A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn*

Fabian Kuhn, Stefan Schmid, Roger Wattenhofer
{kuhn, schmiste, wattenhofer}@tik.ee.ethz.ch
Computer Engineering and Networks Laboratory (TIK), ETH Zurich, 8092 Zurich, Switzerland

Abstract We present a dynamic distributed hash table where peers may join and leave at any time. Our system tolerates a powerful adversary which has complete visibility of the entire state of the system and can continuously add and remove peers. Our system provides worst-case fault-tolerance, maintaining desirable properties such as a low peer degree and a low network diameter.

1 Introduction

Storing and handling data in an efficient way lie at the heart of any data-driven computing system. Compared to a traditional client/server approach, decentralized peer-to-peer (P2P) systems have the advantage to be more reliable, available, and efficient. P2P systems are based on common desktop machines ("peers"), distributed over a large-scale network such as the Internet. These peers share data (as well as the management of the data) that is conventionally stored on a central server. Usually, peers are under control of individual users who turn their machines on or off at any time. Such peers join and leave the P2P system at high rates ("churn"), a problem that is not existent in orthodox distributed systems. In other words, a P2P system consists of unreliable components only. Nevertheless, the P2P system should provide a reliable and efficient service.

Most P2P systems in the literature are analyzed against an adversary who can crash a functionally bounded number of random peers. After crashing a few peers the system is given sufficient time to recover again. The scheme described in this paper significantly differs from this in two major aspects. First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary which can remove and add a bounded number of peers. The adversary cannot be fooled by any kind of randomness. It can choose which peers to crash and how peers join.[1] Note that we use the term "adversary" to model worst-case behavior. We do not consider Byzantine faults. Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of peers. Instead, the adversary can constantly crash peers while the system is trying to stay alive. Indeed, our system is *never fully repaired* but *always fully functional*. In particular, our system is resilient against an adversary which continuously attacks the "weakest part" of the system. Such an adversary could for example insert a crawler into the P2P system, learn the topology of the system, and then repeatedly crash selected peers, in an attempt to partition the P2P network. Our system counters such an adversary by continuously moving the remaining or newly joining peers towards the sparse areas.

Clearly, we cannot allow our adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $\mathrm{O}(\log n)$ peers, n being the total number of peers currently in the system. This model covers an adversary which repeatedly takes down machines by a distributed denial of service attack, however only a logarithmic number of machines at each point in time. Our algorithm relies on messages being delivered timely, in at most constant time between any pair of operational peers. In distributed computing such a system is called *synchronous*. Note that if nodes are synchronized locally, our algorithm also runs in an asynchronous environment. In this case, the propagation delay of the slowest message defines the notion of time which is needed for the adversarial model.

*Research (in part) supported by the Hasler Stiftung under grant number 1828. For a full version including proofs see the *TIK Report 211* (`http://www.tik.ee.ethz.ch`).

[1]We assume that a joining peer knows a peer which already belongs to the system. This is known as the *bootstrap* problem.

The basic structure of our P2P system is a hypercube. Each peer is part of a distinct hypercube node; each hypercube node consists of $\Theta(\log n)$ peers. Peers have connections to other peers of their hypercube node and to peers of the neighboring hypercube nodes. In the case of joins or leaves, some of the peers have to change to another hypercube node such that up to constant factors, all hypercube nodes own the same number of peers at all times. If the total number of peers grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.

The balancing of peers among the hypercube nodes can be seen as a dynamic token distribution problem [13] on the hypercube. Each node of a graph (hypercube) has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all nodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. Our P2P system 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 peers in the system and to adapt the dimension accordingly.

Based on the described structure, we get a fully scalable, efficient P2P system which tolerates $O(\log n)$ worst-case joins and/or crashes per constant time interval. As in other P2P systems, peers have $O(\log n)$ neighbors, and the usual operations (e.g. search) take time $O(\log n)$. In our view a main contribution of the paper, however, is to propose and study a model which allows for dynamic adversarial churn. We believe that our basic algorithms (dynamic token distribution and information aggregation) can be applied to other P2P topologies, such as butterflies, skip graphs, chordal rings, etc. It can even be used for P2P systems that go beyond distributed hash tables (DHT).

