An Analytical Study of Consistency and Performance of DHTs under Churn[1]

Sameh El-Ansary[1], Supriya Krishnamurthy[1], Erik Aurell[1,2] and Seif Haridi[1,3]

[1] Distributed Systems Laboratory
SICS Swedish Institute of Computer Science
P. O. Box 1263 SE-16429 Kista, Sweden

[2]Department of Physics, KTH-Royal Institute of Technology
SE-106 91 Stockholm, Sweden

[3]IMIT, KTH-Royal Institute of Technology
{supriya,sameh,eaurell,seif}@sics.se

SICS Technical Report (Draft) T2004:12
ISSN 1100-3154
ISRN:SICS-T–2004/12-SE

Abstract. *In this paper, we present a complete analytical study of dynamic membership (aka churn) in structured peer-to-peer networks. We use a master-equation-based approach, which is used traditionally in non-equilibrium statistical mechanics to describe steady-state or transient phenomena. We demonstrate that this methodology is infact also well suited to describing structured overlay networks by an application to the Chord system. For any rate of churn and stabilization rates, and any system size, we accurately account for the functional form of: the distribution of inter-node distances, the probability of network disconnection, the fraction of failed or incorrect successor and finger pointers and show how we can use these quantities to predict both the performance and consistency of lookups under churn. Additionally, we also discuss how churn may actually be of different 'types' and the implications this will have for structured overlays in general. All theoretical predictions match simulation results to a high extent. The analysis includes details that are applicable to a generic structured overlay deploying a ring as well as Chord-specific details that can act as guidelines for analyzing other systems.*

Keywords: Peer-To-Peer, Structured Overlays, Distributed Hash Tables, Dynamic Membership in Large- scale Distributed Systems, Analytical Modeling, Master Equations.

[1]This work is funded by the Swedish VINNOVA AMRAM and PPC projects, the European IST-FET PEPITO and 6th FP EVERGROW projects.

An Analytical Study of Consistency and Performance of DHTs under Churn †

Sameh El-Ansary[1], Supriya Krishnamurthy[1], Erik Aurell[1,2] and Seif Haridi[1,3]
[1] Swedish Institute of Computer Science (SICS), Sweden
[2] Department of Physics, KTH-Royal Institute of Technology, Sweden
[3] IMIT, KTH-Royal Institute of Technology, Sweden
{supriya,sameh,eaurell,seif}@sics.se

Abstract. *In this paper, we present a complete analytical study of dynamic membership (aka churn) in structured peer-to-peer networks. We use a master-equation-based approach, which is used traditionally in non-equilibrium statistical mechanics to describe steady-state or transient phenomena. We demonstrate that this methodology is infact also well suited to describing structured overlay networks by an application to the Chord system. For any rate of churn and stabilization rates, and any system size, we accurately account for the functional form of: the distribution of inter-node distances, the probability of network disconnection, the fraction of failed or incorrect successor and finger pointers and show how we can use these quantities to predict both the performance and consistency of lookups under churn. Additionally, we also discuss how churn may actually be of different 'types' and the implications this will have for structured overlays in general. All theoretical predictions match simulation results to a high extent. The analysis includes details that are applicable to a generic structured overlay deploying a ring as well as Chord-specific details that can act as guidelines for analyzing other systems.*

1 Introduction

An intrinsic property of Peer-to-Peer systems is the process of never-ceasing dynamic membership. Structured Peer-to-Peer Networks (aka Distributed Hash Tables (DHTs)) have the underlying principle of arranging nodes in an overlay graph of known topology and diameter. This knowledge results in the provision of performance guarantees. However, dynamic membership continuously "corrupts/churns" the overlay graph and every DHT strives to provide a technique to "correct/maintain" the graph in the face of this perturbation.

Both theoretical and empirical studies have been conducted to analyze the performance of DHTs undergoing "churn" and simultaneously performing "maintenance'. Liben-Nowell et. al [6] prove a lower bound on the maintenance rate required for a network to remain connected in the face of a given dynamic membership rate. Aspnes et. al [2] give upper and lower bounds on the number of messages needed to locate a node/data item in a DHT in the presence of node or link failures. The value of such theoretical studies is that they provide insights neutral to the details of any particular DHT. Empirical studies have also been conducted to complement these theoretical studies by showing how within the asymptotic bounds, the performance of a DHT may vary substantially depending on different DHT designs and implementation decisions. Examples include the work of: Li et. al [5], Rhea et.al [8] and Rowstron et.al [3].

In this paper, we present a new approach to studying churn, based on working with master equations, a widely used tool wherever the mathematical theory of stochastic processes is applied to real-world phenomena [7]. We demonstrate the applicability of this approach to one specific DHT: Chord [9].

A master-equation description for a dynamically evolving system is achieved by first defining a *state* of the system. This is just a listing of the quantities one would need to know for the fullest description of the system. For Chord, the *state* could be defined as a listing of how many nodes there are in the system and what the state (whether correct, incorrect or failed) of each of the pointers of those nodes is. This information is not enough to draw a unique graph of network-connections (because for example, if we know that a given node has an 'incorrect' successor pointer, this still does not tell us which node it is pointing to). However, as we will see, beginning at this level of description is sufficient to keep track of most of the details of the Chord protocols.

Having defined a state, the master-equation is simply an equation for the evolution of the probability of finding the system in this state, given the details of the dynamics. The specific nature of the dynamics plays a role in evaluating all the terms leading to the gain or loss of this probability, *i.e.* keeping track of the contribution of all the events which can bring about changes in the probability in a micro-instant of time.

Using this formalism our results are accurate functional forms of the following: (i) The distribution of inter-node distances when the system is in equilibrium or in general when a network is growing or shrinking. This distribution is independent of any details of Chord and are applicable to any DHT deploying a ring. (ii) Chord-specific inter-node distribution properties. (iii) For every outgoing pointer of a Chord node, we systematically compute the probability that it is in any one

†This work is funded by the Swedish VINNOVA AMRAM and PPC projects, the European IST-FET PEPITO and 6th FP EVERGROW projects.

of its possible states. This probability is different for each of the successor and finger pointers. We then use this information to predict other quantities such as (iv) the probability that the network gets disconnected, (v) lookup consistency (number of failed lookups), and (vi) lookup performance (latency). All quantities are computed as a function of the parameters involved and all results are verified by simulations.

2 Related Work

Closest in spirit to our work is the informal derivation in the original Chord paper [9] of the average number of timeouts encountered by a lookup. This quantity was approximated there by the product of the average number of fingers used in a lookup times the probability that a given finger points to a departed node. Our methodology not only allows us to derive the latter quantity rigorously but also demonstrates how this probability depends on which finger (or successor) is involved. Further we are able to derive an exact relation relating this probability to lookup performance and consistency accurately at any value of the system parameters.

In the works of Aberer et.al [1] and Wang et.al [10], DHTs are analyzed under churn and the results are compared with simulations. However, the main parameter of the analysis is the probability that a random selected entry of a routing table is stale. In our analysis, we determine this quantity from system details and churn rates.

A brief announcement of the results presented in this paper, has appeared earlier in [4].

3 Assumptions & Definitions

Basic Notation. In what follows, we assume that the reader is familiar with Chord. However we introduce the notation used below. We use $\mathcal{K}$ to mean the size of the Chord key space and N the number of nodes. Let $\mathcal{M} = \log_2 \mathcal{K}$ be the number of fingers of a node and $\mathcal{S}$ the length of the immediate successor list, usually set to a value $= O(\log(N))$. We refer to nodes by their keys, so a node n implies a node with key $n \in 0 \cdots \mathcal{K}-1$. We use p to refer to the predecessor, s for referring to the successor list as a whole, and s_i for the i^{th} successor. Data structures of different nodes are distinguished by prefixing them with a node key e.g. $n'.s_1$, etc. Let fin_i.*start* denote the start of the i^{th} finger (Where for a node n, $\forall i \in 1..\mathcal{M}$, $n.fin_i.start = n + 2^{i-1}$) and fin_i.*node* denote the node pointed to by that finger (which is the closest successor of $n.fin_i.start$ on the ring).

