An Adaptive Stabilization Framework for Distributed Hash Tables

Gabriel Ghinita, Yong Meng Teo

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{ghinitag,teoym}@comp.nus.edu.sg

Abstract

Distributed Hash Tables (DHT) algorithms obtain good lookup performance bounds by using deterministic rules to organize peer nodes into an overlay network. To preserve the invariants of the overlay network, DHTs use stabilization procedures that reorganize the topology graph when participating nodes join or fail. Most DHTs use periodic stabilization, in which peers perform stabilization at fixed intervals of time, disregarding the rate of change in overlay topology; this may lead to poor performance and large stabilization-induced communication overhead. We propose a novel adaptive stabilization framework that takes into consideration the continuous evolution in network conditions. Each peer collects statistical data about the network and dynamically adjusts its stabilization rate based on the analysis of the data. The objective of our scheme is to maintain nominal network performance and to minimize the communication overhead of stabilization.

1 Introduction

Distributed Hash Tables (DHT) are structured overlay networks that support large-scale, highly-dynamic distributed computing infrastructures. The paradigm underlying DHT is the virtualization of identifiers of both nodes (peers) and stored items within a single, sparsely populated identifier space [6]. Virtualization is performed using consistent hashing functions [8], such as SHA-1 [2], that achieve implicit load-balancing by uniformly distributing nodes and items over the identifier space. DHT nodes form a distributed data structure with a simple interface consisting of two *primitive operations*: `Put(key,value)` and `Get(key)` [6]. Higher-level services such as distributed storage [5], multicast [14] and publish-subscribe [4] are developed on top of these primitive operations.

At the heart of the `Get/Put` primitives is a *lookup* mechanism that finds the destination node corresponding to the `key` argument. Since every DHT `Get/Put` translates into a key lookup, the efficiency of the lookup algorithm is crucial to the DHT performance.

Multiple DHT protocols have been proposed [3, 13, 16, 17, 18], and they all organize peer routing tables according to *deterministic rules*. These rules have the form of routing table constraints, specified as a relation between the identifier of node n and the identifiers of other peers stored by n in its routing table. Most DHT protocols [3, 16, 17, 18] achieve an upper bound of $O(logN)$ hops lookup performance with $O(logN)$ routing state at each peer, where N is the population size of the network. The logarithmic lookup bound is achieved by using a protocol-specific routing technique, in most cases a variation of hypercube routing. The process of searching for the correct node to store in a routing table entry is denoted by *pinning*.

The topology of DHT overlays changes frequently due to node join, leave and failure events. The process of continuous change in network topology is generically referred to as **churn** [10]. Churn can cause data loss, inconsistent views of data distribution at different peers and incorrectness of routing tables. In this paper, we only consider the effect of churn on the correctness and efficiency of overlay routing. We emphasize that routing underlies every DHT operation, and thus incorrect routing will severely impact DHT functionality.

There are two undesirable effects of churn on DHT routing: *failed lookups* and *increased lookup path length*. A lookup failure occurs in one of the following cases: 1) a lookup request encounters a failed node along its search path, and 2) a lookup request is incorrectly resolved to a node that is not the holder of the searched key. The upper bound on lookup path length holds provided that the set of protocol-specific routing table constraints are satisfied. Since changes in overlay topol-

1-4244-0054-6/06/$20.00

ogy may result in the violation of these constraints, the path length of a lookup message may increase, possibly deteriorating to $O(N)$.

To counter the undesirable effects of churn on routing, DHTs employ the use of overlay graph maintenance procedures - *stabilization routines*. The objective of **stabilization** is to keep the routing information of each overlay node consistent with the permanently changing overlay topology. Each peer verifies if the nodes stored in its routing table are alive, and if its routing table invariants are satisfied. Currently, the most widely used stabilization technique is *periodic stabilization*, employed by DHTs such as CAN [13], Chord [17], Pastry [16], etc. With periodic stabilization, each node invokes corrective routines at fixed intervals of time. At each invocation, a number of messages proportional to the size of the node's routing table may be generated.

Periodic stabilization does not take into consideration the degree of dynamism of the overlay topology, but instead uses a fixed stabilization timer value, independent of network conditions. As a measure of topology dynamism, we use the **churn rate** [10, 11], defined as the cumulative rate of node join and failure events that occur in the network per unit time, and similar to other stabilization-related research [11, 12], we make no distinction between node failure and graceful departure. If periodic stabilization execution rate is high, changes in the system can be quickly detected, but at the disadvantage of increased communication overhead. On the other hand, if the stabilization rate is low and the churn rate is high, routing tables become inaccurate and the DHT performance deteriorates.

The understanding of stabilization in DHT has been an active research area in recent years. The concept of churn was first introduced in [11] as the process of continuous change in network topology, and lower bounds were formulated on the maintenance rate required to avoid network disconnection. In [10], the authors introduce a methodology for evaluating the performance and cost of stabilization, and present a comparative performance analysis of several DHT protocols. In [9], a theoretical study of churn is presented, based on a master-equation approach inspired from statistical mechanics. In [15], the authors study the churn patterns in deployed file-sharing networks and explore several design tradeoffs that increase the ability of DHT systems to handle high churn rates.

