Minimizing Churn in Distributed Systems

Philip Brighten Godfrey
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-25
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-25.html

March 17, 2006

Minimizing Churn In Distributed Systems

P. Brighten Godfrey, Scott Shenker, and Ion Stoica

Abstract

A pervasive requirement of distributed systems is to deal with churn — change in the set of participating nodes due to joins, graceful leaves, and failures. A high churn rate can increase costs or decrease service quality. This paper studies how to reduce churn by selecting which subset of a set of available nodes to use.

First, we provide a comparison of the performance of a range of different node selection strategies in five real-world traces. Among our findings is that the simple strategy of picking a uniform-random replacement whenever a node fails performs surprisingly well. We explain its performance through analysis in a stochastic model.

Second, we show that a class of strategies, which we call "Preference List" strategies, arise commonly as a result of optimizing for a metric other than churn, and produce high churn relative to more randomized strategies under realistic node failure patterns. Using this insight, we demonstrate and explain differences in performance for designs that incorporate varying degrees of randomization. We give examples from a variety of protocols, including anycast, overlay multicast, and distributed hash tables. In many cases, simply adding some randomization can go a long way towards reducing churn.

1 Introduction

Almost every distributed system has to deal with churn: change in the set of participating nodes due to joins, graceful leaves, and failures. There is a price to churn which may manifest itself as dropped messages, data inconsistency, increased user-experienced latency, or increased bandwidth use [18, 23]. Even in peer-to-peer systems which were designed from the outset to handle churn, these costs limit the scenarios in which the system is deployable [6]. And even in a reasonably stable managed infrastructure like PlanetLab [4], there can be a significant rate of effective node failure due to nodes becoming extremely slow suddenly and unpredictably [22].

In this paper, we study how to reduce the churn rate by intelligently selecting nodes. Specifically, we consider a scenario in which we wish to use k nodes out of $n \geq k$ available. How should we select which k to use in order to minimize churn among the chosen nodes over time? This question arises in many cases, such as the following:

- Running a service on PlanetLab in which $n = 500$ nodes are available and we would like $k \approx 20$ to run the service in order to have sufficient capacity to serve requests.
- Selecting a reliable pool of $k \approx 1000$ super-peers from among $n \approx 100,000$ end-hosts participating in a peer-to-peer system.
- Choosing k nodes to be nearest the root of an overlay multicast tree, where failures are most costly.
- In a storage system of n nodes, choosing k nodes on which to place replicas of a file.

To better understand the impact of node selection on churn, we study a set of strategies that we believe are both relevant in practice, and provide a good coverage of the design space.

At the high level, we classify the selection strategies along two axes: (1) whether they use information about nodes to attempt to predict which nodes will be stable, and (2) whether they replace a failed node with a new one. We refer to strategies that base their selection on individual node characteristics (*e.g.*, past uptime or avalability) as *Stability-Predictive* strategies (or *predictive* for short), and ones that ignore such information as *Stability-Agnostic* (or just *agnostic*). On the second axis, we use the term *Fixed* for strategies that never replace a failed node from the original selected set, and *Replacement* for strategies that replace a node as soon as it fails, if another is available.

Predictive Fixed strategies are often used in the deployment of services on PlanetLab, where typically developers pick a set of machines with acceptable past availability, and then run their system exclusively on those machines for days or months. *Predictive Replacement* strategies appear in many protocols that try to dynamically minimize churn. The most common heuristic is to select the nodes which have the longest current uptime [13, 16, 26].

Agnostic strategies can frequently describe systems which do not explicitly try to minimize churn. The simplest form of *Agnostic Replacement* strategy is *Random Replacement (RR)*: replace a failed node with a uniform-random available node. Another important form of agnostic replacement strategy is a *Preference List (PL)* strategy, which arises as a result of optimizing for a metric other than churn: rank the nodes according to some preference order, and pick the top k available nodes. Note that we use the term PL specifically in the case that the preference order is *not* directly related to churn (*e.g.*, latency), and is essentially static. Such PL strategies turn out to describe many systems well. One example of a PL strategy is anycast, where one client aims to select the closest available server(s).

Results

Basic evaluation of strategies. The first part of the paper performs an extensive evaluation of churn resulting from a number of node selection strategies in five real-world traces. Among our conclusions is that replacement strategies yield up to a $5\times$ reduction in churn over the best fixed strategy in sufficiently long traces, intuitively because of their ability to dynamically adapt. This indicates that for some systems, implementing dynamic node reselection may be worth the trouble.

A more surprising finding is that there is a significant difference in churn among agnostic strategies. One might expect that selecting nodes using a metric unrelated to churn should perform similar to RR, since neither strategy uses node-specific stability information. However, it turns out that while PL strategies perform poorly, RR is quite good, *typically within a factor of less than* 2 *of the best predictive strategy.*

To explain the low churn achieved by RR, we analyze it in a stochastic model. While with an exponential session time distribution, RR is no better than Preference Lists, RR's churn rate decreases as the distributions become more skewed, which tends to be the case in realistic scenarios.

Applications to systems design. In the second part of this paper, we explore systems in which different designs or parameter choices "accidentally" induce a PL or RR-like strategy. Consider constructing a multicast tree as follows: each node, upon arrival or when one of its ancestors in the tree fails, queries m random nodes in the system, and connects to the node through which it has the lowest latency to the root. Clearly, increasing m better adapts the tree to the underlying topology, but it also has the nonobvious result that *the tree can suffer from more churn as* m *increases*, as node selection moves from being like RR to being like a PL strategy.

Of course, there will always be a tradeoff between churn and other metrics. What we aim to illuminate is the nonobvious way in which that tradeoff arises. Although this is a simple phenomenon at heart, to the best of our knowledge it has not been studied in the context of distributed systems. This framework can explain previously observed performance differences in new ways, and provide guidance for systems design.

Contributions

In summary, our main contributions are as follows:

- We provide a quantitative guide to the churn resulting from various node selection strategies in real-world traces.
- We demonstrate and analytically characterize the surprisingly good performance of Random Replacement, showing that it is much better than Preference List strategies and in many cases reasonably close to the best strategy. Its simplicity and acceptable performance may make RR an appropriate choice for certain systems.
- Using the difference between RR and PL, we demonstrate and explain performance differences in existing designs for the topology of DHT overlay networks, replica placement in DHTs, anycast server selection, and overlay multicast tree construction. In many protocols, simply adding some randomization can go a long way towards reducing churn.

This paper proceeds as follows. Section 2 evaluates churn under various selection strategies. In Section 3, we give intuition for and analysis of RR and PL startegies. Section 4 explores how the difference between RR and PL affects system design. We discuss tradeoffs between strategies in Section 5 and related work in Section 6, and conclude in Section 7.

2 Churn Simulations

The goal of this section is to understand the basic effects of various selection strategies in a wide variety of systems and node availability environments. To this end, we use a simple model of churn which will serve as a useful rule of thumb for metrics of interest in real systems. We show one such metric here — the fraction of failed route operations in a simulation of the Chord DHT [28] — and we will see others in more depth in Section 4.

In Section 2.1 we give our model of churn. We list the node selection strategies in Section 2.2 and the the traces of node availability in Section 2.3. Section 2.4 presents our simulation methodology. Our results appear in Section 2.5.

2.1 Model

In this section we define churn essentially as the rate of turnover of nodes in the system. Intuitively, this is proportional to the bandwidth used to maintain data in a load-balanced storage system.

System model. At any time, each of n nodes in the system is either *up* or *down*, and nodes that are up are either *in use* or *available*. Nodes fail and recover according to some unknown process. We call a contiguous period of being up a *session* of a node. At any time, the node selector may choose to add or remove a node from use, transitioning it from *available* to *in use* or back. There is a target number of nodes to be in use, $k = \alpha n$ for some $0 < \alpha \leq 1$, which the Replacement strategies we consider will match exactly unless there are fewer than k nodes up. The fixed strategies will pick some static set of k nodes, so they will have fewer than k in use whenever any picked node is down.

Definition of churn. Given a sequence of changes in the set of in-use nodes, let U_i be the set of in-use nodes after the ith change, with U_0 the initial set. Then churn is the sum over each event of the *fraction of the system that has changed state* in that event, normalized by run time T:

$$C = \frac{1}{T} \cdot \sum_{\text{events } i} \frac{|U_{i-1} \ominus U_i|}{\max\{|U_{i-1}|, |U_i|\}},$$

where $\ominus$ is the symmetric set difference. So in a run of length T, if we begin with k nodes in use, two nodes fail simultaneously, and the selection algorithm responds by adding two available nodes, churn is $\frac{1}{T}\left(\frac{2}{k} + \frac{2}{k}\right)$. If each of the k in-use nodes fails, one by one with no reselections, churn is $\frac{1}{T}\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{1}\right) \approx \frac{1}{T} \ln k$.

An important assumption in this definition is that a node which fails and then recovers is of no more use to us than a fresh node. This is reasonable for systems with state that is short-lived relative to the typical period of node downtime, such as in overlay multicast or $i3$ [27]. We discuss the case of storage systems, which have long-term state, in Section 4.4.

2.2 Selection strategies

Predictive Fixed strategies. When deploying a service on a reasonably static infrastructure such as PlanetLab, one could observe nodes for some time before running the system, and then use any of the following heuristics for selecting a "good" fixed set of nodes to use for the lifetime of the system, whenever they are up:

- *Fixed Decent*: Discard the 50% of nodes that were up least during the observation period. Pick k random remaining nodes. (If $k > \frac{n}{2}$, then pick all the remaining nodes and $k - \frac{n}{2}$ random discarded nodes.)
- *Fixed Most Available*: Pick the k nodes that spent the most time up.
- *Fixed Longest Lived*: Pick the k nodes which had greatest average session time.