The paper is organized as follows. In Section 2 we discuss relevant related work. Section 3 gives a short description of the model. A detailed discussion of our P2P system is given in Sections 4 and 5.

2 Related Work

A plethora of different overlay networks with various interesting technical properties have been proposed over the last years (e.g. [1, 3, 5, 6, 9, 10, 12, 15, 16, 20, 23]). Due to the nature of P2P systems, fault-tolerance has been a prime issue from the beginning. The systems usually tolerate a large number of random faults. However after crashing a few peers the systems are given sufficient time to recover again. From an experimental point of view, churn has been studied in [17], where practical design tradeoffs in the implementation of existing P2P networks are considered.

Resilience to worst-case failures has been studied by Fiat, Saia et al. in [8, 18]. They propose a system where, w.h.p., $(1-\varepsilon)$-fractions of peers and data survive the adversarial deletion of up to half of all nodes. In contrast to our work the failure model is static. Moreover, if the total number of peers changes by a constant factor, the whole structure has to be rebuilt from scratch.

Scalability and resilience to worst-case joins and leaves has been addressed by Abraham et al. in [2]. The focus lies on maintaining a balanced network rather than on fault-tolerance in the presence of concurrent faults. In contrast to our paper, whenever a join or leave happens, the network has some time to adapt.

The only paper which explicitly treats arbitrarily concurrent worst-case joins and leaves is by Li et al. [11]. In contrast to our work, Li et al. consider a completely asynchronous model where messages can be arbitrarily delayed. The stronger communication model is compensated by a weaker failure model. It is assumed that peers do not crash. Leaving peers execute an appropriate "exit" protocol and do not leave before the system allows this; crashes are not allowed.

3 Model

We consider the *synchronous message passing model*. In each round, each peer can send a message to all its neighbors. Additionally, we have an adversary $\mathcal{A}(J, L, \lambda)$ which may perform J arbitrary joins and and L arbitrary leaves (crashes) in each interval of λ rounds.

We assume that a joining peer π_1 contacts an arbitrary peer π_2 which already belongs to the system; π_2 then triggers the necessary actions for π_1's integration. A peer may be contacted by several joining peers simultaneously. In contrast to other systems where peers have to do some finalizing operations before leaving, we consider the more general case where peers depart or crash without notice.

4 Algorithm

In this section, we describe the maintenance algorithm which maintains the simulated hypercube in the presence of an adversary which constantly adds and removes peers. The goal of the maintenance algorithm is twofold. It guarantees that each node always contains at least one peer which stores the node's data. Further, it adapts the hypercube dimension to the total number of peers in the system.

This is achieved by two basic components. First, we present a dynamic token distribution algorithm for the hypercube. Second, we describe an information aggregation scheme which allows the nodes to simultaneously change the dimension of the hypercube.

4.1 Dynamic Token Distribution

The problem of distributing peers uniformly throughout a hypercube is a special instance of a *token distribution problem*, first introduced by Peleg and Upfal [13]. The problem has its origins in the area of load balancing, where the workload is modeled by a number of *tokens* or jobs of unit size; the main objective is to distribute the total load equally among the processors. Such load balancing problems arise in a number of parallel and distributed applications including job scheduling in operating systems, packet routing, large-scale differential equations and parallel finite element methods. More applications can be found in [19].

Formally, the goal of a token distribution algorithm is to minimize the maximum difference of tokens at any two nodes, denoted by the *discrepancy* ϕ. This problem has been studied intensively; however, most of the research is about the *static variant* of the problem, where given an arbitrary initial token distribution, the goal is to redistribute these tokens uniformly. In the *dynamic variant* on the other hand, the load is dynamic, that is, tokens may arrive and depart *during* the execution of the token distribution algorithm. In our case, peers may join and leave the simulated hypercube at arbitrary times, so the emphasis lies on the dynamic token distribution problem on a d-dimensional hypercube topology.