To alleviate the shortcomings of periodic stabilization, alternative stabilization techniques have been proposed. *Correction-on-use/correction-on-change*, used by DKS [3], employs a stabilization protocol that is embedded in the DHT lookup protocol, and uses lookup messages to correct routing state. It achieves a good performance-cost compromise, but due to its design, its applicability is restricted to the DKS DHT. The necessity to develop adaptive stabilization techniques that can dynamically respond to varying network conditions has been acknowledged by the research community [6]. However, steps taken so far in this direction [12] have a narrow scope and lack a systematic approach.

In this paper, we present a framework for adaptive stabilization that can be used to instantiate adaptive stabilization techniques for different DHT protocols. To prove the effectiveness of our method, we show how it can be applied to a particular DHT protocol, namely Chord [17], and highlight its advantages over periodic stabilization in terms of both lookup performance and communication cost.

The rest of the paper is organized as follows: in Section 2, we discuss in more detail how periodic stabilization works and we analyze its limitations. In Section 3, we introduce our adaptive stabilization framework for DHT infrastructures. Section 4 presents an instantiation of our framework for Chord DHT. In Section 5, we present a comparative performance analysis of our adaptive stabilization scheme against periodic stabilization. Our concluding remarks are in Section 6.

2 Limitations of Periodic Stabilization

Studies of real P2P system traces [7] show that churn rate varies over time, with occasional peaks. We conducted a set of simulation experiments to capture the behavior of periodic stabilization under diverse network conditions. We chose Chord DHT for our study and used a modified version of the *p2psim* [1] P2P protocol simulation tool. In our simulations, we assume a reliable communication network, in order to isolate the effect of churn on routing. We distinguish between lookup operations generated as a result of user key lookup requests, and lookups performed by the stabilization routines in order to correct routing tables. We use the following performance evaluation metrics:

- **lookup failure:** the percentage of user-generated key lookup operations that fail or return an incorrect result
- **average lookup path length:** the average hop count of successful user-generated key lookup operations
- **communication overhead:** the percentage of stabilization-generated messages over the user-generated messages (user-generated lookups plus node join messages). Each hop along the path of a lookup (an end-to-end message at the underly-

ing communication network layer) is counted as a separate message.

Chord uses a one-dimensional, *m-bit* circular identifier space - the Chord ring. An item with key `key` is stored at *succ*(*key*) - the node that immediately follows `key` on the Chord ring. We denote by $n.id$ the identifier of node n. Each node n maintains a routing table composed of three sections [17]:

- one *successor* and one *predecessor* pointer that point to the node that immediately follows and the node that precedes n on the identifier ring, respectively
- a *successor list* with pointers to the first r consecutive successors of n on the Chord ring
- a *finger table* with m pointers to nodes that are situated at "power-of-2" distances from n: $F = \{f_i | f_i = succ(n.id + 2^i), i = 0, 1, .., m-1\}$.

Chord uses a different stabilization timer for each section of the routing table, in order to factor the relative importance of different pointers in the lookup process [10]. We denote the stabilization rate in Chord with a $s/sl/f$ tuple, where each field represents the time interval between two consecutive stabilization invocations for each routing table section, respectively: a 1/5/10 stabilization rate, for instance, means that the successor and predecessor are checked every second, the successor list every 5 seconds, and the nodes in the finger table every 10 seconds.

We simulated a set of "half-life" scenarios [11] that are commonly-used to study the behavior of DHT under churn: a network that doubles in size, and a network that halves in size. We varied both *stabilization rate* and *churn rate* to capture their relative effect on both lookup failure and communication overhead. We used three different stabilization rates, $S1 = 1/3/10$, $S2 = 3/5/20$ and $S3 = 5/10/30$ corresponding to high, moderate and low stabilization rate, respectively. For the network doubling case, we started with a 500 node network and scheduled 500 node join events, according to a Poisson process with the mean join rate of $1/sec$, $2/sec$ and $5/sec$, corresponding to low, moderate and high churn rates, respectively. For the network halving case, we started with a 1000 node overlay and scheduled 500 node failure events, according to a Poisson process with the same average rates. We considered a user-generated lookup workload of 0.33 *lookups/sec/node*. To isolate the effect of incorrect routing on lookup performance, we only allow one attempt for each key lookup: the lookup initiator node does not perform a lookup retry in case of a failed lookup.

Table 1 summarizes our results. We focus on the network halving experiments, which have a more disruptive effect on lookup failure. For the low churn rate of $1/sec$, $S1$ achieves 3.2% lookup failure, at the expense of 457% communication overhead; translated into total number of messages, this corresponds to a total of 1.3 million stabilization-generated messages, over a period of only 500 seconds. Even at this high stabilization rate, an increase in churn rate to $5/sec$ can cause considerable lookup failure, up to 8.7%. For the low stabilization rate $S3$, the communication overhead decreases to around 150%, but at the expense of 6.5% lookup failure for the low $1/sec$ churn rate, and 23.7% lookup failure for the high $5/sec$ churn rate. The measured average lookup path length for successful lookup operations does not exhibit a significant increase, rising to at most $0.58 \log N$, compared to the ideal value of $0.5 \log N$ [17]. Our conclusion is that the most impairing consequence of churn is lookup failure; those lookup operations that succeed, do so in a number of hops close to the ideal value, due to the uniform distribution of node identifiers on the Chord ring.