Steady State Assumption. λ_j is the rate of joins per node, λ_f the rate of failures per node and λ_s the rate of stabilizations per node. We carry out our analysis for the general case when the rate of doing successor stabilizations $\alpha\lambda_s$, is not necessarily the same as the rate at which finger stabilizations $(1-\alpha)\lambda_s$ are performed. In all that follows, we impose the steady state condition $\lambda_j = \lambda_f$ unless otherwise stated. Further it is useful to define $r \equiv \frac{\lambda_s}{\lambda_f}$ which is the relevant ratio on which all the quantities we are interested in will depend, e.g, $r = 50$ means that a join/fail event takes place every half an hour for a stabilization which takes place once every 36 seconds. Throughout the paper we will use the terms $\lambda_j\Delta t$, $\lambda_f\Delta t$, $\alpha\lambda_s\Delta t$ and $(1-\alpha)\lambda_s\Delta t$ to denote the respective probabilities that a join, failure, a successor stabilization, or a finger stabilization take place during a micro period of time of length Δt.

Parameters. The parameters of the problem are hence: $\mathcal{K}$, N, α and r. All relevant measurable quantities should be entirely expressible in terms of these parameters.

Chord Algorithms & Simulation A detailed description of the algorithms used is provided in Appendix A. Since we are collecting statistics like the probability of a particular finger pointer to be wrong, we need to repeat each experiment 100 times before obtaining well-averaged results. The total simulation sequential real time for obtaining the results of this paper was about 1800 hours that was parallelized on a cluster of 14 nodes where we had $N = 1000$, $\mathcal{K} = 2^{20}$, $\mathcal{S} = 6$, $200 \leq r \leq 2000$ and $0.25 \leq \alpha \leq 0.75$.

4 The Analysis

4.1 Distributional Properties of Inter-Node Distances

During churn, the average inter-node distance is a fluctuating quantity whose distribution is used throughout our analysis. The derivation we present here of this distribution is independent of any details of the DHT implementation and depends solely on the dynamics of the join and leave process. It is hence applicable to any DHT that deploys a circular key space.

Definition 4.1 *Given two keys $u, v \in \{0...\mathcal{K}-1\}$, the "distance" between them is $u - v$ (with modulo-$\mathcal{K}$ arithmetic). We interchangeably say that u and v form an "interval" of length $u - v$. Hence the number of keys inside an interval of length ℓ is $\ell - 1$ keys.*

Definition 4.2 *Let Int_x be the number of intervals of length x, i.e. the number of pairs of consecutive nodes which are separated by a distance of x keys on the ring.*

Theorem 4.1 *For a process in which nodes join or leave with equal rates independently of each other and uniformly on the ring, and the number of nodes N in the network is almost constant with $N << K$, the probability ($P(x) \equiv \frac{Int_x}{N}$) of finding an interval of length x is: $P(x) = \rho^{x-1}(1-\rho)$ where $\rho = \frac{\mathcal{K}-N}{\mathcal{K}}$.*

Proof : By definition $\sum P(x) = 1$ and $\sum x\, P(x) = \mathcal{K}/N$. Further, for the mean number of peers, the join-leave process we consider, simply implies that $\frac{dN}{dt} = \lambda_j - \lambda_f$ We will need to

$Int_x(t+\Delta t)$	Rate of Change
$= Int_x(t) - 1$	$c_{1.1} = (\lambda_f \Delta t) 2P(x)$
$= Int_x(t) - 1$	$c_{1.2} = (\lambda_j \Delta t) \frac{N(x-1)P(x)}{\mathcal{K}-N}$
$= Int_x(t) + 1$	$c_{1.3} = (\lambda_f \Delta t) \sum_{x_1=1}^{x-1} P(x_1)P(x-x_1)$
$= Int_x(t) + 1$	$c_{1.4} = (\lambda_j \Delta t) \frac{2N}{\mathcal{K}-N} \sum_{x1>x} P(x_1)$
$= Int_x(t)$	$1 - (c_{1.1} + c_{1.2} + c_{1.3} + c_{1.4})$

Table 1: Gain and loss terms for $Int(x)$ the number of intervals of length x.

check that an equation for $Int_(x)$ does indeed satisfy the above constraints.

We now write an equation for Int_x by considering all the processes which lead to its gain or loss. These are summarized in table 1

First, a failure of either of the boundary nodes of an interval of size x leads to its loss at rate $c_{1.1}$. That is, since the node killed is randomly picked amongst all the nodes in the interval, the probability that it was participating on either side of an interval of length x is $2P(x)$.

Second, an interval of size x can be lost at rate $c_{1.2}$ if a joining node splits it. Only joining with keys that belong to one of the Int_x intervals can lead to the loss of an interval of length x and in each one of these, there are $x-1$ ways (available keys) for splitting. Therefore $(x-1) \times Int_x$ positions out of the $\mathcal{K} - N$ available keys can destroy an interval of length x. That is, the probability that one of the intervals of length x is destroyed is $\frac{(x-1)Int_x}{\mathcal{K}-N}$ which can be rewritten as $\frac{N(x-1)P(x)}{\mathcal{K}-N}$.

Third, the number of intervals of size x can increase by 1 at rate $c_{1.3}$ if a failure of a boundary node results in the aggregation of two adjacent intervals. To clarify that, we give the following examples. An interval of length 1 cannot be formed by such a process. An interval of length 2 can be formed by the failure of a node if the node that failed was shared between two adjacent intervals of length 1. We are assuming here that the probability of picking two adjacent intervals of length 1 is $P(1)P(1)$. This is in effect assuming that the probability of having two adjacent intervals of size 1, factorises to $P(1)^2$. However for this system, this is an accurate estimation. Thus, in general, the probability of forming an interval of length x is $\sum_{x1=1}^{x-1} P(x_1)P(x-x_1)$.

Fourth, an increase can happen at rate $c_{1.4}$ if a join event splits a larger interval into an interval of size x. For a join to form an interval of length x, it must occur in an interval of length greater than x. In each interval of length $x_1 > x$, there are exactly two ways of forming an interval of length x. Therefore, the probability of forming an interval of length x is equal to $\frac{2\sum_{x_1>x} Int_x}{\mathcal{K}-N}$, which can be rewritten as $\frac{2N\sum_{x_1>x} P(x)}{\mathcal{K}-N}$

Finally, Int_x remains the same if none of the above happens. Therefore the equation for Int_x for $x > 1$ is:

$$\begin{aligned}\frac{dInt_x}{dt} =& -P(x)\left[2\lambda_f + \frac{N\lambda_j(x-1)}{\mathcal{K}-N}\right] \\ &+ \lambda_f \sum_{x_1=1}^{x-1} P(x_1)P(x-x_1) \\ &+ 2\lambda_j \frac{N}{\mathcal{K}-N} \sum_{x_1>x} P(x_1).\end{aligned} \tag{1}$$

The equation for Int_1 is the same as the above except that the second term is missing.

We can check that :

$$\frac{d}{dt}\sum Int_x = \frac{dN}{dt} = \lambda_j - \lambda_f \tag{2}$$

as required.

Further we can check that the constraint:

$$\frac{d}{dt}\sum xInt_x = \frac{d\mathcal{K}}{dt} = 0$$

is also obeyed. Equation 1 can be readily solved leading to the solution:

$$P(x) = \rho^{x-1}(1-\rho) \tag{3}$$

where $\rho = \frac{\mathcal{K}-N}{\mathcal{K}-N(1-\frac{\lambda_j}{\lambda_f})}$. In the special case we are interested in here where $\lambda_j = \lambda_f$, we have $\rho = \frac{\mathcal{K}-N}{\mathcal{K}}$. Note that if $\lambda_j \neq \lambda_f$, then N is actually an increasing/decreasing function of time. ■

Given the above term for ρ we can state the following corollary that gives an intuitive meaning for ρ in the case $\lambda_j = \lambda_f$.