It would be natural to try picking the k nodes that result in minimal churn during the observation period, but unfortunately this problem is NP-complete (see Appendix A.4).

Agnostic Fixed strategies. We look at only a single strategy in this class, which will turn out to be interesting because its performance is similar to Preference List strategies:

- *Fixed Random*: Pick k uniform-random nodes.

Predictive Replacement strategies. The following strategies select a random initial set of k nodes, and pick a replacement only when an in-use node fails. They differ in which replacement they choose:

- *Max Expectation:* Select the node with greatest expected remaining uptime, conditioned on its current uptime. Estimate this by examining the node's historical session times.
- *Longest Uptime:* Select the node with longest current uptime. This is the same as Max Expectation when the underlying session time distribution has decreasing failure rate.
- *Optimal:* Select the node with longest time until next failure. This requires future knowledge, but provides a useful comparison. It is the optimal strategy when future knowledge is available (see Appendix A.3).

Agnostic Replacement strategies.

- *Random Replacement (RR)*: Pick k random initial nodes. When one fails, replace it with a uniform-random available node.
- *Passive Preference List:* Given a ranking of the nodes, when an in-use node fails, replace it with the most preferable available node.
- *Active Preference List:* Given a ranking of the nodes, when an in-use node fails, replace it with the most preferable available node. When a node becomes available that's preferable to one we're using, switch to it, discarding the least preferable in-use node.

In this section, we will assume a randomly ordered preference list chosen and fixed at the beginning of each trial.

2.3 Traces

The traces we use are summarized in Table 1 and described here.

Synthetic traces: We use session times with PDF $f(x) = ab^a/(x+b)^{a+1}$ with exponent $a = 1.5$ and b fixed so that the distribution has mean 30 minutes unless otherwise stated. This is a standard Pareto distribution, shifted b units (without the shift, a node would be guaranteed to be up for at least b minutes). Between each session we use exponentially-distributed downtimes with mean 2 minutes.

PlanetLab All Pairs Ping [29]: this data set consists of pings sent every 15 minutes between all pairs of 200-400 PlanetLab nodes from January, 2004, to June, 2005. We consider a node to be up in one 15-minute interval when at least half of the pings sent to it in that interval succeeded. In a number of periods, all or nearly all PlanetLab nodes were down, most likely due to planned system upgrades or measurement errors. To exclude these cases, we "cleaned" the trace as follows: for each period of downtime at a particular node, we remove that period (i.e. we consider the node up during that interval) when the average number of nodes up during that period is less than half the average number of nodes up over all time. We obtained similar results without the cleaning procedure.

Web Sites [2]: This trace is based on HTTP requests sent from a single machine at Carnegie Mellon to 129 web sites every 10 minutes from September, 2001, to April, 2002. Since there is only a single source, network connectivity problems near the source result in periods when nearly all nodes are unreachable. We attempt to remove such effects using the same heuristic with which we cleaned the PlanetLab data.

Microsoft PCs [8]: 51,662 desktop PCs within Microsoft Corporation were pinged every hour for 35 days beginning July 6, 1999.

Gnutella application-level availability [25]: Each of a set of 17,125 IP addresses participating in the Gnutella peer-to-peer file sharing network was sent a TCP connection request every 7 minutes for about 60 hours in May, 2001. A host was marked as up when it responded with a SYN/ACK within 20 seconds. The majority of those hosts were usually down (see Table 1).

Gnutella IP-level availability [25]: This is the same as the previous trace, except a host was marked as up if it returned a TCP RST packet, indicating it was not running the Gnutella application.

2.4 Simulation setup

We tabulate churn in an event-based simulator which processes transitions in state (*down*, *available*, and *in use*) for each node. We allow the selection algorithm to react immediately after each change in node state. This is a reasonable simplification for applications which react within about 7 minutes, since the time between pings used to produce the traces is at least this much.

We also feed the sequence of events (transitions to or from the *in use* state) into a simple simulator of the Chord protocol included with the $i3$ [27] codebase. Events are node joins and failures and datagrams being sent and

Trace	Length (days)	Mean # nodes up	Median node's mean session time
PlanetLab	527	303	3.9 days
Web Sites	210	113	29 hours
Microsoft PCs	35	41970	5.8 days
Gnutella App-Level	2.5	1846	1.8 hours
Gnutella IP-Level	2.5	8037	3.3 hours

Table 1: The real-world traces used in this paper. The last column says that 50% of PlanetLab nodes had a mean time to failure of ≥ 3.9 days.

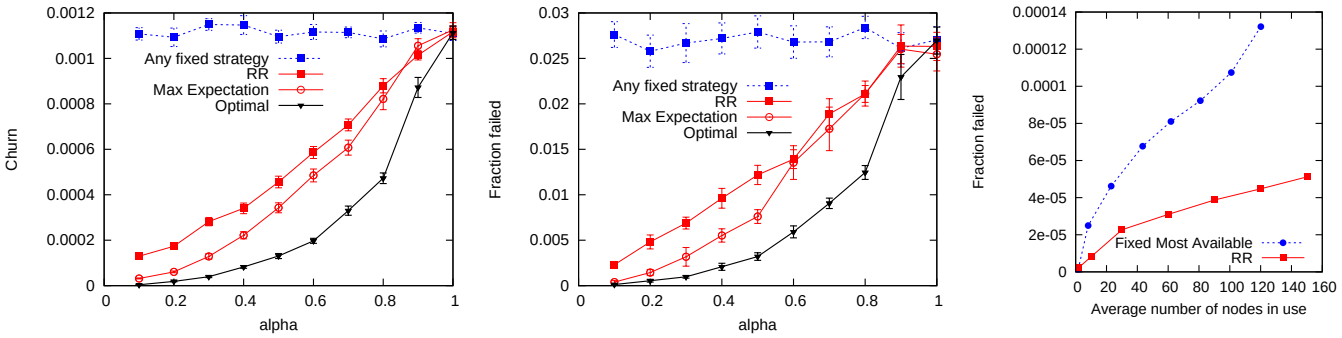


Figure 1: Churn (left) and fraction of requests failed in Chord (center) for varying α, with fixed $k = 50$ nodes in use and the synthetic Pareto lifetimes. Right: Chord in the PlanetLab trace (one trial per data point).

received. Datagram delivery is exponentially distributed with mean 50 ms between all node pairs with no loss (unless the recipient fails while the datagram is in flight). Once per simulated second we request that two random DHT nodes v_1, v_2 each route a message to the owner of a single random key k. The trial has *failed* unless both messages arrive at the same destination. Failure due to message loss was about an order of magnitude more common than failure due to inconsistency (the messages being delivered to two different nodes).

In all cases, we split each trace in half, train the fixed strategies on the first half, simulate the strategies on the whole trace, and report statistics on the second half only. All plots use at least 10 trials and show 95% confidence intervals unless otherwise stated. In plots of the Microsoft and Gnutella traces, we sample 1000 random nodes in each trial.

In the real-world traces, the parameter k does not directly control system size for the fixed strategies, since some nodes have extended downtimes. To provide a fairer comparison, we plot performance as a function of the *average number of nodes in use over time*, controlled behind the scenes by varying k. Replacement strategies have an advantage that this metric doesn't capture: the number of nodes in use is exactly k as long as $\geq k$ nodes are up.

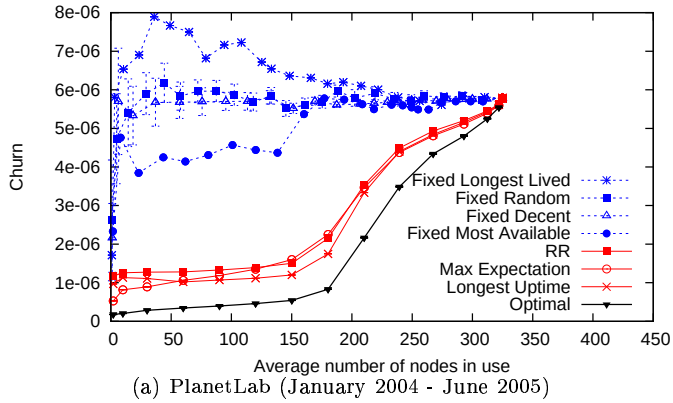
2.5 Results

The results of this section are shown in Figures 1-4. Note that for clarity in the plots, we have shown the Preference List strategies separately (Figure 4).

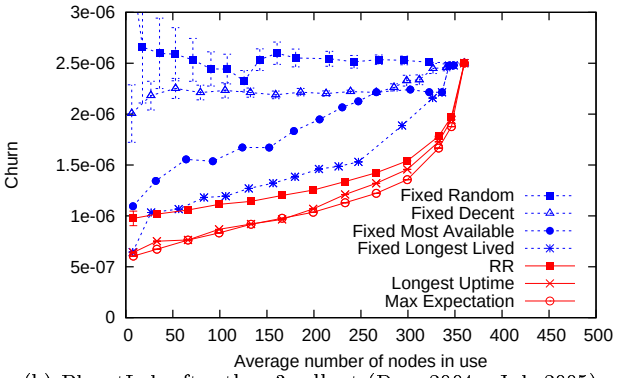
Some basic properties

Figure 1 shows churn in the synthetic Pareto session times as a function of α with fixed k, so that $n = k/\alpha$ varies. Here all fixed strategies are equivalent: since all nodes have the same mean session time, it is not possible to pick out a set of nodes that is consistently good. We can also see that Random Replacement is close to Max Expectation when α is not small. As one would expect, performance is best when $\alpha \ll 1$. In this case, Max Expectation does much better than RR intuitively because it finds the few nodes with very long time to failure.