Churn Rate	Stab. Rate	Lookup Failure %		Comm. Overhead %	
		Double	Halve	Double	Halve
1/sec	S1	0.5	3.2	421	457
	S2	1.1	4.9	211	203
	S3	1.7	6.5	135	139
2/sec	S1	0.8	5.1	414	462
	S2	1.9	9.2	208	205
	S3	2.8	12.3	143	151
5/sec	S1	1.8	8.7	407	445
	S2	2.7	12.7	218	209
	S3	3.6	23.7	154	154

Table 1. Periodic Stabilization

We conclude from our experiments that it is not possible to achieve both low lookup failure and low communication overhead with a fixed stabilization rate. To obtain low lookup failure at high churn rates, a high stabilization rate must be used, but at the cost of high communication overhead incurred during periods of low churn rate.

3 Adaptive Stabilization Framework

We propose an adaptive stabilization framework in which peers estimate the overlay topology dynamism, build a stochastic network model and use statistical analysis to adjust stabilization rate, such that QoS requirements are satisfied. The two components of churn are treated independently: node join is modeled as a Poisson process with rate λ, while node failure is modeled by an exponential distribution with rate μ, i.e. expected node lifetime is $1/\mu$. The parameters λ and μ are computed dynamically, based on estimations of network conditions. Each peer n calculates the probability of its routing table to become incorrect, and exe-

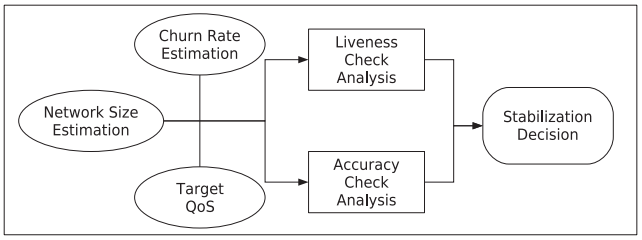


Figure 1. Adaptive Stabilization Framework

cutes the stabilization procedure when this probability exceeds predefined thresholds.

A routing table entry can become incorrect in two situations: 1) the node stored in that entry failed and 2) the node stored in that entry no longer fulfills a constraint imposed by the DHT protocol, due to the joining of new nodes. We identify two distinct operations required to correct these two situations: *liveness check* and *accuracy check*. A liveness check operation (consisting of a ping request-reply) has a cost of $O(1)$ hops, while an accuracy check (consisting of a key lookup) usually requires $O(logN)$ hops. We split DHT stabilization into two distinct processes, liveness check and accuracy check, with the advantage that the effects of node join and failure are dealt with separately. This way, the network can better adapt to scenarios with different compositions of join and failure workloads. Moreover, the cost of stabilization can be more effectively controlled, due to the considerable difference in cost between liveness and accuracy checks. The stabilization decision is taken separately for each single pointer, as different pointers may have different weights in the lookup forwarding process.

An important advantage is that our adaptive stabilization framework is general, and can be used to instantiate stabilization techniques for different DHT protocols. In section 4, we illustrate how an instantiation of our framework can be obtained for Chord.

3.1 Adaptive Stabilization Algorithm

The functionality of our framework is summarized in Figure 1: using an estimation of dynamic network parameters, such as churn rate and network size, and a target QoS requirement, such as maximum allowed *lookup failure*, a peer node locally performs the *liveness check analysis* and *accuracy check analysis* for each routing pointer p it keeps. The analysis yields two *stabilization parameters*:

- P^p_{tout} - the probability of forwarding a lookup message to a dead node pointed by p, and
- P^p_{inacc} - the probability that the peer pointed to by p no longer abides protocol-specific constraints

```
procedure Adaptive_Stabilization
    routine stabilize
        for each p in routing_table
            estimate probability P^p_inacc ;
            if ( P^p_inacc > P^Thr_inacc)
                re-pin p ;
            else
                estimate probability P^p_tout ;
                if ( P^p_tout > P^Thr_tout)
                    ping p ;
    end routine
    begin
        register_notifier(TimerEvent, stabilize) ;
        register_notifier(ExternalEvent, stabilize) ;
    end
end procedure
```

Figure 2. Adaptive Stabilization Pseudocode

If these parameters reach certain thresholds, pointer p is refreshed (ping), or respectively re-pinned.

The algorithm executed by each peer comprises of both an asynchronous and a synchronous component, with actions triggered both by external events and by an internal clock. A message received from another peer, or the detection of the failure of another node represent examples of an asynchronous *ExternalEvent*. Such an event can determine node n to update the statistics it keeps about the network, to recompute the stabilization parameters by using liveness and accuracy analysis, and to re-evaluate the stabilization decision.

The stabilization parameters P^p_{tout} and P^p_{inacc} are dependent of the age of pointer p, and therefore their values can change even during periods when no external event occurs. An internal clock generates a *TimerEvent* that triggers the re-calculation of P_{tout} and P_{inacc}. Note that unlike in the case of periodic stabilization, where a timer expiry will generate unconditional network traffic, in the case of adaptive stabilization the timer is used only to update local state, by local computation, and no network traffic is generated unless a change in the estimation of network conditions dictates it. Computational cost is far less expensive than communication cost, and thus the adaptive stabilization timer can be set as fine-grained as desired, without paying a penalty in communication overhead.

Figure 2 shows the pseudocode of the algorithm executed by each peer. The `stabilize` routine, executed each time an event occurs, computes network statistics and determines the P^p_{tout} and P^p_{inacc} parameters for each routing table entry. If the values of P^p_{inacc} and P^p_{tout} exceed thresholds P^{Thr}_{inacc} and P^{Thr}_{tout}, pointer p is re-pinned or refreshed, respectively. In the rest of this section, we describe how P^p_{tout} and P^p_{inacc} are computed, how the thresholds are chosen, and how the estimations of network size, node join and node failure rate are obtained.