We use two variants of the token distribution problem: In the *fractional token distribution*, tokens are arbitrarily divisible, whereas in the *integer token distribution* tokens can only move as a whole. In our case, tokens represent peers and are inherently integer. However, it turns out that the study of the fractional model is useful for the analysis of the integer model.

We use a token distribution algorithm which is based on the *dimension exchange method* [7, 14]. Basically, the algorithm cycles continuously over the d dimensions of the hypercube. In step s, where $i = s \mod d$, every node $u := \beta_0 ... \beta_i ... \beta_{d-1}$ having a tokens balances its tokens with its adjacent node in dimension i, $v := \beta_0 ... \overline{\beta_i} ... \beta_{d-1}$, having b tokens, such that both nodes end up with $\frac{a+b}{2}$ tokens in the fractional token distribution. On the other hand, if the tokens are integer, one node is assigned $\lceil \frac{a+b}{2} \rceil$ tokens and the other one gets $\lfloor \frac{a+b}{2} \rfloor$ tokens.

It has been pointed out in [7] that the described algorithm yields a perfect discrepancy $\phi = 0$ after d steps for the static fractional token distribution. In [14], it has been shown that in the worst case, $\phi = d$ after d steps in the static integer token distribution. We can show that if the decision to which node to assign $\lceil \frac{a+b}{2} \rceil$ and to which node to assign $\lfloor \frac{a+b}{2} \rfloor$ tokens is made randomly, the final discrepancy is constant in expectation. However, we do not make use of this because it has no influence on our asymptotic results.

In the following, the dynamic integer token distribution problem is studied, where a "token adversary" $\mathcal{A}(J, L, 1)$ adds at most J and removes at most L tokens at the beginning of each step. In particular, we will show that if the initial distribution is perfect, i.e., $\phi = 0$, our algorithm maintains the invariant $\phi \leq 2J + 2L + d$ at every moment of time.

For the dynamic fractional token distribution, the tokens inserted and deleted at different times can be treated independently and be superposed. Therefore, the following lemma holds.

Lemma 4.1. *For the dynamic fractional token distribution, the number of tokens at a node depends only on the token insertions and deletions of the last d steps and on the total number of tokens in the system.*

We can now bound the discrepancy of the integer token distribution algorithm by comparing it with the fractional problem.

Lemma 4.2. *Let v be a node of the hypercube. Let $\tau_v(t)$ and $\tau_{v,f}(t)$ denote the number of tokens at v for the integer and fractional token distribution algorithms at time t, respectively. We have $\forall t : |\tau_v(t) - \tau_{v,f}(t)| \leq \frac{d}{2}$.*

Lemma 4.3. *In the presence of an adversary $\mathcal{A}(J, L, 1)$, it always holds that the integer discrepancy $\phi \leq 2J + 2L + d$.*

4.2 Information Aggregation

When the total number of peers in the d-dimensional hypercube system exceeds a certain threshold, all nodes $\beta_0 \dots \beta_{d-1}$ have to split into two new nodes $\beta_0 \dots \beta_{d-1}0$ and $\beta_0 \dots \beta_{d-1}1$, yielding a $(d+1)$-dimensional hypercube. Analogously, if the number of peers falls beyond a certain threshold, nodes $\beta_0 \dots \beta_{d-2}0$ and $\beta_0 \dots \beta_{d-2}1$ have to merge their peers into a single node $\beta_0 \dots \beta_{d-2}$, yielding a $(d-1)$-dimensional hypercube. Based on ideas also used in [4, 21, 22], we present an algorithm which provides the same estimated number of peers in the system to all nodes in every step allowing all nodes to split or merge synchronously, that is, in the same step. The description is again made in terms of *tokens* rather than peers.