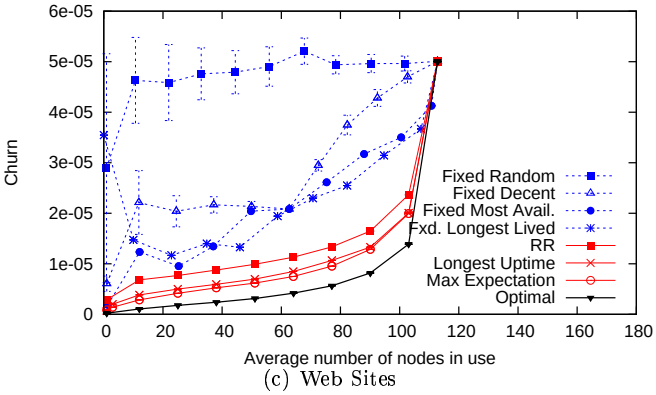
Figure 1 also demonstrates that churn is roughly proportional to fraction of requests failed in Chord, in the synthetic and PlanetLab traces. In the latter case, we vary the number of nodes in use k rather than n, which results



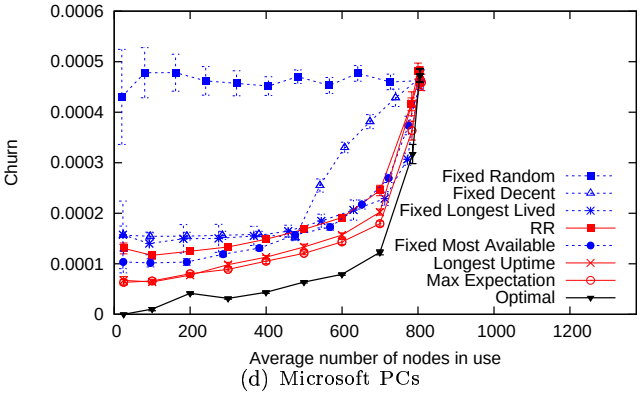
(a) PlanetLab (January 2004 - June 2005)



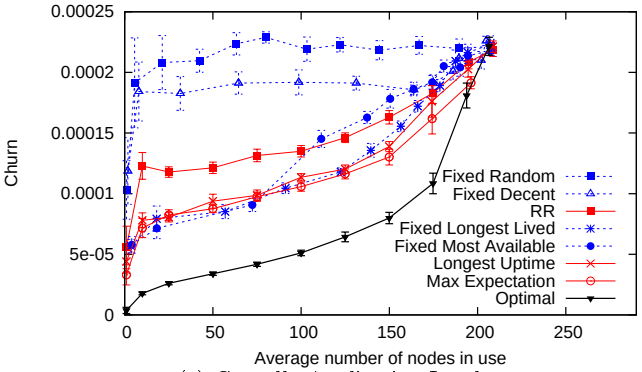
(b) PlanetLab after the v3 rollout (Dec. 2004 - Jul. 2005)



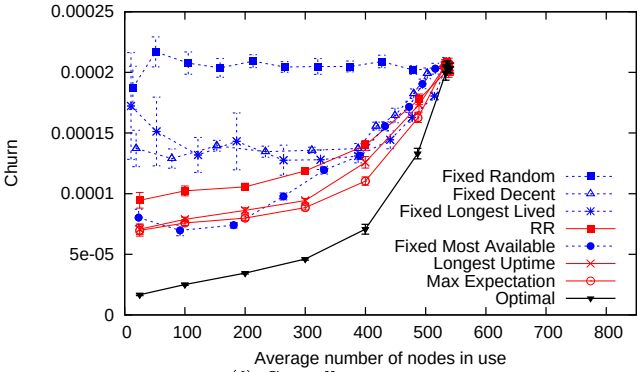
(c) Web Sites



(d) Microsoft PCs



(e) Gnutella Application-Level



(f) Gnutella IP-level

Figure 2: Churn with varying average number of nodes in the traces.

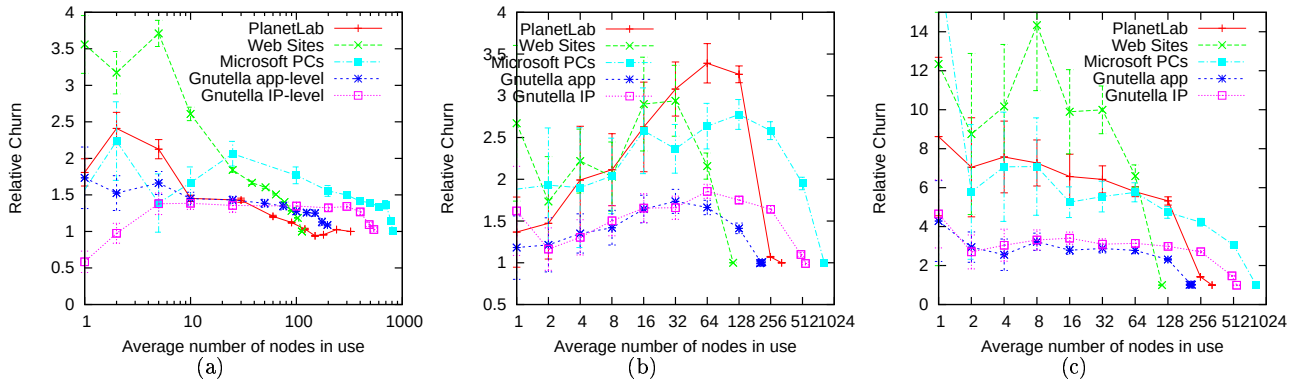


Figure 3: Churn of Random Replacement relative to other strategies: **(a)** RR divided by the best strategy, Max Expectation; **(b)** Passive Preference divided by RR; **(c)** Active Preference divided by RR.

in more failures as k grows since route lengths increase as $O(\log k)$. Not shown is that RR results in 3.3% lower mean message latency in Chord in the PlanetLab trace. We will see how churn affects other systems in Section 4.

Benefit of Replacement over Fixed strategies

Only in the Gnutella traces do the best fixed strategies match the performance of the best replacement strategies, likely since these traces are significantly shorter than the others (Table 1). In any case, fixed strategies are less applicable in a peer-to-peer setting due to the dynamic population.

In the other three traces, the best replacement strategies offer a 1.3-5× improvement over the best fixed strategy, depending on k and increasing with the length of the trace. This suggests that dynamically selecting nodes for a long-running distributed application would be worthwhile when churn has a sufficient impact on cost or service quality.

In the PlanetLab trace, the fixed strategies are particularly poor. This is primarily due to a period of uncharacteristically high churn from late October until early December, 2004, coinciding with the PlanetLab V3 rollout. During this period, fixed strategies had an order of magnitude higher churn than at other periods, while the replacement strategies increased by only about 50%. While this is impressive on the part of the replacement strategies, the rollout period may not be representative of PlanetLab as a whole. Restricting the simulation to the 6-month period after the rollout (Figure 2.5), the smart fixed strategies offer some benefit, and there is less separation between strategies in general. However, all of the replacement strategies are still more effective than the best fixed strategy.

Agnostic strategies

Figure 3(a) shows the churn of Random Replacement divided by the churn of Max Expectation, the overall best strategy (other than Optimal, which requires future knowledge). As in the synthetic distributions, RR's relative performance is worse for small k, but is usually within a factor 2 of Max Expectation.

Figure 4 shows churn under the Preference List strategies with a random preference ordering in the PlanetLab trace. Active Preference is similar to, and worse than, Fixed Random. Intuitively, this is because both strategies pay for every failure that occurs on a fixed set of k nodes. Additionally, according to our definition of churn, Active Preference pays to add preferred nodes as soon as they recover.

The Passive Preference strategy becomes more similar to Fixed Random as k increases. While it doesn't pay for every failure on the top k nodes, it is usually using those nodes and pays for most of the failures.

Figures 3(b) and (c) show churn under the Passive and Active Preference List strategies, respectively, divided by the churn of RR. RR is generally 1.2-3× better than Passive and 2.5-10× better than Active Preference.

In the next section, we give more precise intuition for — and analysis of — the differing performance of RR and Preference List startegies. In Section 4 we will show how that difference affects system design.

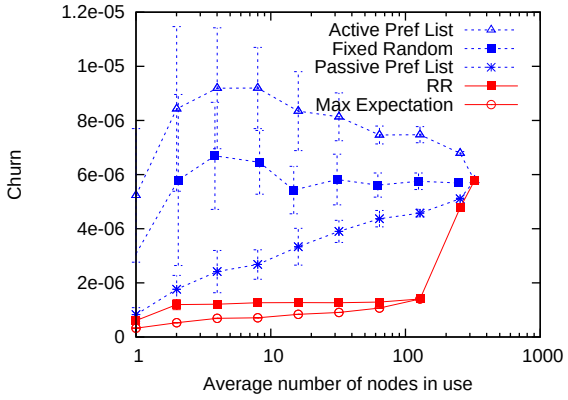


Figure 4: Preference List strategies in the PlanetLab trace. Note the log-scale x axis.