Corollary 4.1.1 *Given a ring of $\mathcal{K}$ keys populated by N nodes, $\rho \equiv \frac{\mathcal{K}-N}{\mathcal{K}}$ is the ratio of the unpopulated keys to the total number of keys, i.e. the probability of picking a key at random and finding it empty is ρ.*

The proof of the above theorem does assume that (in the case $\lambda_j = \lambda_f$) the number of nodes N is fairly constant. Indeed at first sight this seems to be strictly true from Eq. 2. However, just as in a random walk, the variance in this case increases with time. We will comment more on the properties of the variance later. For the moment, we note that the above result can be generalised to also include the case when N is a largely fluctuating quantity. In this case we only need to multiply the N dependent terms in Eq. 1 with $Prob(N,t)$: the probability that there are N nodes in the system at time t, and average over N.

We now derive some properties of this distribution which will be used in the ensuing analysis.

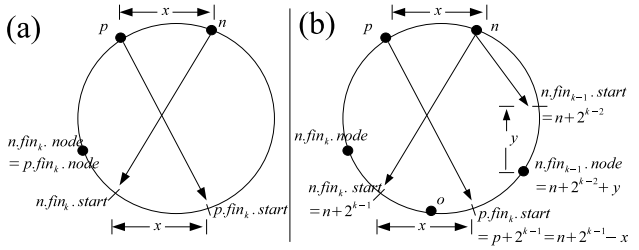


Figure 1: (a) Case when n and p have the same value of $fin_k.node$. (b) Case where a newly joined node p copies the k^{th} entry of its successor node n as the best approximation for its own k^{th} entry (by the join protocol). In this case, there could be a node o which is the 'correct' entry for $p.fin_k.node$. However, since p is newly joined, the only information it has access to is the finger table of n.

Property 4.1 *For any two keys u and v, where $v = u + x$, let b_i be the probability that the first node encountered inbetween these two keys is at $u + i$ (where $0 \leq i < x$). Then $b_i \equiv \rho^i(1-\rho)$. The probability that there is definitely atleast one node between u and v is: $a(x) \equiv 1 - \rho^x$. Hence the conditional probability that the first node is at a distance i given that there is atleast one node in the interval is $bc(i,x) \equiv b(i)/a(x)$.*

Explanation : Consider b_i first. For any key u, the probability that the first node encountered is at u itself (b_0) is $1-\rho$ from Corollary 4.1.1. Similarly the probability that the first node encountered is at $u+1$ (b_1) is $\rho(1-\rho)$. In general, the probability that the first populated node starting from u is at $u+i$ is $b(i) \equiv (\rho)^i(1-\rho)$. Given this, the probability that there is atleast one node between u and $v = u + x$ (not including the case when the node is at v) is $\sum_{i=0}^{x-1} b_i = 1 - \rho^x \equiv a(x)$. ∎

Property 4.2 *The probability that a node and atleast one of its immediate predecessors share the same k^{th} finger is $p_1(k) \equiv \frac{\rho}{1+\rho}(1-\rho^{2^k-2})$. This is $\sim 1/2$ for $\mathcal{K} >> 1$ and $N << \mathcal{K}$.Clearly $p_1 = 0$ for $k = 1$. It is straightforward (though tedious) to derive similar expressions for $p_2(k)$ the probability that a node and atleast two of its immediate predecessors share the same k^{th} finger, $p_3(k)$ and so on.*

Explanation : If the distance between node n and its predecessor p is x, the distance between $n.fin_k$.*start* and $p.fin_k$.*start* is also x (see Fig. 1(a)). If there is no node inbetween $n.fin_k$.*start* and $p.fin_k$.*start* then $n.fin_k$.*node* and $p.fin_k$.*node* will share the same value. From Eq. 3, the probability that the distance between n and p is x is $\rho^{x-1}(1-\rho)$. However, x has to be less than 2^{k-1}, otherwise $p.fin_k$.*node* will be equal to n. The probability that no node exists between $n.fin_k$.*start* and $p.fin_k$.*start* is ρ^x (by Property 4.1). Therefore the probability that the $n.fin_k$.*node* and $p.fin_k$.*node* share the same value is: $\sum_{x=1}^{2^{k-1}-1} \rho^{x-1}(1-\rho)\rho^x = \frac{\rho}{1+\rho}(1-\rho^{2^k-2})$ ∎

Property 4.3 *We can similarly assess the probability that the join protocol results in further replication of the k^{th} pointer. Let us define the probability $p_{\text{join}}(i,k)$ as the probability that a newly joined node, chooses the i^{th} entry of its successor's finger table for its own k^{th} entry. Note that this is unambiguous even in the case that the successor's i^{th} entry is repeated. All we are asking is, when is the k^{th} entry of the new joinee the same as the i^{th} entry of the successor? Clearly $i \leq k$. Infact for the larger fingers, we need only consider $p_{\text{join}}(k,k)$, since $p_{\text{join}}(i,k) \sim 0$ for $i < k$. Using the interval distribution we find, for large k, $p_{\text{join}}(k,k) \sim \rho(1-\rho^{2^{k-2}-2}) + (1-\rho)(1-\rho^{2^{k-2}-2}) - (1-\rho)\rho(2^{k-2}-2)\rho^{2^{k-2}-3}$. This function goes to 1 for large k.*

Explanation : A newly joined node p, tries to assign $p.fin_k.node$ to the best approximate value from the finger table of its successor n. This approximate value might turn out to be $n.fin_k.node$, especially for the larger fingers. If p chooses the k_{th} entry of n as its own k_{th} entry, it must be because the $k-1^{th}$ entry of n (if distinct, as is always the case for large k) does not afford it a better choice. The condition for this is : $p.fin_k.start > n.fin_{k-1}.node$. If the distance between $n.fin_k.start$ and $p.fin_k.start$ is x, and the distance between $n.fin_{k-1}.start$ and $n.fin_{k-1}.node$ is y (see Fig. 1 (b)), then the constraint on x and y is $n + 2^{k-1} - x > n + 2^{k-2} + y$ or $x + y < 2^{k-2}$. We also have the added constraint that $x < 2^{k-1}$, since otherwise $p.fin_k.node$ would simply be n. Thus the probability $p_{\text{join}}(k,k)$ is:

$$\sum_{x=1}^{2^{k-1}-1} \sum_{y=1}^{2^{k-2}-x} P(x)P(y) = \sum_{z=2}^{2^{k-2}-1} \rho^{z-2}(1-\rho)^2(z-1) \quad (4)$$

∎

where we have put in the expressions for $P(x)$ and $P(y)$ from Eq. 3 and converted the double summation to a single one. This expression can be summed easily to obtain the result quoted above.

We can also analogously compute $p_{\text{join}}(i,k)$ for any i. The only trick here is to estimate the probability that starting from i, the last *distinct* entry of n's finger table *does not* give p a better choice for its k_{th} entry. This can again readily be computed using property 4.1.