Assume that in order to compute the total number of tokens in a d-dimensional hypercube, each node $v = \beta_0 ... \beta_{d-1}$ maintains an array $\Gamma_v[0...d]$, where $\Gamma_v[i]$ for $i \in [0, d]$ stores the estimated number of tokens in the sub-cube consisting of the nodes sharing v's prefix $\beta_0 ... \beta_{d-1-i}$. Further, assume that at the beginning of each step, an adversary inserts and removes an arbitrary number of tokens at arbitrary nodes. Each node $v = \beta_0 ... \beta_{d-1-i} ... \beta_{d-1}$ then calculates the new array $\Gamma'_v[0...d]$. For this, v sends $\Gamma_v[i]$ to its adjacent node $u = \beta_0 ... \overline{\beta_{d-1-i}} ... \beta_{d-1}$, for $i \in [0, d-1]$. Then, $\Gamma'_v[0]$ is set to the new number of tokens at v which is the only node with prefix $\beta_0 ... \beta_{d-1}$. For $i \in [1, d]$, the new estimated number of tokens in the prefix domain $\beta_0 ... \beta_{d-1-(i+1)}$ is given by the total number of tokens in the domain $\beta_0 ... \beta_{d-1-i}$ plus the total number of tokens in domain $\beta_0 ... \overline{\beta_{d-1-i}}$ provided by node u, that is, $\Gamma'_v[i+1] := \Gamma_v[i] + \Gamma_u[i]$.

Lemma 4.4. *Consider two arbitrary nodes v_1 and v_2 of the d-dimensional hypercube. Our algorithm guarantees that $\Gamma_{v_1}[d] = \Gamma_{v_2}[d]$ at all times t. Moreover, it holds that this value is the correct total number of tokens in the system at time $t - d$.*

5 Simulated Hypercube

Based on the components presented in the previous sections, both the topology and the maintenance algorithm are now described in detail. In particular, we show that, given an adversary $\mathcal{A}(d+1, d+1, 6)$ which inserts and removes at most $d+1$ peers in any time interval of 6 rounds, 1) the out-degree of every peer is bounded by $\Theta(\log^2 n)$ where n is the total number of peers in the system, 2) the network diameter is bounded by $\Theta(\log n)$, and 3) every node of the simulated hypercube has always at least one peer which stores its data items, so no data item will ever be lost.

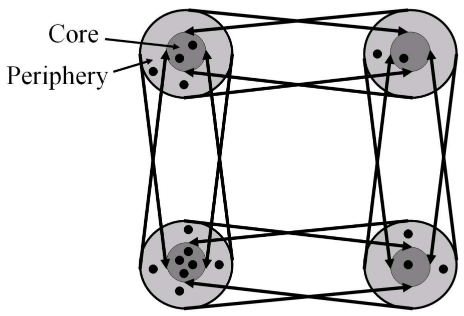

Figure 1: A simulated 2-dimensional hypercube with four nodes, each consisting of a core and a periphery. All peers within the same node are completely connected to each other, and additionally, all peers of a node are connected to all core peers of the neighboring nodes. Only the core peers store data items, while the peripheral peers may move between the nodes to balance biased adversarial changes.

5.1 Topology

We start with a description of the overlay topology. As already mentioned, the peers are organized to simulate a d-dimensional hypercube, where the hypercube's nodes are represented by a group of peers. A data item with identifier id is stored at the node whose identifier matches the first d bits of the hash-value of id.

The peers of each node v are divided into a *core* $\mathcal{C}_v$ of at most $2d + 3$ peers and a *periphery* $\mathcal{P}_v$ consisting of the remaining peers; all peers within the same node are completely connected (*intra-connections*). Moreover, every peer is connected to all *core* peers of the neighboring nodes (*inter-connections*). Figure 1 shows an example for $d = 2$.

The data items belonging to node v are replicated on all core peers, while the peripheral peers are used for the balancing between the nodes according to the peer distribution algorithm and do not store any data items. The partition into core and periphery has the advantage that the peers which move between nodes do not have to replace the data of the old node by the data of the new nodes in most cases.