3 Analysis

Why does picking a random replacement for each failed node produce much lower churn than using a fixed random set of nodes, or the top k nodes on a preference list? We answer that question within a stochastic model defined in Section 3.1. We give intuition for why Preference List strategies are as bad as Fixed Random in Section 3.2, and why RR does better in Section 3.3.

Our main analytical results are in Section 3.4. We derive RR's expected churn rate, show that its churn decreases as the session time distributions become more skewed, and show that if all nodes have equal mean session time, RR has no worse than twice the churn of any fixed or Preference List strategy. However, if there are very few nodes with high mean session time, RR can be much worse.

3.1 Stochastic model

We use the following renewal process. For each node v_i, there is a distribution of session times with given PDF f_i and mean μ_i. At time 0 all nodes are up. Each node draws a session time ℓ_1 from its distribution independently of all other nodes, fails at time ℓ_1, recovers instantaneously, draws another session time ℓ_2, fails at time $\ell_1 + \ell_2$, and so on until the end of the run at some given time T. We will be interested in the expected churn as $T \to \infty$.

To simplify the exposition, we will assume all nodes have equal mean μ unless otherwise specified.

3.2 Fixed and Preference List strategies

Fixed strategies are very easy to analyze in this model. Since nodes recover instantaneously, our definition of churn reduces to $\frac{2}{kT}$ times the number of failures ($\frac{1}{k}$ for each failure and $\frac{1}{k}$ for each recovery, normalized by time T). As $T \to \infty$, the number of failures on any node approaches its expected value T/μ, so the total number of failures on the k selected nodes approaches $\frac{Tk}{\mu}$. Thus all fixed strategies result in expected churn $\frac{2}{kT} \cdot \frac{Tk}{\mu} = 2/\mu$.

Now consider Passive Preference and suppose S is the set of k most preferred nodes. Like fixed strategies, each failure of some node $v \in S$ causes us to pay $\frac{2}{kT}$ for the failure and replacement. Since recovery is instantaneous, the next time *some other* node fails, v must be its replacement (at any time there will be at most one node in S not in use). As k grows, the rate of failures of in-use nodes grows, so we switch back to v more and more quickly. In particular, the probability that we switch back to v before its next failure approaches 1. Thus, for large k, Passive Preference pays for nearly every failure on $\{v_1, \ldots, v_k\}$ and its churn approaches $2/\mu$ also.

Active Preference is similar, but it pays to switch back to v after its recovery, yielding twice the churn, $4/\mu$.

3.3 Intuition for Random Replacement

RR's good performance is an example of the classic "waiting time paradox". When RR picks a node v_i after a failure, the replacement's time to failure (TTF) is *not* simply drawn from the session time distribution f_i. Rather, RR is (roughly) selecting the *current* session of a random node. This is skewed towards longer sessions since a node spends longer in a long session than in a short one.

Alternately, consider some node in the system. As it proceeds through a session, the probability that it has been picked by RR increases, simply because there have been more times that it was considered as a potential replacement. Thus, nodes with longer uptimes are more likely to have been picked. And for realistic distributions, nodes with longer uptimes are less likely to fail soon.

But RR does very badly when stable nodes are rare. Suppose $k = 1$ and all nodes have exponential session times, one with mean $r \gg 1$ and $n-1$ with mean 1. When RR selects a node, its expected time to failure is $\frac{1}{n}(r) + \frac{n-1}{n}(1) \approx 1$ when $n \gg r$, so its churn is 2. But the best fixed strategy has mean TTF r and churn $2/r$.

A rigorous and general analysis of RR takes some more work and is the subject of the next section.

3.4 Analysis of Random Replacement

We now derive RR's churn rate in terms of the session time distributions and α, assuming large n and T but not assuming equal means (Theorem 1), and show that the analysis matches simulations even for $n = 20$ (Figure 5). From this we show that the churn of RR decreases as the distributions become more "skewed" (Corollary 1). We will define this rigorously, but as an example, the Pareto distribution becomes more skewed as the exponent parameter a decreases [1]. Finally, we show that for any session distributions that have equal mean, RR has at most twice the expected churn of any fixed or Preference List strategy (Corollary 2).

To simplify the analysis, we assume nodes belong to an arbitrarily large constant number d of groups of n/d nodes, such that the nodes within each group i have the same session time distribution f_i. Additionally, our analysis assumes that the session time distributions have the property that the system converges to a steady state, in the following sense.

Definition 1 ***(Stability)*** *Let C be the churn rate and L_i be a session time of node i chosen uniformly at random over all sessions in a run of length T. Let random variables X_i and R_i be the length of L_i and the number of reselections during L_i. Finally, let $c = \frac{\alpha n}{2} \cdot E[C]$ be the expected rate of reselections. Then the session time distributions $f_1, \ldots, f_d$ are* stable *if they have finite mean and variance, $E[C] > 0$, and $\forall i$,*

$$\Pr[(1-\varepsilon)cX_i \leq R_i \leq (1+\varepsilon)cX_i] \geq 1 - \varepsilon$$

$\forall \varepsilon > 0$, $\alpha \in (0,1)$, and sufficiently large n and T.

This property is trivially true in the (uninteresting) case that all nodes have exponentially distributed session times with common mean. We conjecture that in fact it is true quite generally. Our main analytical result is the following.

Theorem 1 *Let C be the churn in a trial of length T using Random Replacement. If the node session time distributions (f_i) are stable and $\alpha \in (0,1)$, then as $n, T \to \infty$, $E[C]$ is given by the unique solution to*

$$E[C] \;=\; \frac{2}{\alpha d} \sum_{i=1}^{d} \frac{1}{\mu_i} \left(1 - E\left[\exp\left\{-\frac{\alpha}{2(1-\alpha)} E[C] \cdot L_i\right\}\right]\right),$$

where random variable L_i has PDF f_i.

Proof: See Appendix A.1. ■

Figure 5 shows agreement of this analysis with a simulation for $n = 20$ and Pareto-distributed session times with PDF $f(x) = ab^a/(x+b)^{a+1}$, as in Figure 1. We vary a and fix b so that $\mu = 1$. Even though the analysis assumes large n, it differs from the simulation by $\leq 1.5\%$ in nearly all cases, and is always within the simulation's 95% confidence intervals.

We next characterize the churn of RR in terms of how "skewed" the session time distributions are, in the sense of the Lorenz partial order:

Definition 2 *Given two random variables $X, X' \geq 0$ with CDFs F and F', respectively, we say $X' \succeq X$ ("X' is more skewed than X") when $E[X'] = E[X] < \infty$, the PDFs of X and X' exist, and for all $y \in [0,1]$,*

$$E[X' \,|\, X' \geq x'] \quad \geq \quad E[X \,|\, X \geq x]\,,$$

where $x' = F'^{-1}(y)$ and $x = F^{-1}(y)$.

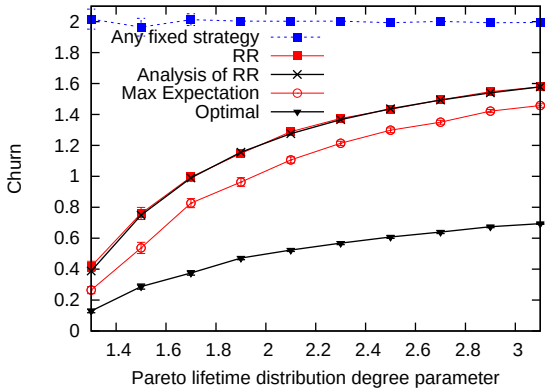


Figure 5: Simulation and analysis of churn with varying lifetime distribution, $n = 20$, and $\alpha = \frac{1}{2}$.

Note that x' and x are the yth percentile values of X' and X, so intuitively this definition compares the tails of the two distributions.

Our first corollary states that RR's expected churn decreases as the session time distributions become more skewed.

Corollary 1 *Let C and C' be the expected churn of RR as given by Theorem 1 under session time distributions (f_i) and (f'_i), respectively, and fixed α. If $f'_i \succeq f_i$ for all $i \in \{1, \ldots, d\}$, then $E[C'] \leq E[C]$.*

Thus, for fixed mean session times, the least skewed distribution — essentially the case that session times are deterministically equal to their mean — is the worst case for RR. In the special case that all mean session times are equal, we have the following:

Corollary 2 *If the session time distributions are stable and have equal mean, RR's expected churn is at most twice the expected churn of any fixed or Preference List strategy.*

The proofs appear in Appendix A.2.

4 Applications

We have seen that Random Replacement consistently outperforms Preference List strategies (Section 2.5) essentially because it takes advantage of heavy-tailed session time distributions (Section 3). In this section, we study how these two classes of strategies come up in real systems.