4.2 Successor Pointers

We now turn to estimating various quantities of interest for Chord. In all that follows we will evaluate various *average* quantities, as a function of the parameters. However this same

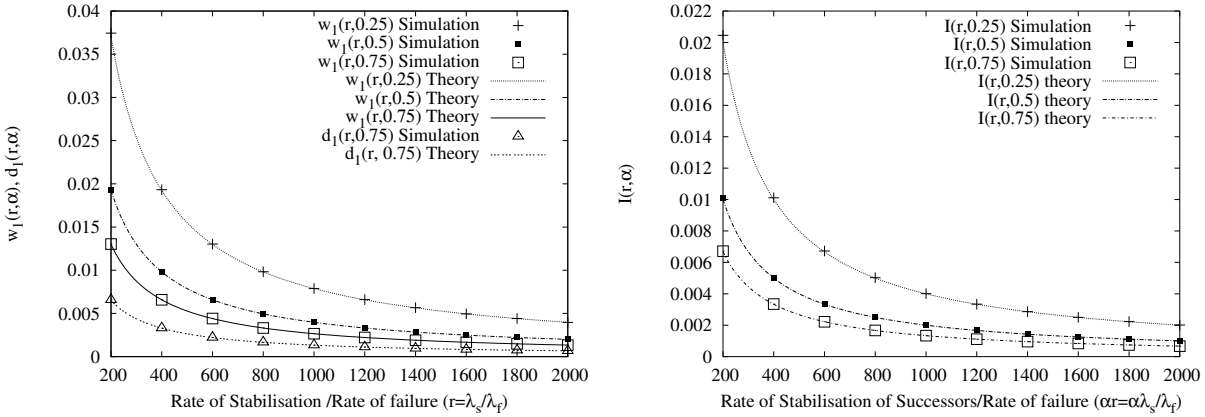


Figure 2: Theory and Simulation for $w_1(r,\alpha)$, $d_1(r,\alpha)$, $I(r,\alpha)$

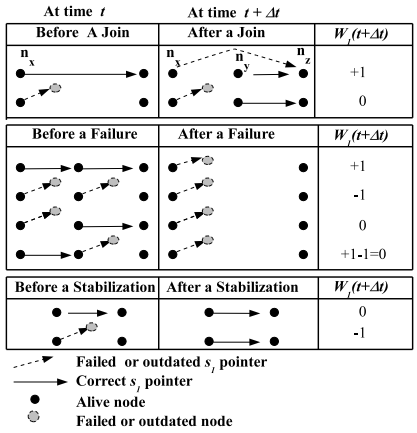


Figure 3: Changes in W_1, the number of wrong (failed or outdated) s_1 pointers, due to joins, failures and stabilizations.

formalism can also be used for evaluating higher moments like the variance.

In the case of Chord, we need consider only one of three kinds of events happening at any micro-instant: a join, a failure or a stabilization. One assumption made in the following is that such a micro-instant of time exists, or in other words, that we can divide time till we have an interval small enough that in this interval, only any one of these three processes occur. Another (more serious) assumption is that the state of the system is a *product* of the state of all the nodes. Nodes are hence assumed to have, for the most part, states independent of each other , *i.e.* the probability of two adjacent nodes having a wrong successor pointer is taken to be the product of the individual nodes having wrong successor pointers (though as we will see, in the case of finger pointers, we do also consider the case when adjacent nodes might have correlated fingers). However, this ansatz works very well.

Consider first the successor pointers. Let $w_k(r,\alpha)$, $d_k(r,\alpha)$ denote the fraction of nodes having a *wrong* k^{th} successor pointer or a *failed* one respectively and $W_k(r,\alpha)$, $D_k(r,\alpha)$ be the respective *numbers* . A *failed* pointer is one which points to a departed node and a *wrong* pointer points either to an incorrect node (alive but not correct) or a dead one. As we will see, both these quantities play a role in predicting lookup consistency and lookup length.

Change in $W_1(r,\alpha)$	Rate of Change
$W_1(t+\Delta t) = W_1(t) + 1$	$c_{2.1} = (\lambda_j \Delta t)(1 - w_1)$
$W_1(t+\Delta t) = W_1(t) + 1$	$c_{2.2} = \lambda_f (1 - w_1)^2 \Delta t$
$W_1(t+\Delta t) = W_1(t) - 1$	$c_{2.3} = \lambda_f w_1^2 \Delta t$
$W_1(t+\Delta t) = W_1(t) - 1$	$c_{2.4} = \alpha \lambda_s w_1 \Delta t$
$W_1(t+\Delta t) = W_1(t)$	$1 - (c_{2.1} + c_{2.2} + c_{2.3} + c_{2.4})$

Table 2: Gain and loss terms for $W_1(r,\alpha)$: the number of wrong first successors as a function of r and α.

By the protocol for stabilizing successors in Chord, a node periodically contacts its first successor, possibly correcting it and reconciling with its successor list. Therefore, the number of wrong k^{th} successor pointers are not independent quantities but depend on the number of wrong first successor pointers. We first consider s_1 here, and then briefly discuss the other cases towards the end of this section.

We write an equation for $W_1(r,\alpha)$ by accounting for all the events that can change it in a micro event of time Δt. An illustration of the different cases in which changes in W_1 take place due to joins, failures and stabilizations is provided in Fig. 3. In some cases W_1 increases/decreases while in others it stays unchanged. For each increase/decrease, table 2 provides the corresponding probability.

By our implementation of the join protocol, a new node n_y, joining between two nodes n_x and n_z, has its s_1 pointer always correct after the join. However the state of $n_x.s_1$ before the join makes a difference. If $n_x.s_1$ was correct (pointing to n_z) before

Change in $W_1(r,\alpha)$	Rate of Change
$N_{bu}(t+\Delta t) = N_{bu}(t) + 1$	$c_{3.1} = (\lambda_f \Delta t) d_1(r,\alpha)$
$N_{bu}(t+\Delta t) = N_{bu}(t) + 1$	$c_{3.2} = \lambda_f \Delta t (1-d_1) d_2$
$N_{bu}(t+\Delta t) = N_{bu}(t) - 1$	$c_{3.3} = \alpha \lambda_s \Delta t P_{bu}(2,r,\alpha)$
$N_{bu}(t+\Delta t) = N_{bu}(t)$	$1 - (c_{3.1} + c_{3.2} + c_{3.3})$

Table 3: Gain and loss terms for $N_{bu}(2,r,\alpha)$: the number of nodes with dead first *and* second successors

the join, then after the join it will be wrong and therefore W_1 increases by 1. If $n_x.s_1$ was wrong before the join, then it will remain wrong after the join and W_1 is unaffected. Thus, we need to account for the former case only. The probability that $n_x.s_1$ is correct is $1 - w_1$ and from that follows the term $c_{2.1}$.

For failures, we have 4 cases. To illustrate them we use nodes n_x, n_y, n_z and assume that n_y is going to fail. First, if both $n_x.s_1$ and $n_y.s_1$ were correct, then the failure of n_y will make $n_x.s_1$ wrong and hence W_1 increases by 1. Second, if $n_x.s_1$ and $n_y.s_1$ were both wrong, then the failure of n_y will decrease W_1 by one, since one wrong pointer disappears. Third, if $n_x.s_1$ was wrong and $n_y.s_1$ was correct, then W_1 is unaffected. Fourth, if $n_x.s_1$ was correct and $n_y.s_1$ was wrong, then the wrong pointer of n_y disappeared and $n_x.s_1$ became wrong, therefore W_1 is unaffected. For the first case to happen, we need to pick two nodes with correct pointers, the probability of this is $(1-w_1)^2$. For the second case to happen, we need to pick two nodes with wrong pointers, the probability of this is w_1^2. From these probabilities follow the terms $c_{2.2}$ and $c_{2.3}$.

Finally, a successor stabilization does not affect W_1, unless the stabilizing node had a wrong pointer. The probability of picking such a node is w_1. From this follows the term $c_{2.4}$.

Hence the equation for $W_1(r,\alpha)$ is:

$$\frac{dW_1}{dt} = \lambda_j(1-w_1) + \lambda_f(1-w_1)^2 - \lambda_f w_1^2 - \alpha\lambda_s w_1$$

Solving for w_1 in the steady state and putting $\lambda_j = \lambda_f$, we get:

$$w_1(r,\alpha) = \frac{2}{3+r\alpha} \approx \frac{2}{r\alpha} \tag{5}$$

This expression matches well with the simulation results as shown in Fig. 2. $d_1(r,\alpha)$ is then $\approx \frac{1}{2} w_1(r,\alpha)$ since when $\lambda_j = \lambda_f$, about half the number of wrong pointers are incorrect and about half point to dead nodes. Thus $d_1(r,\alpha) \approx \frac{1}{r\alpha}$ which also matches well the simulations as shown in Fig. 2. We can also use the above reasoning to iteratively get $w_k(r,\alpha)$ for any k.