5.2 6-Round (Maintenance) Algorithm

The *6-round (maintenance) algorithm* maintains the simulated hypercube topology described in the previous section given an adversary $\mathcal{A}(d+1, d+1, 6)$. In particular, it ensures that 1) every node has at least one core peer all the times and hence no data is lost; 2) each node always has between $3d + 10$ and $45d + 86$ peers; 3) only peripheral peers are moved between nodes, thus the unnecessary copying of data is avoided.

In the following, we refer to a complete execution of all six rounds (round 1 – round 6) of the maintenance algorithm as a *phase*. Basically, the 6-round algorithm balances the peers across one dimension in every phase according to the token distribution algorithm as described in Section 4.1; additionally, the total number of peers in the system is computed with respect to an earlier state of the system by the information aggregation algorithm of Section 4.2 to expand or shrink the hypercube if the total number of peers exceeds or falls below a certain threshold. In our system, we use the lower threshold $LT := 8d + 16$ and the upper threshold $UT := 40d + 80$ for the total number of peers *per node on average*.[2]

While peers may join and leave the system at arbitrary times, the 6-round algorithm considers the (accumulated) changes only once per phase. That is, a snapshot of the system is made in round 1; rounds 2 – 6 then ignore the changes that might have happened in the meantime and depend solely on the snapshot at the beginning of the phase.

Round 1: Each node v makes the snapshot of the currently active peers. For this, each peer in v sends a packet with its own ID and the (potentially empty) ID set of its joiners to all adjacent peers *within* v.

Round 2: Based on the snapshot, the core peers of a node v know the total number of peers in the node and send this information to the neighboring core with which they have to balance in this phase (cf. Section 4.1). The cores also exchange the new estimated total number of peers in their domains with the corresponding adjacent cores (cf. Section 4.2). Finally, each peer informs its joiners about the snapshot.

Round 3: Given the snapshot, every peer within a node v can compute the new periphery (snapshot minus old core). This round also prepares the transfer for the peer distribution algorithm across dimension i: The smaller of the two nodes determines the peripheral peers that have to move and sends these IDs to the neighboring core.

Round 4: In this round, the peer distribution algorithm is continued: The core which received the IDs of the new peers sends this information to the periphery. Additionally, it informs the new peers about the neighboring cores, etc.

The dimension reduction is prepared if necessary: If the estimated total number of peers in the system is beyond the threshold, the core peers of a node which will be reduced send their data items plus the identifiers of all their peripheral peers (with respect to the situation *after* the transfer) to the core of their adjacent node in the largest dimension.

Round 5: This round finishes the peer distribution, establishes the new peripheries, and prepares the building of a new core. If the hypercube has to grow in this phase, the nodes start to split, and vice versa if the hypercube is going to shrink.

Given the number of transferred peers, all peers can now compute the new peripheries. Moreover, they can compute the new core: It consists of the peers of the old core which have still been alive in Round 1, plus the $2d + 3 - |\mathcal{C}|$ smallest IDs in the new periphery, where $\mathcal{C}$ is the set of the old core peers which have still been alive in Round 1. The old core then informs all its neighboring nodes (i.e., their old cores) about the new core.

If the hypercube has to grow in this phase, the smallest $2d + 3$ peers in the new periphery of the node that has to be split become the new core of the expanded node, and half of the remaining peripheral peers build its periphery. Moreover, the necessary data items are sent to the core of the expanded node, and the neighboring (old) cores are informed about the IDs of the expanded core.

If the hypercube is about to shrink, all old cores in the lower half of the hypercube (the surviving sub-cube) inform their periphery about the peers arriving from the expanded node and the peers in the expanded node about the new core and its periphery. The data items are copied to the peers as necessary.

Round 6: In this round, the new cores are finally built: The old core forwards the information about the new neighboring cores to the peers joining the core.

Moreover, if the hypercube has been reduced, every peer can now compute the new periphery. If the hypercube has grown, the old core forwards the expanded cores of its neighbors to *all* peers in its expanded node.