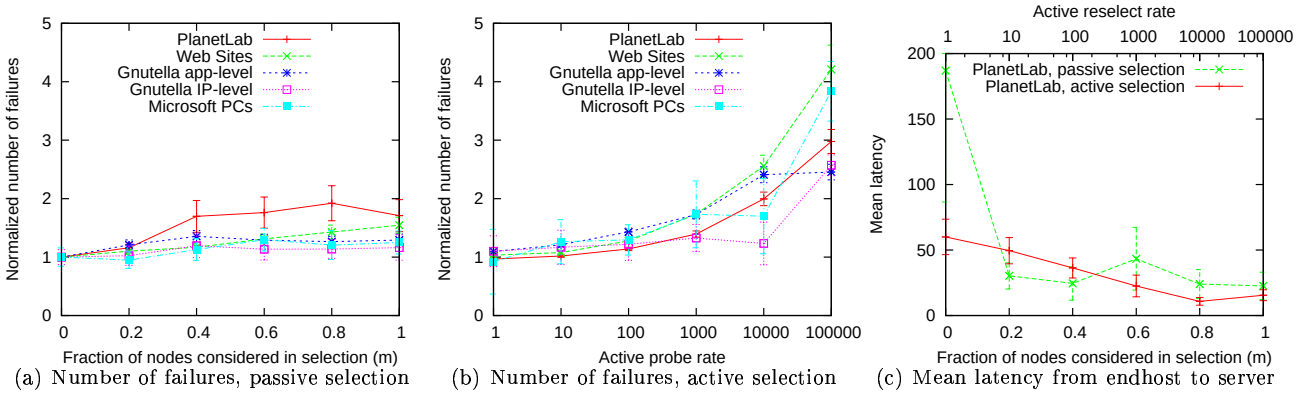
We begin in Section 4.1 with a simple example, anycast server selection, in which there are natural analogies for strategies on the spectrum between RR and PL, and doing *less* work (in terms of optimizing latency) decreases churn. In Section 4.2 we discuss how two classes of proposed DHT topologies behave like Active Preference and RR, and show that randomizing the Chord topology decreases the fraction of failed lookups by 21%.

In Section 4.3, we show how strategies similar to RR and PL occur in overlay multicast tree construction. Our results also provide further insight into an initially surprising effect observed by [26], that a random parent selection algorithm was better than a certain longest-uptime heuristic.

Finally, Section 4.4 explores two strategies for placing replicas in DHTs. A significant difference in their associated maintenance bandwidth had been previously observed but only partly explained. We show that the performance difference is also due to behavior very close to RR in one, and PL in the other.

4.1 Anycast

To give a simple instantiation of preference list strategies, consider an endhost which desires to communicate with any of a set of n acceptable servers. The endhost begins by connecting to a random server. Whenever its current server fails, it queries m random servers and connects to the one to which it has lowest latency. For $m = 1$, this is Random Replacement with $k = 1$ nodes in use, and for $m = n$ it is a Passive Preference List strategy with $k = 1$. If it seems unrealistic to query all n servers before picking a replacement, consider that recent work aims to efficiently provide the service of finding the closest server [3, 31].



(a) Number of failures, passive selection (b) Number of failures, active selection (c) Mean latency from endhost to server

Figure 6: Anycast simulation results. Number of failures are normalized by the number of failures with $m = 1$ in the passive selection case, which corresponds to the Random Replacement strategy.

We simulated the resulting number of failures as a function of m in each of the traces in a simple simulator with events at the level of node joins and failures, as in Section 2. For the PlanetLab trace, we used latencies recorded in one snapshot of the trace[1], placing the endhost at a random node in each trial. For the other traces, we used a random ordering on nodes and show only number of failures (for which all that matters is the relative ordering of distances from the endhost). We did not simulate the mechanism by which the node obtains the list of possible servers, which in practice could be provided by (for example) [3, 31].

Figure 6(a) shows that as we move from $m = 1$ to $m = n$, there is an increase in number of failures of between 16% and 71%, depending on the trace. The decrease for larger m in PlanetLab may be due to correlation between locality and time of failure (nodes in same site).

Suppose now that the endhost tries to actively improve its server selection: in addition to picking the closest of m random servers when the in-use server fails, the endhost periodically probes m servers and, if it finds one closer than its current server, switches to it. To study this, we fix $m = 10$ but vary the rate r of the periodic exponentially-distributed active probes, in terms of the expected total number of probes during the entire trace (so the fastest rate we test, $r = 100,000$, corresponds to one active reselect every 7.6 minutes in the PlanetLab trace or every 2.2 seconds in the Gnutella traces). As $r \to \infty$, this becomes an Active Preference List strategy. Figure 6(b) shows a more dramatic 2.2-4$\times$ increase in number of failures. Note that we have not counted a switch as a failure.

Of course, in both cases there is a tradeoff with latency (Figure 6(c)). The right point in the tradeoff space depends on the particular application, but these results show that we should expect stability to suffer as latency is better optimized, and conversely that doing a little *less* work is an easy way to reduce the failure rate.

4.2 DHT neighbor selection

In a DHT, each node v is assigned an identifier $id(v)$ in the DHT's keyspace. Ownership of the keys is partitioned among the nodes. In Chord, a key k is owned by the node whose ID most closely follows k in the (modular) keyspace, and most other DHTs are similar.

Each node in a DHT maintains links to certain other nodes as a function of the IDs of the nodes. Generally these come in two types: *sequential* neighbors, such as the successor list in Chord: each node v maintains links to about $\log n$ nodes whose IDs are closest to v's. These are used to maintain consistency of the partitioning of the keyspace among nodes. Second, nodes have *long-distance* neighbors, such as the finger table in Chord, to provide short routes between any pair of nodes. We will compare different ways of selecting long-distance neighbors.

Deterministic topologies

In the first class of topologies, used in Chord [28], CAN [21], and others, each node v maintains links to the owners of certain other IDs which are a deterministic function of v's ID. For example, Chord's keyspace is $\{0, \ldots, N-1\}$, where $N = 2^{160}$, and node v maintains links called *fingers* to the owners of $id(v)+2^i \pmod N$ for each $i \in \{0, \ldots, (\log_2 N)-1\}$.

[1] Specifically, inter-node latencies are the minimum RTT among 10 pings in a snapshot of June 30, 2005, the last in the trace. We used the 353 nodes in this snapshot which had $\geq 50\%$ of inlinks and $\geq 50\%$ of outlinks up.

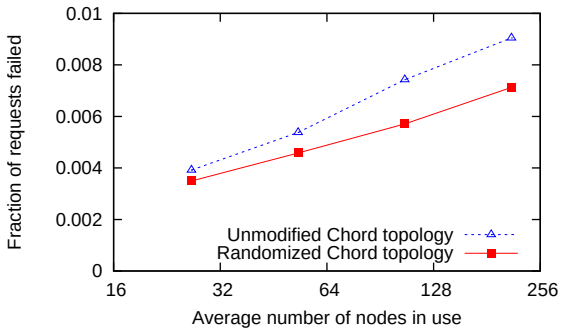


Figure 7: DHT neighbor selection simulation in the Gnutella trace. Each data point is the average of 3 trials.

This results in links to $\Theta(\log n)$ distinct nodes, where n is the number of nodes in the system. Each node periodically performs lookup operations to find the current owner of the appropriate key for each of its fingers, updating its links as ownership changes due to node arrivals and departures. In Chord, the owner of a key x is the node whose ID most closely follows x. Thus the choice of each finger i for a node v can be described as an Active Preference List strategy with $k = 1$ nodes in use, where the preference ordering ranks a node w according to the distance from $id(v) + 2^i$ to $id(w)$.

Randomized topologies

In the second class of topologies, links are chosen randomly. Symphony [19] was the first design to explicitly choose random neighbors, but many other topologies have enough underlying flexibility [14] that trivial modifications of the original design allow them to choose from many potential long-distance neighbors. For example, a natural way to randomize Chord is to select the ith finger as the owner of a random key in $\{id(v)+2^i, \ldots, id(v)+2^{i+1}\}$. When that link fails, we can choose a new random neighbor in the same range. Unsurprisingly, this strategy is essentially RR.

Simulation results

We simulated these two variants of Chord using the simulator and methodology described in Section 2.4. In each trial we sampled n random nodes from the Gnutella App-level trace and simulated a run of Chord over those n nodes, with deterministic and random neighbor selection. Since most of the nodes are usually down, we plot results as a function of $\tilde{n}$, the average number of nodes up. Figure 7 shows that with $\tilde{n} \approx 212$, the randomized topology has about 21% fewer failed requests due to the lower finger failure rate.

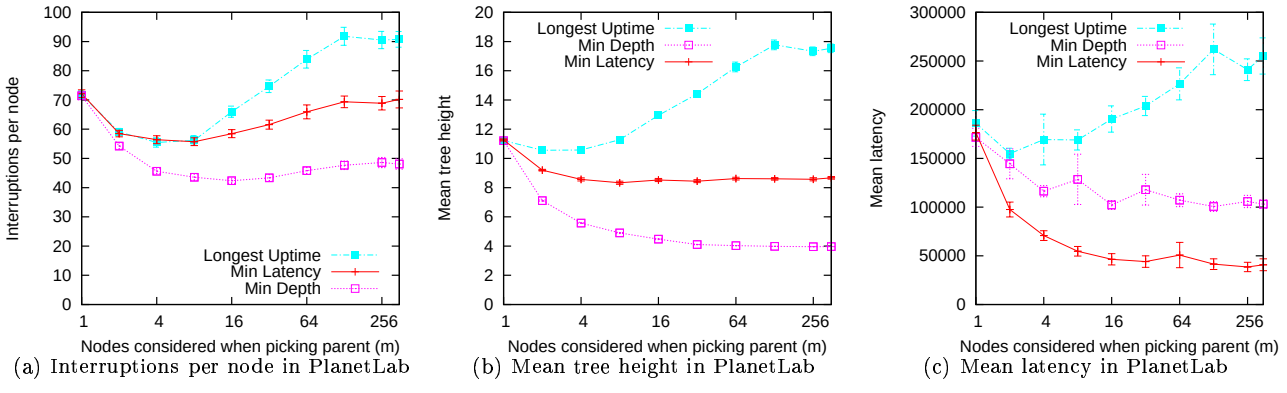
Several advantages of non-deterministic topologies are well known, most notably the ability to use proximity neighbor selection to reduce latency [14]. In the work most similar to this section, Ledlie et al [16] used Longest Uptime for finger selection. In the same Gnutella trace, their simulations showed a 42% reduction in maintenance bandwidth when compared to a proximity-optimizing neighbor selection strategy, albeit at the cost of increasing latency by 50%. In contrast, what we highlight here is that *the overlay links chosen by randomized topologies are inherently more stable than deterministic ones, even without explicitly picking neighbors based on their expected stability.*