4.3 Break-up (Network Disconnection) Probability

We demonstrate below, how calculating $d_k(r,\alpha)$: the fraction of nodes with dead k^{th} pointers, helps in estimating precisely

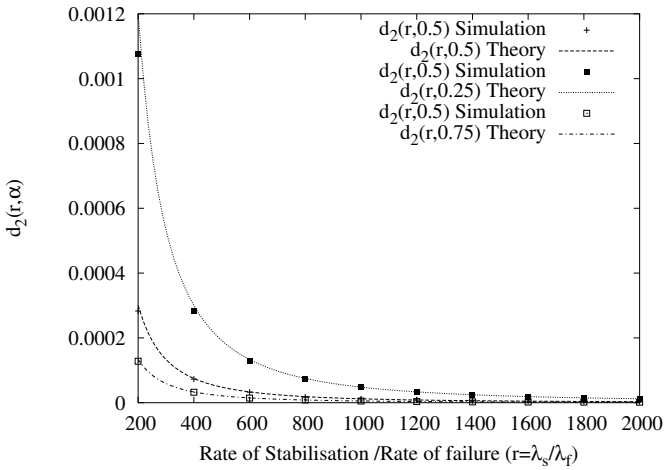


Figure 4: Theory and Simulation for $d_2(r,\alpha)$

.

the probability that the network gets disconnected for any value of r and α. Let $P_{bu}(n,r,\alpha)$ be the probability that n consecutive nodes fail. If $n = \mathcal{S}$, the length of the successor list, then clearly the node gets disconnected from the network and the network breaks up. For the range of r considered in Fig. 2, $P_{bu}(\mathcal{S},r,\alpha) \sim 0$. However should we go lower, this starts becoming finite. The master equation analysis introduced here can be used to estimate $P_{bu}(n,r,\alpha)$ for any $1 \le n \le \mathcal{S}$. We indicate how this might be done by considering the case $n = 2$. Let $N_{bu}(2,r,\alpha)$ be the number of configurations in which a node has both s_1 and s_2 dead and $P_{bu}(2,r,\alpha)$ be the fraction of such configurations. Table 3 indicates how this is estimated within the present framework.

A join event does not affect this probability in any way. So we need only consider the effect of failures or stabilization events. The term $c_{3.1}$ accounts for the situation when the *first* successor of a node is dead (which happens with probability $d_1(r,\alpha)$ as explained above). A failure event can then kill its second successor as well and this happens with probability $c_{3.1}$. The second term is the situation that the first successor is alive (with probability $1-d_1$) but the second successor is dead (with probability d_2). This probability is $\sim 2/\alpha r$. (the second successor of a node being dead either implies that the first successor of *its* first successor is dead with probability d_1, or that it has not stabilized recently, and hence has not corrected its second successor pointer.This happens with probability $\sim 1/\alpha r$. These two terms add up to $2/\alpha r$). A stabilization event reduces the number of such configurations by one, if the node doing the stabilization had such a configuration to begin with.

Solving the equation for $N_{bu}(2,r,\alpha)$, one hence obtains that $P_{bu}(2,r,\alpha) \sim 3/(\alpha r)^2$. As Fig. 4 shows, this is a precise estimate.

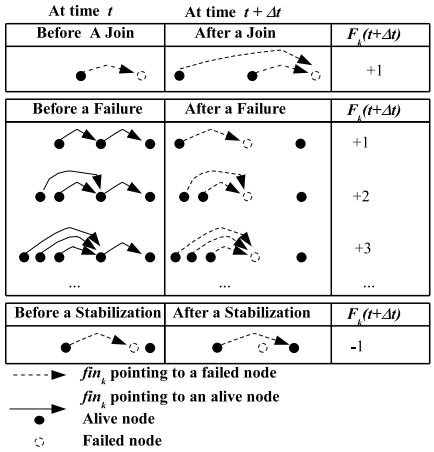


Figure 5: Changes in F_k, the number of failed fin_k pointers, due to joins, failures and stabilizations.

We can similarly estimate the probabilities for three consecutive nodes failing, *etc*, and hence also the disconnection probability $P_{bu}(\mathcal{S}, r, \alpha)$. This formalism thus affords the possibility of making a precise prediction for when the system runs the danger of getting disconnected as a function of the parameters.

Lookup Consistency By the lookup protocol, a lookup is inconsistent if the immediate predecessor of the sought key has a wrong s_1 pointer. However, we need only consider the case when the s_1 pointer is pointing to an alive (but incorrect) node since our implementation of the protocol always requires the lookup to return an alive node as an answer to the query. The probability that a lookup is inconsistent $I(r, \alpha)$ is hence $w_1(r, \alpha) - d_1(r, \alpha)$. This prediction matches the simulation results very well, as shown in Fig. 2.

4.4 Failure of Fingers

We now turn to estimating the fraction of finger pointers which point to failed nodes. As we will see this is an important quantity for predicting lookups, since failed fingers cause timeouts and increase the lookup length. We need however only consider fingers pointing to *dead* nodes. Unlike members of the successor list, *alive* fingers even if outdated, always bring a query closer to the destination and do not affect consistency or substantially even the lookup length. Therefore we consider fingers in only two states, alive or dead (failed). By our implementation of the stabilization protocol (see Appendix A), fingers and successors are stabilized entirely independently of each other. Thus even though the first finger is also always the first successor, this information is not used by the node in updating the finger.

Let $f_k(r, \alpha)$ denote the fraction of nodes having their k^{th} finger pointing to a failed node and $F_k(r, \alpha)$ denote the respective number. For notational simplicity, we write these as simply F_k and f_k. We can predict this function for any k by again estimating the gain and loss terms for this quantity, caused by a join, failure or stabilization event, and keeping only the most relevant terms. These are listed in table 4 and illustrated in Fig. 5

$F_k(t + \Delta t)$	Rate of Change
$= F_k(t) + 1$	$c_{4.1} = (\lambda_j \Delta t) \sum_{i=1}^{k} p_{join}(i, k) f_i$
$= F_k(t) - 1$	$c_{4.2} = (1 - \alpha)\frac{1}{\mathcal{M}} f_k(\lambda_s \Delta t)$
$= F_k(t) + 1$	$c_{4.3} = (1 - f_k)^2[1 - p_1(k)](\lambda_f \Delta t)$
$= F_k(t) + 2$	$c_{4.4} = (1 - f_k)^2(p_1(k) - p_2(k))(\lambda_f \Delta t)$
$= F_k(t) + 3$	$c_{4.5} = (1 - f_k)^2(p_2(k) - p_3(k))(\lambda_f \Delta t)$
$= F_k(t)$	$1 - (c_{4.1} + c_{4.2} + c_{4.3} + c_{4.4} + c_{4.5})$

Table 4: Some of the relevant gain and loss terms for F_k, the number of nodes whose kth fingers are pointing to a failed node for $k > 1$.

A join event can play a role here by increasing the number of F_k pointers if the successor of the joinee had a failed i^{th} pointer (occurs with probability f_i) and the joinee replicated this from the successor as the joinee's kth pointer. (occurs with probability $p_{join}(i, k)$ from property 4.3). For large enough k, this probability is one only for $p_{join}(k, k)$, that is the new joinee mostly only replicates the successor's kth pointer as its own kth pointer. This is what we consider here.

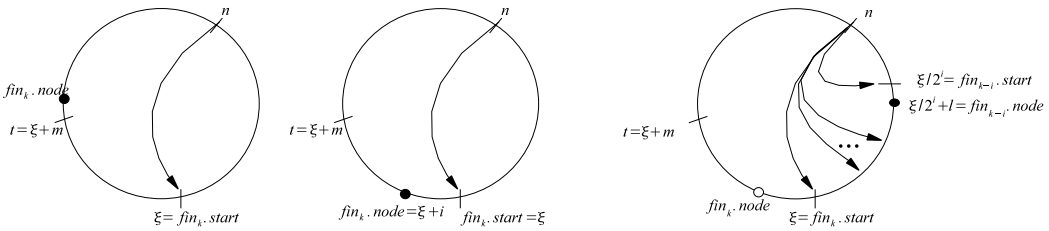
A stabilization evicts a failed pointer if there was one to begin with. The stabilization rate is divided by $\mathcal{M}$, since a node stabilizes any one finger randomly, every time it decides to stabilize a finger at rate $(1 - \alpha)\lambda_s$.