[2]Note that since we consider the threshold *on average*, and since these values are provided with a delay of d phases in a d-dimensional hypercube (see Lemma 4.4), the number of peers at an individual node may lie outside $[LT, UT]$.

Theorem 5.1. *Given an adversary $\mathcal{A}(d+1, d+1, 6)$ which inserts and removes at most $d+1$ peers per phase, the described 6-round algorithm ensures that 1) every node always has at least one core peer and hence no data is lost; 2) each node has between $3d+10$ and $45d+86$ peers, yielding a logarithmic network diameter; 3) only peripheral peers are moved between nodes, thus the unnecessary copying of data is avoided.*

In order to enhance clarity, we described a scheme which is as simple as possible. Instead of a complete bipartite graph between adjacent hypercube nodes one could e.g. use a bipartite matching. This reduces the node degree from $\mathrm{O}(\log^2 n)$ to $\mathrm{O}(\log n)$. Apart from better node degrees, all our results still hold up to constant factors.

6 Conclusions

We presented a first distributed hash table which provably tolerates dynamic worst-case joins and leaves. We believe that our approach opens several exciting P2P research challenges. For example: How well perform classic P2P proposals when studied with a dynamic failure model or what is the adversary/efficiency tradeoff when studying dynamic models?

References

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *Proc. 9th Int. Conference on Cooperative Information Systems (CoopIS)*, pages 179–194, 2001.

[2] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov. A Generic Scheme for Building Overlay Networks in Adversarial Scenarios. In *Proc. 17th Int. Symp. on Parallel and Distributed Processing (IPDPS)*, page 40.2, 2003.

[3] I. Abraham, D. Malkhi, and O. Dobzinski. LAND: Stretch (1 + ε) Locality-Aware Networks for DHTs. In *Proc. 15th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 550–559, 2004.

[4] K. Albrecht, R. Arnold, M. Gähwiler, and R. Wattenhofer. Aggregating Information in Peer-to-Peer Systems for Improved Join and Leave. In *4th IEEE Int. Conference on Peer-to-Peer Computing (P2P)*, 2004.

[5] J. Aspnes and G. Shah. Skip Graphs. In *Proc. 14th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 384–393, 2003.

[6] B. Awerbuch and C. Scheideler. The Hyperring: A Low-Congestion Deterministic Data Structure for Distributed Environments. In *Proc. 15th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 318–327, 2004.

[7] G. Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal on Parallel Distributed Computing*, 7:279–301, 1989.

[8] A. Fiat and J. Saia. Censorship Resistant Peer-to-Peer Content Addressable Networks. In *Proc. 13th Symp. on Discrete Algorithms (SODA)*, 2002.

[9] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. 4th USENIX Symp. on Internet Technologies and Systems (USITS)*, 2003.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proc. of ACM ASPLOS*, November 2000.

[11] X. Li, J. Misra, and C. G. Plaxton. Active and Concurrent Topology Maintenance. In *Proc. 18th Ann. Conference on Distributed Computing (DISC)*, 2004.

[12] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. 21st Ann. Symp. on Principles of Distributed Computing (PODC)*, pages 183–192, 2002.

[13] D. Peleg and E. Upfal. The Token Distribution Problem. *SIAM Journal on Computing*, 18(2):229–243, 1989.

[14] C. G. Plaxton. Load Balancing, Selection and Sorting on the Hypercube. In *Proc. 1st Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 64–73, 1989.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proc. 9th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proc. of ACM SIGCOMM 2001*, 2001.

[17] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. USENIX Ann. Technical Conference*, 2004.

[18] 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.

[19] B. A. Shirazi, K. M. Kavi, and A. R. Hurson. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Science Press, 1995.

[20] 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. ACM SIGCOMM Conference*, 2001.

[21] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computing Systems*, 21(2):164–206, 2003.

[22] R. van Renesse and A. Bozdog. Willow: DHT, Aggregation, and Publish/Subscribe in One Protocol. In *Proc. 3rd Int. Workshop on Peer-To-Peer Systems (IPTPS)*, 2004.

[23] B. Y. Zhao, L. Huang, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.