4.3 Multicast

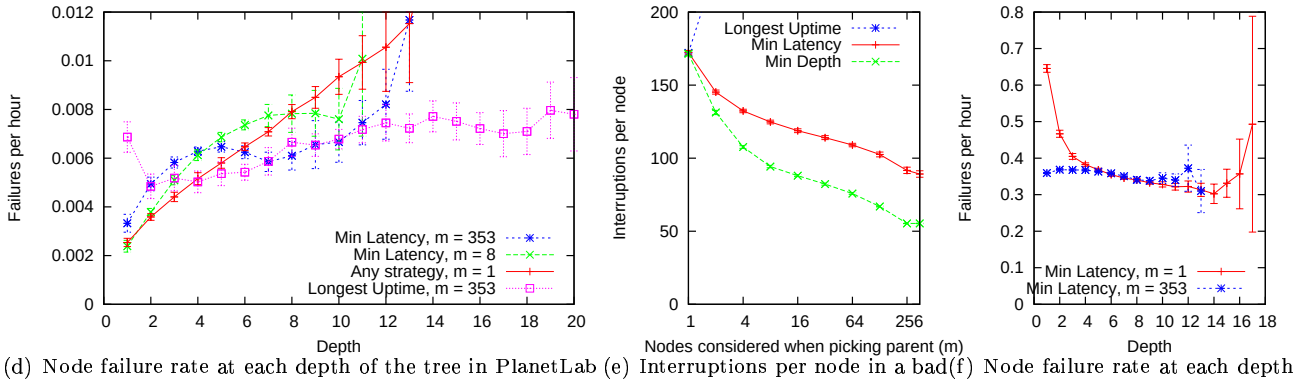
In this section we simulate how preference list strategies can affect the stability of overlay multicast trees, within a class of algorithms also studied by Sripanidkulchai et al [26].

Simulation setup

We deal with a single-source multicast tree. The root is always present without failure. When a node v joins, it contacts m random suitable nodes. A node is *suitable* for v when it is connected to the tree, has available bandwidth for another child, and v has network connectivity to it. The node then picks one of those m nodes as its parent in the tree, according to one of several strategies we will describe momentarily. Whenever a node fails, each of its



(a) Interruptions per node in PlanetLab (b) Mean tree height in PlanetLab (c) Mean latency in PlanetLab



(d) Node failure rate at each depth of the tree in PlanetLab (e) Interruptions per node in a bad case for RR (f) Node failure rate at each depth of the tree in a bad case for RR

Figure 8: Multicast simulation results.

descendants experiences an *interruption* in the hypothetical multicast stream, and repeats the join procedure. Thus a failure near the root may disrupt the structure of a large subtree.

We use three strategies for selecting the parent among the m suitable nodes: (1) the node with *Longest Uptime*; (2) the node at *Minimum Depth* from the root; or (3) the node which would result in the *Minimum Latency* along v's path through the tree to the root. The first two strategies were also simulated in [26].

Unfortunately, we could not test under the trace and node bandwidth bounds used in [26] since their data is not publicly available. Instead, we use the PlanetLab trace with uniform node capacities: each can take at most $d = 4$ children unless otherwise stated. We use PlanetLab latencies as in the previous section. For 3.71% of node-pairs (v, w), the pings from v to w failed. This produces complication: when a node v joins there might be no node in the tree which can accept another child and which v can contact. In this case we have v wait a random amount of time in $[0.5, 1.5]$ seconds and then retry. Such events are extremely rare: only 161 of over 62 million total parent-selection operations in the simulations used to produce Figure 8 needed to be retried. Finally, we note a minor difference between our simulation and that of [26]: we have a node query m suitable parents and pick the best, rather than querying m nodes, filtering out the unsuitable ones, and picking the best.

We report the total number of interruptions. Additionally, we periodically sample the tree height (maximum number of hops from root to a leaf) and total latency (sum over all nodes of the latency through the tree to the root). We take the mean of these metrics over all samples within each trial, and finally take the mean over all trials.

Results

We begin by discussing the Min Latency strategy. We will then confirm and offer additional interpretation of two results of [26] regarding the Min Depth and Longest Uptime strategies.

Figure 8(a) shows that optimizing latency both helps and hurts the number of interruptions. The case $m = 1$ is random parent selection. As we begin increasing m, minimizing latency to the root has the side effect of reducing tree height, which reduces the number of interruptions because there are fewer opportunities for failure along a node's path to the root. But for $m \geq 4$ the tree height is essentially constant (Figure 8(b)) and the trees become less stable:

the there are 26% more interruptions at $m = 353$ than at $m = 4$. We found there were 34% more interruptions for a maximum of $d = 2$ children and 10.4% more for $d = 8$ (not shown).

The interior structure of the trees reveals the proximate cause of this increased instability. Figure 8(d) shows that smaller m actually results in more stable nodes closest to the root, where failures affect the most descendants, while $m = n$ does a poorer job of getting the best nodes near the root.

We claim that the ultimate cause of this increase in failure rate for the Min Latency strategy is due to the Preference List effect. The case is not as clear as in the previous examples: even with $m = n$ the trees produced are not deterministic since the nodes re-join after an ancestor failures in random order. However, consider the $\leq d$ children of some node v in the tree. After one of the children fails, eventually a new child will join. With $m = n$ the new child is likely to be a nearby node, while with $m = 1$ the new child is selected more like RR. Then with $m = 1$ we should expect the children of v to be more stable, and hence v's grandchildren will experience fewer interruptions.

To test this hypothesis, if the nodes had session time distributions in which Random Replacement performed *worse* than Preference List strategies, performance should *improve* as $m \to n$. By Corollary 1, such an (unrealistic) bad case is when node session times are essentially constant, e.g. uniform in $[9, 11]$. Figures 8(e) and (f) show that in this case, number of interruptions is indeed a monotonically decreasing function of m and there are are worse nodes near the root when $m = 1$.

We now discuss two results of Sripanidkulchai et al [26]. First, in tests using a fixed m, they found that Min Depth best optimized stability among the strategies they tested, which Figure 8(a) confirms. Interestingly, we find that even Min Depth benefits from some randomization as well, showing 13% fewer interruptions at $m = 16$ than $m = 353$.

Second, Sripanidkulchai et al [26] found it surprising that the Longest Uptime parent selection performed more poorly than random selection ($m = 1$) in many cases, and they determined the cause was that it built much taller trees. We obtained similar results in Figure 8(a) for $m > 32$. However, we also find that using Longest Uptime, the nodes near the root are less stable than in the $m = 1$ case (Figure 8(d)). In fact, Longest Uptime is not quite optimizing the right metric: interruptions depend not only on the parent's stability, but on the stability of all nodes on the path to the root. Thus, given the results of this paper, it should not be surprising that random selection (which, rather than being agnostic, does a decent job optimizing for the *right* metric) can be better than Longest Uptime.

4.4 DHT replica placement

In this section, we compare two strategies, *Root Set* and *Directory*, for placing file replicas in distributed hash table-based storage systems. The metric we study is the rate at which new replicas have to be created, which directly affects the maintenance bandwidth of the system.

In contrast with the other applications we discuss, a storage system has persistent state. Thus, a node failure may be *transient* in that the node recovers with useful replicas, for example if the failure was a network outage or a reboot. Since our traces do not include information about the cause of downtime, we will assume that recovered nodes include no old state. This assumption will apply to both placement strategies that we compare. Moreover, we will provide evidence that the effects we describe were present in evaluations which took transient failures into account [5,30]. See [5,30] for a detailed discussion of handling transient failures, as well as various tradeoffs between the Root Set and Directory strategies which we do not discuss here.

In a DHT, nodes are assigned identifiers (IDs) in a keyspace. Each file or object o stored in the DHT is also assigned a key $key(o)$. The node whose identifier most closely follows $key(o)$ serves as the object's coordinator or *root* $r(o)$. For redundancy, some number k of replicas of o are stored on some set of nodes.

Root Set strategy

The first strategy for placing those replicas is used in slightly varying forms by DHash [9], PAST [24] Bamboo [23], and Total Recall [5], among others: put replicas on the k nodes whose IDs most closely follow $key(o)$, or the "root set". Specifically, when a node in the root set fails, we add the next closest to the set, causing one replication; when a node joins with an ID that places it in the root set, a replica of o is sent to it and we drop the replica on the node whose ID was farthest from $key(o)$, again causing one replication. Note that the root set is exactly the set of in-use nodes under an Active Preference List strategy with a static pseudorandom preference list, since the DHT nodes typically choose IDs as a hash of their IP address. Unlike the applications to anycast and multicast, the preference order here is not a user-observable metric like latency, but rather is distance in the DHT's keyspace.

Directory strategy

The second class of strategies is used by Weatherspoon et al [30] and optionally in Total Recall [5]. The root of the object o stores a directory of current locations of replicas of o, which may be on any node in the system. When a replica's node fails, the root selects a random new node on which to place a replica. The directory is assumed to be small relative to the size of an object replica, so the cost of replicating it (e.g., by using the Root Set strategy) is negligible.