Given a node n with an alive k^{th} finger (occurs with probability $1 - f_k$), when the node pointed to by that finger fails, the number of failed k^{th} fingers (F_k) increases. The amount of this increase depends on the number of immediate predecessors of n that were pointing to the failed node with their k^{th} finger. That number of predecessors could be 0, 1, 2,.. etc. Using property 4.2 the respective probabilities of those cases are: $1 - p_1(k)$, $p_1(k) - p_2(k)$, $p_2(k) - p_3(k)$,... etc.

Solving for f_k in the steady state, we get:

$$f_k = \frac{\left[2\tilde{P}_{rep}(k) + 2 - p_{\text{join}}(k) + \frac{r(1-\alpha)}{\mathcal{M}}\right]}{2(1 + \tilde{P}_{rep}(k))} - \frac{\sqrt{\left[2\tilde{P}_{rep}(k) + 2 - p_{\text{join}}(k) + \frac{r(1-\alpha)}{\mathcal{M}}\right]^2 - 4(1 + \tilde{P}_{rep}(k))^2}}{2(1 + \tilde{P}_{rep}(k))} \quad (6)$$

where $\tilde{P}_{rep}(k) = \Sigma p_i(k)$. In principle its enough to keep even three terms in the sum. The above expressions match very well with the simulation results (Fig. 7).



Prob.	$1-a(m)$	$a(m)(1-f_k)$	$a(m)f_k h_k(i)bc(l,\xi/2^i)$
Cost $(C_t=C_{\xi+m})$	$C_{\xi+m}=C_\xi$	$C_{\xi+m}=1+bc(i,m)C_{m-i}$ $0\le i<m$	$C_{\xi+m}=1+1+C_{\xi_i-l+m}+(i-1)$ $1\le i\le k-1, 1\le l<\xi/2^i$
	(a)	(b)	(c)

Figure 6: Cases that a lookup can encounter with the respective probabilities and costs.

4.5 Cost of Finger Stabilizations and Lookups

In this section, we demonstrate how the information about the failed fingers and successors can be used to predict the cost of stabilizations, lookups or in general the cost for reaching any key in the id space. By cost we mean the number of hops needed to reach the destination *including* the number of time-outs encountered en-route. Timeouts occur every time a query is passed to a dead node. The node does not answer and the originator of the query has to use another finger instead. For this analysis, we consider timeouts and hops to add equally to the cost. We can easily generalize this analysis to investigate the case when a timeout costs some factor n times the cost of a hop.

Define $C_t(r,\alpha)$ (also denoted C_t) to be the expected cost for a given node to reach some target key which is t keys away from it (which means reaching the first successor of this key). For example, C_1 would then be the cost of looking up the adjacent key (1 key away). Since the adjacent key is always stored at the first alive successor, therefore if the first successor is alive (which occurs with probability $1-d_1$), the cost will be 1 hop. If the first successor is dead but the second is alive (occurs with probability $d_1(1-d_2)$), the cost will be 1 hop + 1 timeout = 2 and the *expected* cost is $2\times d_1(1-d_2)$ and so forth. Therefore, we have $C_1 = 1-d_1+2\times d_1(1-d_2)+3\times d_1d_2(1-d_3)+\cdots \approx 1+d_1 = 1+1/(\alpha r)$.

For finding the expected cost of reaching a general distance t we need to follow closely the Chord protocol, which would lookup t by first finding the closest preceding finger. For the purposes of the analysis, we will find it easier to think in terms of the closest preceding *start*. Let us hence define ξ to be the start of the finger (say the k^{th}) that most closely precedes t. Hence $\xi = 2^{k-1}+n$ and $t=\xi+m$, i.e. there are m keys between the sought target t and the start of the most closely preceding finger. With that, we can write a recursion relation for $C_{\xi+m}$ as follows:

$$
\begin{aligned}
C_{\xi+m} &= C_\xi\,[1-a(m)] \\
&+ (1-f_k)a(m)\left[1+\sum_{i=0}^{m-1} bc(i,m)C_{m-i}\right] \\
&+ f_k a(m)\Bigg[1+\sum_{i=1}^{k-1} h_k(i) \\
&\sum_{l=0}^{\xi/2^i-1} bc(l,\xi/2^i)(1+(i-1)+C_{\xi_i-l+m})+O(h_k(k))\Bigg]
\end{aligned}
\tag{7}
$$

where $\xi_i \equiv \sum_{m=1,i}\xi/2^m$ and $h_k(i)$ is the probability that a node is forced to use its $k-i^{th}$ finger owing to the death of its k^{th} finger. The probabilities a, b, bc have already been introduced in section 4, and we define the probability $h_k(i)$ below.

The lookup equation though rather complicated at first sight merely accounts for all the possibilities that a Chord lookup will encounter, and deals with them exactly as the protocol dictates.

The first term (Fig. 6 (a)) accounts for the eventuality that there is no node intervening between ξ and $\xi+m$ (occurs with probability $1-a(m)$). In this case, the cost of looking for $\xi+m$ is the same as the cost for looking for ξ.

The second term (Fig. 6 (b)) accounts for the situation when a node does intervene inbetween (with probability $a(m)$), and this node is alive (with probability $1-f_k$). Then the query is passed on to this node (with 1 added to register the increase in the number of hops) and then the cost depends on the length of the distance between this node and t.

The third term (Fig. 6 (c)) accounts for the case when the intervening node is dead (with probability f_k). Then the cost increases by 1 (for a timeout) and the query needs to find an alternative lower finger that most closely precedes the target.

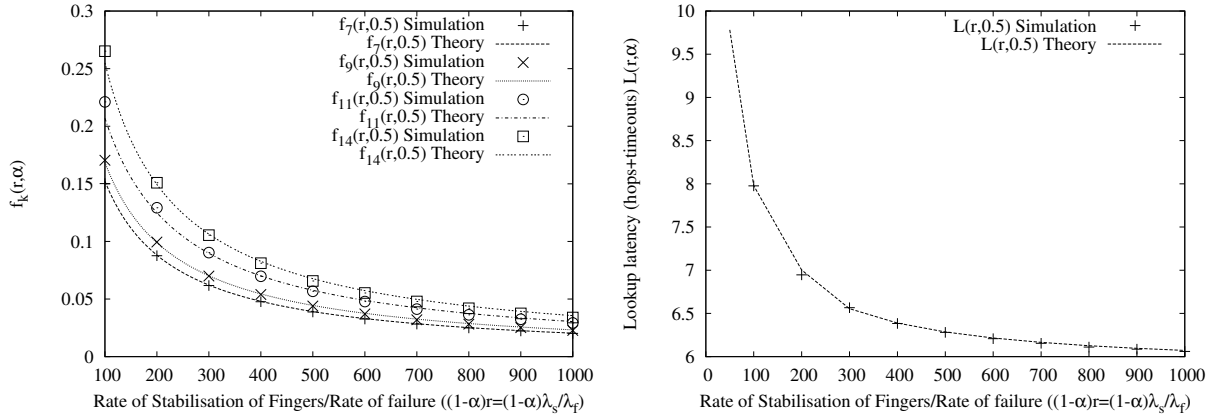


Figure 7: Theory and Simulation for $f_k(r,\alpha)$, and $L(r,\alpha)$

Let the $k-i^{th}$ finger (for some i, $1 \leq i \leq k-1$) be such a finger. This happens with probability $h_k(i)$, i.e., the probability that the lookup is passed back to the $k-i^{th}$ finger either because the intervening fingers are dead or share the same finger table entry as the k^{th} finger is denoted by $h_k(i)$. The start of the $k-i^{th}$ finger is at $\xi/2^i$ and the distance between $\xi/2^i$ and ξ is equal to $\sum_{m=1,i}\xi/2^m$ which we denote by ξ_i. Therefore, the distance from the *start* of the $k-i^{th}$ to the target is equal to ξ_i+m. However, note that $fin_{k-i}.node$ could be l keys away (with probability $bc(l,\xi/2^i)$) from $fin_{k-i}.start$ (for some l, $0 \leq l < \xi/2^i$). Therefore, after making one hop to $fin_{k-i}.node$, the remaining distance to the target is ξ_i+m-l. The increase in cost for this operation is $1+(i-1)$; the 1 indicates the cost of taking up the query again by $fin_{k-i}.node$, and the $i-1$ indicates the cost for trying and discarding each of the $i-1$ intervening nodes. The probability $h_k(i)$ is easy to compute given property 4.1 and the expression for the f_k's computed in the previous section.