3.2 Liveness Check

During the lookup process, node n forwards a lookup message to the node in its routing table that is the next hop for the given destination key, say node p. If p is not alive, the lookup message is lost. The initiator node of the lookup will timeout and conclude that something is wrong along the lookup path. The application layer of the initiating node will decide whether to retry or not, and how many retry attempts will be issued.

In order to maintain good network performance, we must minimize the probability P^p_{tout} for any node n in the network to forward a message to a dead node p in its routing table. Given a random destination key of the lookup, P^p_{tout} is the product of the probabilities of two independent events: P^p_{fwd}, the probability that node n forwards a message to p, and P^p_{dead}, the probability that node p is dead. Thus,

$$P^p_{tout} = P^p_{fwd} \times P^p_{dead} \quad (1)$$

P^p_{fwd} depends on the DHT lookup algorithm used and the identifier of node p. Distinct pointers are used to forward lookups directed to disjoint partitions of the identifier space. The formulation of P^p_{fwd} must take into account the details of the DHT protocol-specific lookup process. In Section 4 we will show how P^p_{fwd} is computed in Chord.

Consider node n that stores a pointer to node p, which was last known to be alive at time T^p_s - the time n performed the last liveness check for p. In our churn model, nodes fail according to an exponential distribution with rate parameter μ; the probability of node p being dead at current time t is

$$P^p_{dead}(t) = 1 - e^{-\mu(t-T^p_s)} \quad (2)$$

Substituting the formulations of P^p_{fwd} and P^p_{dead} in (1), we can estimate the probability of timeout (P_{tout}) and schedule the next liveness check operation for node p such that P_{tout} does not exceed threshold P^{Thr}_{tout}. The threshold must be in agreement with the target lookup failure probability P_f, that is specified as an input to our framework. Since for most DHT protocols a lookup path consists of $\log_2 N/2$ hops on average [6], the per-pointer threshold value can be set to

$$P^{Thr}_{tout} = P_f{}^{\frac{2}{\log_2 N}} \quad (3)$$

where the forwarding timeouts along the lookup path are assumed to be independent events.

3.3 Accuracy Check

There are two negative consequences of pointer inaccuracy: *incorrect lookups* and *increased lookup path length*. When a new node n joins the network, it takes an amount of time until this event is propagated to other peers that should point to n, and thus it is possible for a lookup operation to skip node n, rendering its stored items inaccessible.

The DHT lookup hop-count upper bound, logarithmic in the size of the network, holds only as long as routing table invariants are satisfied. The invariants guarantee that the remaining distance to destination is reduced at each hop, such that the lookup is completed in $O(logN)$ hops. If pointers are inaccurate, the distance to destination reduced at each hop is smaller than nominal, and the number of hops to destination increases, in the worst case degenerating to $O(N)$.

Since routing table constraints are specified by deterministic rules, the accuracy of pointer p maintained by node n can only be affected by nodes that join in a well-determined region of the identifier space. In most DHT protocols, there is an ideal theoretical value for each pointer p. Based on this observation, node n can estimate what is the probability that pointer p is still accurate by comparing the current value of p to its ideal value, and by estimating the probability of a new node joining in the gap between the actual and ideal value of p.

Modeling the joining process of new nodes as a Poisson process with rate λ, and assuming that newly-joined nodes are uniformly distributed over the identifier space, then if pointer p was known to be accurate at the time it was pinned T^p_{pin}, the probability of it becoming inaccurate at current time t is given by

$$P^p_{inacc} = \frac{distance(p_{current}, p_{ideal})}{id\_size} \times \lambda(t - T^p_{pin}) \quad (4)$$

where id_size is the size of the identifier space. As opposed to the P_{tout} threshold in (3), which depends only on the node life/death statistical model used, the P_{inacc} threshold is dependent on the DHT protocol used, since the routing table accuracy constraints are protocol-dependent. In Section 4 we will describe a particular example of this formulation for Chord DHT.

3.4 Gathering Network Statistics

In Sections 3.2 and 3.3 we have used estimates of global network parameters such as μ, λ and network size N. In this section, we show how each peer can obtain an accurate estimation of these parameters in a completely decentralized manner. Each peer n maintains for each pointer p in its routing table three timestamps: the time p was pinned - T^p_{pin}, the time p was last checked for liveness - T^p_s, and the time when the node pointed by p joined the overlay - T^p_{join}.

Node Failure Rate. Each peer maintains a history of node failure events it has observed, in a manner similar to [12]. The history covers a time range T_{hist}

equal to the current time minus the time of the oldest failure event recorded in the history. We denote by F the number of events in the history and by $rsize$ the size of the routing table. From equation (2), we obtain the estimation of the failure rate as

$$\mu = -ln\left(1 - \frac{F}{rsize}\right) \times \frac{1}{T_{hist}} \quad (5)$$

The size of the history determines how fast a node reacts to changes in node failure rate: a small size can determine node n to overreact each time a new failure is recorded, while a large history size can be biased by earlier periods of time with no failure events, and may be slow to react to sudden churn rate changes. In our implementation, we use a history size of 25% of the routing table size, which we have found to be accurate within 17% of the real value of μ.