Results

We performed a simple simulation of where these strategies would place replicas, and the resulting mean number of replication operations. We did not simulate the actual DHT protocol, and as in our other simulations, events are on the level of node joins and failures.

We find that if we ran a DHT over all the nodes in the PlanetLab trace, with $k = 8$ replicas per file, the root set file placement strategy would incur $\approx 7.3\times$ as many replications as the directory-based strategy. In fact, it is not hard to see that the number of replications is proportional to our definition of churn, so the ratios for various k can be obtained from Figure 3(c).

However, this result may be pessimistic because in the Root Set strategy as we have defined it, the arrival of some node v_1 in the root set causes the *deletion* of the replica on some node v_2. In fact, that replica would be useful, for example if v_1 failed again. Relatedly, as has been noted [30], Root Set has to do the work of adding the replica to v_1, while Directory is unaffected by node arrivals. To eliminate both of these characteristics, suppose we modify the Root Set strategy so that when a node arrives we leave the replica set unchanged, but when a node fails we place a replica as before on the node whose ID is closest to $key(o)$ that does not already have a replica. In this case, the two strategies are more comparable since they both create replicas only in response to node failures. The modified Root Set is a Passive Preference List strategy, and with $k = 8$ replicas in the PlanetLab trace still results in twice as many replications as the Directory strategy (Figure 3(b)).

Past studies [5, 7, 30] also observed fewer replications in the directory-based strategy. Although our simulations are considerably simplified, we note that [5, 7, 30] additionally found that the Directory strategy resulted in worse load balance than Root Set. This is a hallmark of RR's good performance: as per the intuition of Section 3.3, as a node proceeds through a session, it receives more and more replicas, and (in realistic distributions) also becomes less likely to fail.

In conclusion, there are multiple reasons for Directory's better performance: unmodified Root Set replicates in response to some node joins, and in Total Recall the directory-based strategy was performing lazy replication to mask transient failures while their Root Set lacked that optimization. However, our result shows that *a previously unrecognized part of the difference is that the Directory strategy results in inherently fewer node replica failures.*

5 Discussion

When would one use Random Replacement?

As we have seen in Section 4, RR appears in a variety of real systems. Our results are thus useful in better describing the performance of those systems.

However, if a system designer were intentionally implementing node selection to minimize churn, the results of Section 2 show that Longest Uptime offers somewhat better performance. Is there any case in which we would intentionally pick RR?

There are several cases in which RR would be easier to implement and may offer a better tradeoff between churn and system complexity. For example, when failures are due to the network, it may be hard for a node v to determine when it has "failed" and thus report its uptime. If v notices a dropped connection to some other node w, this may be due to the departure of w or a problem on the network path between v and w.

Even when it is easy to determine the uptime of a node, there may be incentive for nodes to lie about their uptime to obtain better service, such as faster file transfer in a P2P file distribution system. In this case, RR would be more robust to misbehaving peers than LU.

Finally, if we are dealing with a protocol that has already been standardized, there may be no support for querying a node's uptime. A client could potentially implement RR node selection — to pick DNS servers, for example — without support from the protocol and still obtain reasonable stability.

What about load balance?

In all effective node selection strategies, including RR, stable nodes are used more on average. What performance can we expect in the context of a shared infrastructure like PlanetLab, where users might compete for overutilized stable nodes?

The parameter α, the fraction of nodes needed, gives a way to analyze the total churn experienced by all users: we can take α to be the utilization of the distributed system as a whole. However, our results do not address fairness between users, which we leave to future work.

6 Related work

In the special case of instantaneous recovery times, there is a precise correspondance between our model of churn (Section 2.1) and page replacement in a two-level memory system: each page is a machine; the pages that are *not* in cache are the set of in-use machines; and a page access corresponds to a node failure and instantaneous recovery. Churn is thus twice the number of page faults. What we call Longest Uptime is then exactly the pervasive Least Recently Used (LRU) policy, and Random Replacement is the Random Replacement (RR) policy.

There has been a substantial amount of work on analysis of page replacement algorithms including LRU and RR; see e.g. [10–12] and the discussion in [11]. Stochastic analysis of page replacement algorithms has generally been limited to the "independent reference model" in which one page P_t is accessed in each timestep t, where the (P_t) are i.i.d. This corresponds to the special case of our model that node session times are exponentially distributed (with possibly unequal means). Thus a major difference is that our model analyzed in Sec. 3 is not limited to memoryless session times.

Longest Uptime is a common heuristic which has been studied in contexts including DHT neighbor selection [16], selecting superpeers [13], and selecting parents in an overlay multicast tree [26]. The Accordion DHT [17] selects neighbors by computing the conditional probability that a node is currently up given when it was last contacted and how long it was up before that, assuming session times fit a Pareto distribution with learned parameters. Mickens [20] used complicated statistical techniques to predict future node uptime, and experimented with placing file replicas in Chord on successors with greatest predicted time to live.

Weatherspoon et al [30] share our goal of minimizing churn, but in the context of storage systems. They avoid much of the unnecessary cost of transient failures through lazy rereplication. In contrast, our results are not specific to storage systems, but we model only non-transient failures in which a node recovers without its former data.

7 Conclusion

This paper has provided a guide to performance of a range of node selection strategies in real-world traces. We have highlighted and explained analytically the surprisingly good performance of Random Replacement relative to smart predictive strategies, and relative to Preference List strategies. Through the difference in churn between RR and PL strategies, we have explained the performance implications of a variety of existing distributed systems designs. These results also show that some dynamic randomization is an easy way to reduce churn in many protocols. An area of our ongoing work is to demonstrate these differences in a deployment of a large distributed system.

Acknowledgements

We thank the authors of [2, 8, 25, 29] for supplying their traces, and Anwitaman Datta, Jane Valentine, and Hakim Weatherspoon for helpful discussions. We also wish to acknowledge the contribution from Intel Corporation, Hewlett-Packard Corporation, IBM Corporation, and the National Science Foundation grant EIA-0303575 in making hardware and software available for the CITRIS Cluster which was used in producing these research results. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] B. C. Arnold. *Majorization and the Lorenz order: A Brief Introduction*, volume 43. Lecture Notes in Statistics, Springer-Verlag, 1987.

[2] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. On correlated failures in survivable storage systems. Technical Report CMU-CS-02-129, Carnegie Mellon University, May 2002.

[3] H. Ballani and P. Francis. Towards a global IP anycast service. In *Proc. SIGCOMM*, 2005.

[4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. NSDI*, March 2004.

[5] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *NSDI*, 2004.

[6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer neetworks: Pick two. In *Proc. HOTOS*, May 2003.

[7] C. Blake and R. Rodrigues. High availability in DHTs: Erasure coding vs. replication. In *Proc. IPTPS*, 2005.

[8] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS*, 2000.

[9] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, 2001.

[10] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. ACM SIGMETRICS*, pages 143–152, 1990.

[11] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. In *Discrete Applied Mathematics*, pages 207–229, 1992.

[12] P. A. Franaszek and T. J. Wagner. Some distribution-free aspects of paging algorithm performance. In *Journal of the Association for Computing Machinery*, pages 31–39, Jan. 1974.

[13] L. Garces-Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. Urvoy-Keller. Hierarchical P2P systems. In *Proc. ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par)*, Klagenfurt, Austria, 2003.

[14] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proc. ACM SIGCOMM*, 2003.

[15] F. B. Hildebrand. *Advanced Calculus for Applications*. Prentice-Hall, 2nd edition, 1976.

[16] J. Ledlie, J. Shneidman, M. Amis, M. Mitzenmacher, and M. Seltzer. Reliability- and capacity-based selection in distributed hash tables. Technical report, Harvard University Computer Science, Sept. 2003.

[17] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. NSDI*, 2005.

[18] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proc. INFOCOM*, 2005.

[19] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: distributed hashing in a small world. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[20] J. Mickens and B. Noble. Predicting node availability in peer-to-peer networks. In *ACM SIGMETRICS poster*, 2005.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.

[22] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proc. WORLDS*, 2005.

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

[24] A. Rowstron and P. Druschel. Storage management and caching in apst, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[25] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. MMCN*, San Jose, CA, USA, January 2002.

[26] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *Proc. SIGCOMM*, 2004.

[27] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. SIGCOMM*, 2002.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.

[29] J. Stribling. Planetlab all pairs ping. http://infospect.planet-lab.org/pings.

[30] H. Weatherspoon, B.-G. Chun, C. W. So, and J. Kubiatowicz. Long-Term Data Maintenance: A Quantitative Approach. Technical Report UCB/CSD-05-1404, EECS Department, University of California, Berkeley, July 2005.

[31] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. SIGCOMM*, 2005.

A Proofs

A.1 Proof of Theorem 1

Lemma 1 *The equation*

$$x = \frac{2}{\alpha d}\sum_{i=1}^{d}\frac{1}{\mu_i}\left(1 - E\left[\exp\left\{-\frac{\alpha}{2(1-\alpha)}x \cdot L_i\right\}\right]\right) \tag{1}$$

has at most one positive solution in terms of x, assuming $\alpha \in (0,1)$ and $E[L_i] < \infty$ for all i.