$$\begin{aligned} h_k(i) =& a(\xi/2^i)(1-f_{k-i}) \\ & \times\Pi_{s=1,i-1}(1-a(\xi/2^s)+a(\xi/2^s)f_{k-s}), i<k \quad (8) \\ h_k(k) =& \Pi_{s=1,k-1}(1-a(\xi/2^s)+a(\xi/2^s)f_{k-s}) \end{aligned}$$

Eqn.8 accounts for all the reasons that a node may have to use its $k-i^{th}$ finger instead of its k^{th} finger. This could happen because the intervening fingers were either dead or not distinct. The probabilities $h_k(i)$ satisfy the constraint $\sum_{i=1}^{k} h_k(i) = 1$ since clearly, either a node uses any one of its fingers or it doesn't. This latter probability is $h_k(k)$, that is the probability that a node cannot use any earlier entry in its finger table. In this case, n proceeds to its successor list. The query is now passed on to the first alive successor and the new cost is a function of the distance of this node from the target t. We indicate this case by the last term in Eq. 7 which is $O(h_k(k))$. This can again be computed from the inter-node distribution and from the functions $d_k(r,\alpha)$ computed earlier. However in practice, the probability for this is extremely small except for targets very close to n. Hence this does not significantly affect the value of general lookups and we ignore it for the moment.

The cost for general lookups is hence

$$L(r,\alpha) = \frac{\Sigma_{i=1}^{\mathcal{K}-1} C_i(r,\alpha)}{\mathcal{K}}$$

The lookup equation is solved recursively, given the coefficients and C_1. We plot the result in Fig 7. The theoretical result matches the simulation very well.

5 What is Churn?

We now discuss a broader issue, connected with churn, which arises naturally in the context of our analysis. As we mentioned earlier, all our analysis is performed in the steady state where the rate of joins (λ_j) is equal to the rate of failures λ_f. However the rates λ_j and λ_f can themselves each be chosen in one of two different ways. They could either be "per-network" or "per-node". In the former case, the number of joinees (or the number of failures) *does not* depend on the current number of nodes in the network. This is the case when a poisson model is considered either for arrivals or departures. Put in another way, this is like saying that on average, there is always a fixed number of nodes joining or failing per time interval, irrespective of the total number of nodes in the network. In the case when these rates are chosen to be per-node, the number of joinees or failures *does* depend on the current number of occupied nodes). We consider three possibilities here, when λ_j is per-network and λ_f is per-node; both are per-network or (as is the case studied in this paper) both are per-node. In all three cases, since the system is always studied in the steady state where the total number of joinees per unit time is equal to the total number of failures per unit time, the equation for the mean is always $dN/dt = 0$. We hence expect the mean behaviour to

be the same, atleast in the regime when N is roughly constant. However the behaviour of fluctuations is very different in each of these three cases.

In the first case, the steady state condition is $\lambda_j/N_o = \lambda_f$, where N_o is the initial number of nodes in the system. The equation for the mean is $dN/dt = \lambda_j/N - \lambda_f$, which ensures that N cannot deviate too much from the steady state value. Similarly one can write an equation for the second moment N^2: $dN^2/dt = (\lambda_j/N + \lambda_f) + 2(\lambda_j - N\lambda_f)$. While the first term is a 'noise' term which encourages fluctations, the second term becomes stronger the larger the deviation from N_o and hence strongly damps out fluctuations. Thus the number of nodes in the system remains close to its initial value.

In the second case, where the join and failure rates are both per-network the equation for the mean is $dN/dt = \lambda_j/N - \lambda_f/N$. Hence putting $\lambda_j = \lambda_f$ ensures the steady state condition. However in this case, the equation for the second moment is $dN^2/dt = (\lambda_j/N + \lambda_f/N)$. The joins-failures process thus makes the system execute a "random-walk" in N, where the "steps" of the walk depend on N and are smaller if N is larger. For such a system, fluctuations are not bounded and a large deviation can and will take the system to the $N = 0$ state eventually. The time for this to happen scales with N as N^3 for this process.

The third case (which is also the case considered in this paper) is when both rates are per-node. This is very similar to the second case. The equation for the mean is just $dN/dt = \lambda_j - \lambda_f$ as mentioned earlier. Again setting $\lambda_j = \lambda_f$ ensures steady state. The equation for the second moment is now $dN^2/dt = (\lambda_j + \lambda_f)$. There is thus again no "repair" mechanism for large fluctuations, and the system will be eventually driven to extinction. In this case the process on N is just an ordinary random walk and the time taken to hit the $N = 0$ state scales as N^2.

Which of these 'types' of churn is the most relevant? In the real world, the churn felt by a DHT, might possibly be some time-varying mixture of these three, and will also possibly depend on the application. It is hence probably of importance to study all these mechanisms and their implications in detail.

6 Discussion and Conclusion

To summarize, in this paper, we have presented a detailed theoretical analysis of a DHT-based P2P system, Chord, using a Master-equation formalism. This analysis differs from existing theoretical work done on DHTs in that it aims not at establishing bounds, but on precise determination of the relevant quantities in this dynamically evolving system. From the match of our theory and the simulations, it can be seen that we can predict with an accuracy of greater than 1% in most cases.

Though this analysis is not *exact* (in the sense that there are approximations made to make the analysis simpler), yet it provides a methodology to keep track of most of the relevant details of the system. We expect that the same analysis can be done for most other DHT's in a similar manner, thus helping to establish quantitative guidelines for their comparisn.

Apart from the usefulness of this approach for its own sake, we can also gain some new insights into the system from it. For example, we see that the fraction of dead finger pointers f_k is an increasing function of the length of the finger. Infact for large enough $\mathcal{K}$, all the long fingers will be dead most of the time, making routing very inefficient. This implies that we need to consider a different stabilization scheme for the fingers (such as, perhaps, stabilizing the longer fingers more often than the smaller ones), in order that the DHT continues to function at high churn rates.

References

[1] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth, *Efficient, self-contained handling of identity in peer-to-peer systems*, IEEE Transactions on Knowledge and Data Engineering **16** (2004), no. 7, 858–869.

[2] James Aspnes, Zoë Diamadi, and Gauri Shah, *Fault-tolerant routing in peer-to-peer systems*, Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM Press, 2002, pp. 223–232.

[3] Miguel Castro, Manuel Costa, and Antony Rowstron, *Performance and dependability of structured peer-to-peer overlays*, Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), IEEE Computer Society, 2004.

[4] Supriya Krishnamurthy, Sameh El-Ansary, Erik Aurell, and Seif Haridi, *A statistical theory of chord under churn*, The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05) (Ithaca, New York), February 2005.

[5] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and Frans Kaashoek, *Comparing the performance of distributed hash tables under churn*, The 3rd International Workshop on Peer-to-Peer Systems (IPTPS'02) (San Diego, CA), Feb 2004.

[6] David Liben-Nowell, Hari Balakrishnan, and David Karger, *Analysis of the evolution of peer-to-peer systems*, ACM Conf. on Principles of Distributed Computing (PODC) (Monterey, CA), July 2002.

[7] N.G. van Kampen, *Stochastic Processes in Physics and Chemistry*, North-Holland Publishing Company, 1981, ISBN-0-444-86200-5.

[8] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz, *Handling churn in a DHT*, Proceedings of the 2004 USENIX Annual Technical Conference(USENIX '04) (Boston, Massachusetts, USA), June 2004.

[9] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, IEEE Transactions on Networking **11** (2003).