Node Join Rate. Peer n estimates node join rate based on the T^p_{join} of each of its pointers p. An increase in node join rate will be reflected by a decrease in the average age of nodes in the routing table. To obtain a good reaction time to sudden increases in join rate, we use only the ages of the youngest 25% of the nodes in the routing table (this reduces the bias of a small number of nodes with very old ages). Let *Ages* be the array of all pointer ages sorted in increasing order; then the estimation of global node join rate is

$$\lambda = \frac{N}{4} \times \frac{1}{Ages[rsize/4]} \quad (6)$$

Using this method, we are able to obtain an estimate of λ accurate within 22% of its real value.

Network Size. Each overlay node stores in its routing table the identifiers and addresses of other peers. In most DHT protocols, each node keeps a separate list with its immediate neighbors in the identifier space: in Chord, this is the *successor list*, in Pastry the *neighborhood set*, etc. If we compute the average distance between these nodes, we can obtain a good estimation of the network population size by dividing the identifier space size to the average inter-node distance.

Furthermore, in the case of some DHT protocols, each pointer in node n's routing table has a theoretical ideal value, relative to the identifier of n. By evaluating the distance between a pointer's ideal value and its actual value, we obtain a good estimation of the density of nodes in the identifier space, which in turn can help us estimate the network size.

4 Adaptive Stabilization in Chord

In this section, we show how our proposed framework applies to a specific DHT protocol: Chord DHT [17]. In Chord, a lookup message for key `key` is forwarded hop-by-hop to its destination $succ(key)$. Each hop n chooses as the next hop $next_n(key)$ its farthest finger which does not overshoot `key`. We assume that *recursive* routing is used: the lookup initiator n_i delegates the lookup task to $p_1 = next_{n_i}(key)$, p_1 in turn delegates to $p_2 = next_{p_1}(key)$ and so forth until $p_{lasthop}$ is reached, such that the successor of $p_{lasthop}$ is $succ(key)$.

4.1 Liveness Check

To perform liveness check, we need to determine the value of P^p_{fwd} in equation (1). We denote by P^i_{fwd} the forwarding probability for i^{th} finger (shorthand for $P^{f_i}_{fwd}$). In an idealized Chord ring, where each finger f_i points to a node at a distance of exactly 2^i, and lookup keys are randomly distributed, all pointers are used with equal frequency. However, in a real *m-bit* Chord ring where node identifiers are randomly distributed, there is a bias of $B = \frac{2^m}{2N}$ on average between the ideal and the actual value of f_i. For this reason, P^i_{fwd} is not constant over the set $\{f_i\}$; instead, it depends on the sparsity of nodes in the identifier space, which we define as

$$H = \log_2 \frac{2^m}{N} = m - \log_2 N \quad (7)$$

Assuming node identifiers and search keys are randomly distributed over the identifier space, an approximative formulation of P^i_{fwd} (we omit details due to space limitations) is given by

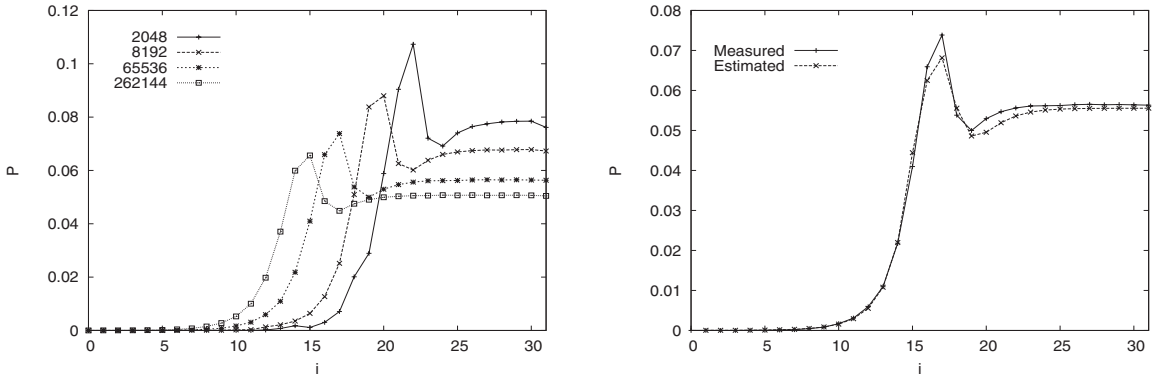
$$P^i_{fwd} = \begin{cases} \frac{2^{i-H}}{\log N}, & 0 \le i \le H \\ \frac{1}{2+\log N}\left(\frac{2^{i+1-H}}{2^{2(i-H)}+1} - \right. & \\ \left. \frac{2^{i-H-2}}{2^{2(i-H-2)}+1} + 1\right), & H < i < m \end{cases} \quad (8)$$

Figure 3(a) shows the plot of P^i_{fwd} for a set of simulated Chord networks with different population sizes. Figure 3(b) illustrates the correlation between the measured and estimated values of P^i_{fwd}. Equation (8), which we use in our framework implementation, has been validated for a large set of network and identifier space sizes.

4.2 Accuracy Check

Inaccuracy of pointers in Chord can cause both incorrect lookups and increased lookup path length. An *incorrect lookup* occurs when the last hop $p_{lasthop}$ of a lookup for `key` wrongfully identifies its successor as $succ(key)$, due to the fact that a new node has joined between $p_{lasthop}$ and its successor.

