, Andrea Richa, and Christian Scheideler. Linearization: Locally self-stabilizing sorting in graphs. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2007.

[26] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.

[27] J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu. Dynamically fault-tolerant content addressable networks. In *Proc. 1st Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[28] Christian Scheideler and Stefan Schmid. A distributed and oblivious heap. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, 2009.

[29] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology P2P systems. In *Proc. 5th IEEE International Conference on Peer-to-Peer Computing*, pages 39–46, 2005.

[30] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM '01*, 2001. See also `http://www.pdos.lcs.mit.edu/chord/`.