[10] Shengquan Wang, Dong Xuan, and Wei Zhao, *On resilience of structured peer-to-peer systems*, GLOBECOM 2003 - IEEE Global Telecommunications Conference, Dec 2003, pp. 3851–3856.

A Our Implementation of Chord

A.1 Joins, Failures & Ring Stablization

Initialization. Initially, the predecessor p, successors ($s_{1..\mathcal{S}}$) and fingers ($fin_{1..\mathcal{M}}$) are all assigned to nil.

Joins (Fig. 8). A new node n joins by acquiring its successor from an initial random contact node c. It also starts its first stabilization of the successors and initializes its fingers.

Stablization of Sucessors (Fig. 8). The function *fixSuccessors* is triggered periodically with rate $\alpha\lambda_s$. A node n tells its first alive successor y that it believes itself to be y's predecessor and expects as an answer y's predecessor $y.p$ and successors $y.s$. The response of y can lead to three actions:
Case A. Some node exists between n and y (i.e. n's belief is wrong), so n prepends $y.p$ it to its successors list as a first successor and retries *fixSuccessors*.
Case B. y confirms n's belief and informs n of y's old predecessor $y.p$. Therefore n considers $y.p$ as an alternative/initial predecessor for n. Finally, n reconciles its successors list with $y.s$.
Case C. y agrees that n is its predecessor and the only task of n is to update its successors list by reconciling it with $y.s$.

By calling *iThinkIamYourPred* (Fig. 8), some node x informs n that it believes itself to be n's predecessor. If n's predecessor p is not alive or nil, then n accepts x as a predecessor and informs x about this agreement by returning x. Alternatively, if n's predecessor p is alive (discovering that will be explained shortly in section A.3), then there are two possibilities: The first is that x is in the region between n and its current predecessor p therefore n should accept x as a new predecessor and inform x about its old predecessor. The second is that p is already pointing to x so the state is correct at both parties and n confirms that to x by informing it that x is the predecessor of n. In all cases the function returns a predecessor and a successors list.

The function *firstAliveSuccessor* (Fig. 8) iterates through the successors list. In each iteration, if the first successor s_1 is alive, it is returned. Otherwise, the dead successor is dropped from the list and nil is appended to the end of the list. If the first successor is *nil* this means that all immediate successors are dead and that the ring is disconnected.

A.2 Lookups and Stablization of Fingers

Stablization of Fingers (Fig. 9). Stabilization of fingers occurs at a rate $(1-\alpha)\lambda_s$. Each time the *fixFingers* function is triggered, a random finger fin_i is chosen and a lookup for $fin_i.start$ is performed and the result is used to update $fin_i.node$.

Initialization of Fingers (Fig. 9). After having initialized its first successor s_1, a node n sets all fingers with starts between n and s_1 to s_1. The rest of the fingers are initialized by taking a copy of the finger table of s_1 and finding an approximate successor to every finger from that finger table.

Lookups (Fig. 10). A lookup operation is a fundamental operation that is used to find the successor of a key. It is used by many other routines and its performance and consistency are the main quantities of interest in the evaluation of any DHT. A node n looking up the successor of k runs the *findSuccessor* algorithm which can lead to the following cases:

Case A. If k is equal to n then n is trivially the successor of

```
n.join(c)
  s_1 = c.findSuccessor(n)
  fixSuccessors()
  initFingers(s_1)
n.fixSuccessors()
  y = firstAliveSuccessor()
  {y.p, y.s} = y.iThinkIamYourPred(n)
  if (y.p ∈]me, y[) //Case A
    prepend(y.p)
    fixSuccessors()
  elsif (y.p ∈]y, me[) //Case B
    considerANewPred(y.p)
    reconcilce(y.s)
  else //Case C: y.p == me
    reconcile(y.s)

n.firstAliveSuccessor()
  while (true)
    if (s_1 == nil)
      //Broken Ring!!
    if (isAlive(s_1))
      return (s_1)
    ∀i ∈ 1..(S − 1)
      s_i = s_{i+1}
    s_S = nil

n.iThinkIAmYourPred(x)
  if ((isNotAlive(p) or (p == nil))
    p = x
    return({s, x})
  if (x ∈]p, me[)
    oldp = p
    p = x
    return({s, oldp})
  else
    return({s, p})

n.considerANewPred(x)
  if (isNotAlive(p)
    or (p == nil)
    or (x ∈]p, n[))
    p = x

n.reconcile(s')
  for i = 1..(S − 1)
    s_{i+1} = s'_i

n.prepend(y)
  for i = S..2
    s_i = s_{i−1}
  s_1 = y
```

Figure 8: Joins and Ring Stabilization Algorithms

```
n.initFingers(s_1)
  f' = s_1.f
  ∀i ∈ 1..M s.th. (fin_i.start ∈]n, s_1]),
    fin_i.node = s_1
  ∀j ∈ 1..M s.th. (fin_j.start ∉]n, s_1]),
    fin_j.node = localSuccessor(f', fin_j.start)

n.localSuccessor(f,k)
  for i = 1..M
    if (k ∈]n, fin_i])
      return(fin_i)
  return(nil)

n.fixFingers(k)
  1 ≤ i = random() ≤ M
  fin_i.node =
      findSuccessor(fin_i.start)
```

Figure 9: Initialization and Stabilization of Fingers

```
n.findSuccessor(k)
  //Case A: k is exactly equal to n
  if (k == n)
    return(n)
  //Case B: k is between n and s_1
  if (k ∈]n, s_1])
    return(firstAliveSuccessorNoChange());
  //Case C: Forward to the lookup to
  //the closest preceding alive finger
  cpf = closestAlivePrecedingFinger(k);
  if (cpf == nil)
    y = firstAliveSuccessorNoChange();
    if (k ∈]n, y])
      return(y);
    cpf = closestAlivePrecedingSucc(k);
    return(cpf.findSuccessor(k))
  else
    return (cpf.findSuccessor(k));
n.firstAliveSuccessorNoChange()
  i = 1
  while (true)
    if (s_i == nil)
      //Broken Ring!!
    if (isAlive(s_i))
      return (s_i)
    i++
```

```
n.closestAlivePrecedingFinger(k)
  for i = M..1
    if ((fin_i ∈]n, k[)
        and (fin_i ≠ nil)
        and isAlive(fin_i))
      return(fin_i)
  return(nil)
```

```
n.closestAlivePrecedingSucc(k)
  for i = S..1
    if ((s_i ∈]n, k[)
        and (s_i ≠ nil)
        and isAlive(s_i))
      return(s_i)
  return(cpf)
```

Figure 10: The Lookup Algorithm

k.

Case B. If $k \in]n, s_1]$ then n has found the successor of k, but it could be that s_1 failed and n did not discover that as yet. However, entries in the successors list can act as backups for the first successor. Therefore, the first alive successor of n is the successor of k. Note that, in this case, while we try to find the first alive successor, we do not change the entries in the successors list. This is mainly because, for the sake of the analysis, we want that the successor list is only changed at rate $\alpha\lambda_s$ by the *fixSuccessors* function and is not affected by any other rate.

Case C. The lookup should be forwarded to a node closer to k, namely the closest alive finger preceding k in n's finger table. The call to the function *closestAlivePrecedingFinger* returns such a node if possible and the lookup is forwarded to it. However, it could be the case that all alive preceding fingers to k are dead. In that case, we need to use the successors list as a last resort for the lookup. Therefore, we locate the first alive successor y and if $k \in]n, y]$ then y is the successor of k. Otherwise, we locate the closest alive preceding successor to k and forward the lookup to it.

A.3 Failures

Throughout the code we use the call $isAlive$ and $isNotAlive$. A simple interpretation of those routines would be to equate them to a performance of a ping. However, a correct implementation for them is that they are discovered by performing the operation required. For instance, a call to $firstAliveSuccesor$ in Fig. 8 is performed to retrieve a node y and then call $y.iThinkIamYourPred$, so alternatively the first alive successor could be discovered by iterating on the successor list and calling $iThinkIamYourPred$.