If finger pointers become inaccurate, the remaining distance to destination of a lookup request will no longer halve at each intermediate node. The distance



(a) Measured forwarding probability for Chord fingers: 32-bit identifier space and varying network size

(b) Measured vs estimated forwarding probability for Chord fingers: 32-bit identifier space and 65536 nodes

Figure 3. Analysis of Chord Fingers Forwarding Probability

gained at each hop may become too small, requiring more than $O(logN)$ hops for lookup completion.

To prevent incorrect lookups, we must ensure that the successor pointer of each node n is accurate, i.e. points to the correct successor of n on the Chord ring. Using the estimation of node join rate λ, each node n evaluates the probability of a new node joining between n and n's successor s as:

$$P_{inacc}^{succ} = \frac{distance(n.id, s.id)}{2^m} \lambda(t - T_{succ}) \qquad (9)$$

where T_{succ} is the time of the last accuracy check for n's successor. Given a target lookup failure of maximum P_f, the time interval required between two consecutive successor checks is obtained from the condition $P_{inacc}^{succ} < P_f$.

To prevent an increase in lookup path length, each node n evaluates for each of its fingers f_i the probability that f_i does no longer fulfill the condition $f_i = succ(n.id + 2^i)$. Given the current values for the set of fingers f_i, and considering the ideal value of $f_i = n.id + 2^i$, we can formulate the probability for a finger to be inaccurate at current time t as

$$P_{inacc}^{i} = \frac{f_i.id - n.id - 2^i}{2^m} \lambda(t - T_{pin}^{i}) \qquad (10)$$

where λ is the estimated join rate and T_{pin}^{i} is the time when f_i was last pinned.

4.3 Gathering Network Statistics

The **Node Failure Rate** and **Join Rate** estimation procedures described in 3.4 are independent of DHT protocol, and can be used without further modification for Chord. To estimate overlay population size, we compute the average inter-node distance between the successive nodes starting with the predecessor and ending with the farthest successor in the successor list. The estimated population size is the size of the identifier space divided by the inter-node distance. During our experiments, we have found this estimation to be accurate within 15% of the real network size.

5 Performance Evaluation

In this section, we evaluate using simulation the performance of our proposed adaptive stabilization technique for Chord DHT. In Sections 5.1 and 5.2 we compare adaptive stabilization (AS) with periodic stabilization (PS) for constant and variable churn rate, respectively. In Section 5.3 we analyze the performance-cost tradeoff that can be achieved with adaptive stabilization.

5.1 Constant Churn Rate

To evaluate the performance of AS at constant churn rate, we revisit the "half-life" scenarios described in Section 2. For AS, we have chosen the target lookup failure P_f to be the same as the lookup failure obtained by the high $S1 = 1/3/10$ stabilization rate at the low $1/sec$ churn rate. We set P_f to 1% for the network doubling and to 3% for the network halving case. The average RTT between peer nodes is 200ms. We consider the same low, moderate and high churn rates as in Section 2, of 1, 2 and 5/sec respectively.

Table 2 summarizes the AS results in comparison with the PS results obtained in Section 2. For network doubling, AS achieves the target lookup failure at all three node join rates, with a slightly 0.1% over the nominal P_f for the 5/sec join rate. The communication overhead of AS is three times and 1.4 times lower than $S1$ for low and moderate node join rates, respectively. For high join rate, AS still meets the target P_f, with an increase in cost of 20% compared to $S1$, but with an improvement by a factor of 1.6 in lookup failure over $S1$. For low and moderate node join rate, the cost of

Churn Rate	Stab. Rate	Lookup Failure %		Comm. Overhead %	
		Double	Halve	Double	Halve
1/sec	S1	0.5	3.2	421	457
	S2	1.1	4.9	211	203
	S3	1.7	6.5	135	139
	AS	0.9	2.9	141	142
2/sec	S1	0.8	5.1	414	462
	S2	1.9	9.2	208	205
	S3	2.8	12.3	143	151
	AS	0.9	3.1	296	305
5/sec	S1	1.8	8.7	407	445
	S2	2.7	12.7	218	209
	S3	3.6	23.7	154	154
	AS	1.1	3.4	489	552

Table 2. AS vs PS: Constant Churn Rate

AS is comparable with that of $S3$ and $S2$ respectively, but with an improved lookup failure by a factor ranging from 1.9 to 2.1.

For network halving, at low node failure rate, AS matches the lookup failure of $S1$ but at roughly the same cost as $S3$, which is only a third of $S1$'s cost. Translated into number of messages, AS generates only 0.4 million messages compared to $S1$'s 1.3 million in a time window of 500 seconds. For moderate node failure rate, AS still meets the target P_f of 3% with a cost lower than $S1$'s cost by a factor of 1.5, while $S1$ only manages a lookup failure larger by a factor of 1.7 than AS. For high node failure rate, AS misses the target P_f but only by a small margin of 0.4%. However, the 3.4% lookup failure is still lower by a factor of 2.6 compared to $S1$, and at only a 24% increase in cost compared to $S1$. The reason that AS does not achieve the target P_f is the latency of the communication network. The failure of a node can not be detected in a shorter time than the RTT, and for this reason, at high churn rate, it may be impossible to detect failures that occur more often than the average RTT of 200ms, regardless of the stabilization technique used.

As in the case of PS in Section 2, we have found that with AS the effect of churn on average lookup path length is not significant, with a path length increase from the nominal $0.5 \log N$ to $0.56 \log N$.

The strength of AS resides in its ability to self-tune according to changing network conditions, so we expect its full potential to be exploited at variable churn rates. However, AS has one important advantage over PS even for constant churn rates: its ability to determine a suitable stabilization rate *on-the-fly*. Even if for a constant churn rate there exists an ideal PS instance that can perform well, determining that particular combination of parameters is a difficult task, and can only be done through a trial-and-error process, requiring the churn rate to be known in advance. With AS, nominal network performance can be achieved without any prior knowledge of node join/failure rate, and in most cases at a lower cost than PS.

5.2 Variable Churn Rate

We consider a network with two different steady-state regimes, each characterized by a low, but constant churn rate. There is a significant difference in network size between the two states. The transition from one state to the other produces abruptly, with nodes that join/fail at rates much higher than the churn rates in the steady states. This model can be used to represent a corporate network, for instance, with two "office-hours" and "after-working-hours" steady states, and two high-churn "peak-hour" periods.

We consider a 500-node initial bootstrap network, corresponding to the first steady-state; 2500 new nodes join the network at a rate of 3/sec and leave at the same rate 6 hrs later. During the 6 hrs period, corresponding to the second steady-state, peers abide an on-off pattern of being connected for an interval of time exponentially distributed with mean of 60 minutes, and disconnected for an interval of time exponentially distributed with mean of 30 minutes. The average size of the network in the second steady-state is 2200 nodes. The target lookup failure P_f for AS is set to 2%.

Figure 4 shows the evolution in time of the churn rate, lookup failure and stabilization cost, measured in number of messages/node/second. All metrics are averaged over 200 seconds intervals. Figure 4(a) shows the churn pattern, with the initial surge consisting mainly of join events, and the final surge of mainly failure events. Figure 4(b) shows the evolution of lookup failure over time: with both $S1$ and $S2$ PS rates, the lookup failure increases sharply, to 3% and 8% respectively for the mainly-join workload; for the mainly-fail workload, lookup failure increases even further, to 7.5% and 12% respectively. AS, tuned for a target lookup failure of 2%, achieves a 1.8% and 2.2% lookup failure for the node join and failure cases, better than both $S1$ and $S2$. The low PS rate $S3$ caused severe network disconnection, and was excluded from our results.

Figure 4(c) shows the stabilization cost per-node. The communication cost of $S1$ and $S2$ remains almost constant over time: the cost decreases slightly during the join process, as new nodes generate mostly join lookup messages, which are not considered as part of the stabilization traffic. For the fail process, the number of stabilization messages increases slightly, due to the numerous timeout-triggered stabilization executions. In the case of AS, we observe how the rate of stabilization adjusts to the churn pattern, with increases

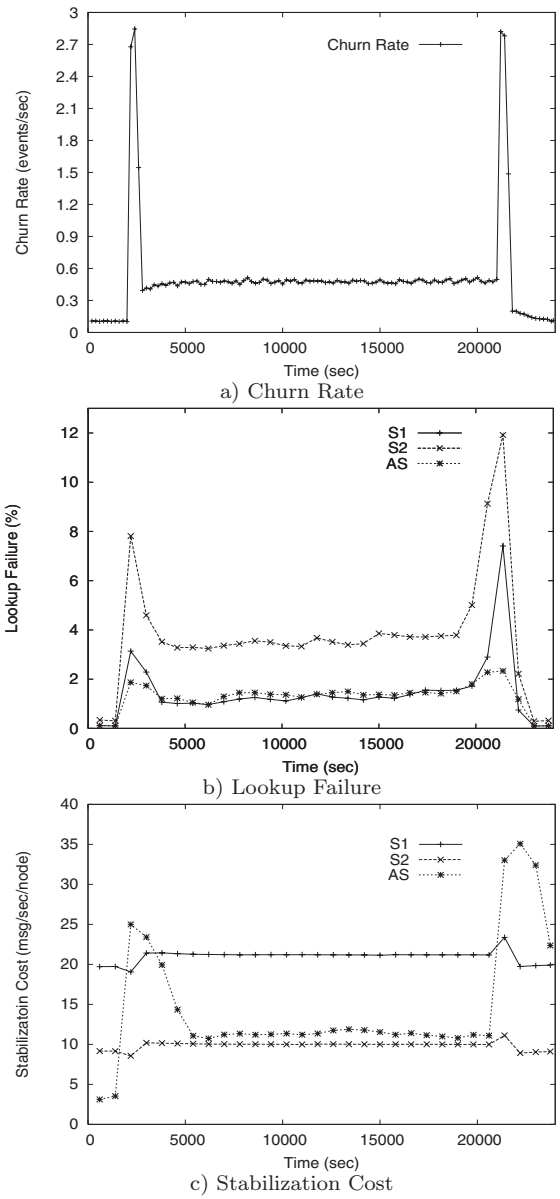


Figure 4. AS vs PS: Variable Churn Rate

during periods of high churn rate. Once the churn rate lowers, AS reacts and decreases the stabilization rate.

Over the entire simulated period, $S1$ generates a 420% communication overhead and achieves a peak lookup failure of 7.5%, while AS reduces the peak lookup failure by a factor of 3.4, to 2.2%, with an overall communication overhead of only 172%. For comparison, $S2$ generates a 154% communication overhead, but only achieves a modest lookup failure, larger by a factor of 5.4 than AS.

AS achieves both low lookup failure and reduced overall communication overhead by adjusting the stabilization rate according to the estimated churn rate. For periods of high churn, AS may generate more overhead than PS, but that overhead is needed to maintain nominal network performance. On the other hand, AS lowers the overall stabilization cost by generating less traffic during periods of low churn rate.

5.3 Performance-cost Tradeoff

During high churn rate periods, AS attempts to maintain lookup failure below the P_f threshold by increasing the stabilization rate. We evaluate the dependence between the communication overhead of stabilization and the target lookup failure P_f, in order to determine what is a good tradeoff in practice between lookup performance and stabilization cost.

We consider a fail-only workload, which has a more disruptive effect on lookup performance. First, we determine the theoretical dependence between P_f and stabilization cost, for a fixed node failure workload with rate μ. For simplification, we disregard finger forwarding probability, and consider that $P_{fwd} = 1$ for each finger, which corresponds to an upper bound for the traffic generated by AS in practice, with $P_{fwd} < 1$. The interval T between two consecutive liveness checks of the same pointer is given by

$$1 - e^{-\mu T} = 1 - (1 - P_f)^{\frac{2}{\log_2 N}} \tag{11}$$

Considering each liveness check (a ping request/reply) as a single message, the number of messages generated per node/per pointer/per time unit interval is

$$\#msg = 1/T = -\frac{\mu \log_2 N}{2 \ln(1 - P_f)} \tag{12}$$

We simulated a 5000-node network that halves in size with a node failure rate of 10/sec. The average RTT between nodes is 1000ms. Figure 5 shows the cost of stabilization in *messages/node/pointer/sec* for the theoretical estimation in (12) and the experimental measurement. As P_f approaches zero, the denominator in (12) will also approach zero. To safeguard our implementation against an undesirable traffic spike, we limit the interval between successive stabilizations of the same pointer to the RTT-value. In the performance-cost tradeoff experiment we have set an RTT value of 1000ms, and we can observe that the cost is limited at 1 *msg/node/pointer/sec*.

AS manages to achieve in practice a performance-cost tradeoff as good as the theoretical estimation given by (12). In those cases where the physical RTT limitation of the underlying communication network prevents AS (as well as any other stabilization technique) to achieve the target lookup failure at high churn rates, an upper-bound on the stabilization rate is enforced, in order to avoid a surge in stabilization traffic that could cause network congestion.

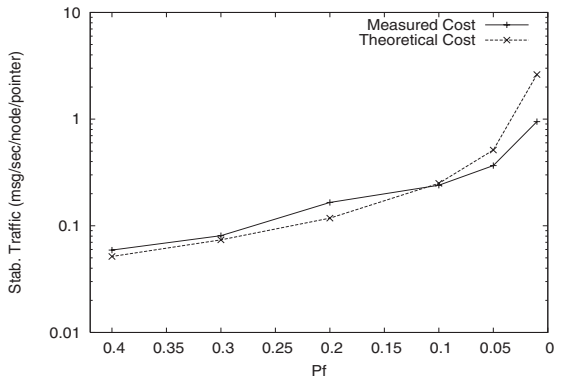


Figure 5. AS: Performance-cost tradeoff

6 Conclusions

We have proposed an adaptive stabilization framework for DHT that dynamically adjusts the rate of stabilization according to the evolution of network conditions. Peers collect statistical information about the network and determine the required stabilization rate in order to achieve target QoS objectives.

The contribution of this paper is two-fold. First, we identify the principles of stabilization common to all DHT protocols - the liveness and accuracy check analyses - and we evaluate the effect of the two separate churn components (node join and failure) on DHT routing. Second, we formulate a mathematical model for our framework and we provide its instantiation for the Chord DHT. Our experimental evaluation shows that adaptive stabilization outperforms periodic stabilization in terms of both lookup failure and communication overhead, for constant and variable churn rate. Furthermore, adaptive stabilization provides a predictable performance-cost tradeoff model that can help in the decision of choosing a QoS threshold.

In future work, we plan to address workload modeling for real P2P system traces and to extend the *AS* framework to support general churn workloads, other than exponential distributions. In addition, we plan to investigate more accurate methods to estimate global network conditions using only local state.

References

[1] *p2psim:* The Peer-to-Peer Network Simulator. *http://pdos.csail.mit.edu/p2psim*.

[2] Secure Hash Standard. *U.S. Dept. Commerce/NIST, Springfield, VA, FIPS 180-1, Apr. 1995*.

[3] L. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N,k,f):a Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P applications. *The 3rd Int'l Workshop CCGRID2003*, 2003.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1489–1499, 2002.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *In Proc. of ACM SOSP'01, Banff, Canada*, 2001.

[6] S. El-Ansary and S. Haridi. *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad-Hoc Wireless, and Peer-to-Peer Networks*, chapter An overview of structured overlay networks. CRC, 2004.

[7] P. Gummadi, S. Saroiu, and S. Gribble. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Multimedia Systems Journal*, 9(2):170–184, 2003.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[9] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi. A statistical theory of Chord under churn. In *Proc. of the 5th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[10] J. Li, J. Stribling, T. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of the 3rd Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[11] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proc. of Principles of Distributed Computing*, pages 233–242, 2002.

[12] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. of the 2nd Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. *Tech. Report TR-00-010, Berkeley, CA*, 2000.

[14] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2233:14–26, 2001.

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

[16] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[17] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM 2001*, pages 149–160, 2001.

[18] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.