Proof: It is sufficient that the RHS is concave in x, since in this case the equation has at most two solutions, one of which is always at $x = 0$. We can justify differentiating the RHS w.r.t. x under the integral implied by the expectation, since $\mathrm{E}[L_i] < \infty$ (see [15], p. 365). Thus, the second derivative is $-\frac{\alpha}{2d(1-\alpha)^2}\sum_{i=1}^{d}\frac{1}{\mu_i}\left(\int_0^\infty \exp\left\{-\frac{\alpha}{2(1-\alpha)}x \cdot \ell\right\}\ell^2 f(\ell)d\ell\right)$ which is clearly negative for all x since $f(\ell) \geq 0$. ■

Lemma 2 *In the stochastic model of Section 3.1, churn is equal to $\frac{2}{T\alpha n}$ times the number of failures of in-use nodes.*

Proof: Since all recoveries are instantaneous, each failure of an in-use node increases churn by $\frac{1}{T\alpha n}$ and each subsequent node reselection increases churn by the same amount. ■

Proof: (of Theorem 1) We will analyze the expected churn using the fact that the number of failures is equal to the total number of times a node is selected. Let L_i be a session time of node i selected uniformly at random over all its sessions in the run of the system. (We will abuse notation and refer to L_i as both a session and the length of that session.) Let R_i be the number of reselections during L_i, and let $\beta = 1 - \alpha$. Finally, define

$$S_i = \begin{cases} 1 & \text{if } i \text{ is selected in lifetime } L_i \\ 0 & \text{o.w.} \end{cases}$$

Since each reselect picks one out of the βn available nodes u.a.r., if there are r selections during a particular node's session, the probability it is selected during that session is

$$\begin{aligned} \Pr[\text{node } i \text{ picked after } r \text{ uniform-random reselects}] &= 1 - \Pr[\text{none of } r \text{ reselects picks node } i] \\ &= 1 - \left(1 - \frac{1}{\beta n}\right)^r, \\ &\leq (1+\varepsilon)\left(1 - e^{-r/\beta n}\right), \end{aligned} \tag{2}$$

for any $\varepsilon > 0$ and sufficiently large n (since $\beta > 0$). Thus,

$$\begin{aligned} E[S_i] &= E\left[S_i \cdot 1_{(R_i \leq (1+\varepsilon)cL_i)} + S_i \cdot 1_{(R_i > (1+\varepsilon)cL_i)}\right] \\ &\leq E\left[(1+\varepsilon)\left(1 - e^{-(1+\varepsilon)cL_i/\beta n}\right)\right] + \Pr[R_i > (1+\varepsilon)cL_i] \quad \text{(by Eq. 2)} \\ &\leq (1+\varepsilon)^2 E\left[1 - e^{-cL_i/\beta n}\right] + \varepsilon \qquad \text{(since } f_i\text{'s are stable)} \\ &\leq (1+\varepsilon)E\left[1 - e^{-cL_i/\beta n}\right] + \varepsilon, \end{aligned} \tag{3}$$

where in the last step we have reset ε appropriately for notational convenience. Now letting N_i be the number of sessions that node i has in time $[0,T]$, by Lemma 2, we have

$$\begin{aligned} E[C] &= \frac{2}{\alpha nT}\sum_{i=1}^{n} E\left[N_i S_i\right] \\ &= \frac{2}{\alpha nT}\sum_{i=1}^{n} E\left[N_i S_i \cdot 1_{(N_i \leq (1+\varepsilon)EN_i)} + N_i S_i \cdot 1_{(N_i > (1+\varepsilon)EN_i)}\right] \\ &\leq \frac{2}{\alpha nT}\sum_{i=1}^{n}\left((1+\varepsilon)(EN_i)E\left[S_i\right] + E\left[N_i \cdot 1_{(N_i > (1+\varepsilon)EN_i)}\right]\right) \\ &\leq \frac{2}{\alpha nT}\sum_{i=1}^{n}\left((1+\varepsilon)(EN_i)E\left[S_i\right] + \varepsilon E\left[N_i\right]\right), \end{aligned}$$

where the last step follows from the Central Limit Theorem, which we can apply since we have assumed finite mean and variance of the distribution f_i. Now by the Strong Law of Large Numbers, $EN_i/T \to 1/\mu_i$ as $T \to \infty$, where μ_i is the mean session time of node i . Thus,

$$\begin{aligned} E[C] &\leq \frac{2}{\alpha n T} \sum_{i=1}^{n} \frac{(1+\varepsilon)T}{\mu_i} \left(\varepsilon + E\left[S_i\right]\right) \\ &\leq \frac{2}{\alpha n} \sum_{i=1}^{n} \frac{(1+\varepsilon)}{\mu_i} \left(\varepsilon + (1+\varepsilon) E\left[1 - e^{-cL_i/\beta n}\right] + \varepsilon\right) \quad \text{(by Eq. 3)} \\ &\leq O(\varepsilon') + (1+\varepsilon') \frac{2}{\alpha n} \sum_{i=1}^{n} \frac{1}{\mu_i} \left(1 - E\left[\exp\left\{-cL_i/\beta n\right\}\right]\right), \end{aligned}$$

for any $\varepsilon' > 0$ and sufficiently large n and T. Since we have assumed (Section 3.4) that the nodes are divided into d groups of n/d equivalent nodes, for notational convenience we can say w.l.o.g. that node $i \in \{1, \ldots d\}$ belongs to group i, so that the above bound reduces to

$$E[C] \leq O(\varepsilon') + (1+\varepsilon') \frac{2}{\alpha d} \sum_{i=1}^{d} \frac{1}{\mu_i} \left(1 - E\left[\exp\left\{-cL_i/\beta n\right\}\right]\right).$$

A similar technique gives the lower bound

$$E[C] \geq (1-\varepsilon') \frac{2}{\alpha d} \sum_{i=1}^{d} \frac{1}{\mu_i} \left(1 - E\left[\exp\left\{-cL_i/\beta n\right\}\right]\right).$$

Since these bounds are true for any $\varepsilon' > 0$, the result follows after substituting $c = \frac{\alpha n}{2} \cdot E[C]$ as given in Definition 1 (see Section 3.4). The uniqueness of the solution for $E[C]$ follows from Lemma 1. ■

A.2 Worst-case analysis of Random Replacement

Definition 3 *A function $f(X)$ is* Schur convex *(resp.* concave*)* in X *when $X' \succeq X$ implies $f(X') \geq f(X)$ (resp. $f(X') \leq f(X)$).*

Theorem 2 ***(Theorem 3.2 in [1])*** *$X' \succeq X$ if and only if $E[X'] = E[X]$ and $E[h(X)]$ is Schur convex in X for every continuous convex function $h : \mathbb{R}^+ \to \mathbb{R}$.*

Proof: (of Corollary 1) Fixing some j, we must show that the positive solution for x in Equation 1 (given in the statement of Lemma 1) is Schur concave in L_j. It is sufficient to show that the RHS is Schur concave in L_j for every $x > 0$, and hence that $\mathrm{E}\left[\exp\left\{-\frac{\alpha}{2(1-\alpha)} \cdot x \cdot L_j\right\}\right]$ is Schur convex. Rewriting that expression as $\mathrm{E}\left[h\left(L_j\right)\right]$ where $h(y) := \exp\left\{-\frac{\alpha}{2(1-\alpha)} \cdot x \cdot y\right\}$, we see that h is continuous and convex, so by Theorem 2, $\mathrm{E}\left[h\left(L_j\right)\right]$ is Schur convex. ■

Proof: (of Corollary 2) Let D be the degenerate random variable which is constantly 1. A standard fact (see [1]) is that $L \succeq D \cdot \mathrm{E}[L]$ for any L. Thus, since $\mathrm{E}[C]$ is Schur concave in each L_i, the positive solution to Equation 1 is maximized when L_i is constantly μ_i for all i. In this case, when $\mu_i = \mu$ for all i, Equation 1 reduces to

$$x = \frac{2}{\alpha\mu} \left(1 - \exp\left\{-\frac{\alpha}{2(1-\alpha)} \cdot x \cdot \mu\right\}\right)$$

whose only positive solutions which are $\leq 4/\mu$. Thus, $\mathrm{E}[C] \leq 4/\mu$ while any fixed or Preference List strategy has churn $\geq 2/\mu$. ■

A.3 Facts concerning dynamic strategies

We require a dynamic strategy to keep k nodes in use whenever $\geq k$ are up, and as many as possible otherwise. So during any time period in which $\leq k$ nodes are up, all (legal) dynamic strategies have the exact same behavior and the same churn. For the Facts that follow, we may therefore restrict our attention to the case that there are always

$\geq k$ nodes up. In this case, each failure and reselect costs $1/k$, and we can think about the strategy of in-use nodes as k *chains* $v^1, \ldots, v^k$ where each $v^i = (v^i_1, \ldots, v^i_{m_i})$ is such that v^i_j is selected in response to the failure of v^i_{j-1}.

A (graceful) *leave* is an event in which the selection algorithm decides to transition a node from *in use* to *available*. By the definition of churn in Section 2.1, a failure costs as much as a leave, which results in the following.

Fact 1 *Fix the pattern of failures. Suppose some strategy of node selections (v^i_j) leaves a machine and has churn C. Then there is another strategy (w^i_j) which never leaves and has churn $\leq C$.*

Proof: Let v^i_j be a node which is left at some time t_1 before its next failure at time t_2. Consider two cases: (1) node v^i_j is not used during $[t_1, t_2)$. Then let v^i_ℓ, $\ell > j$, be the node in use in chain i at time t_2 by strategy v. Form strategy w by deleting all nodes in chain i between j and ℓ, and just staying on node v^i_j during $[t_!, t_2)$. The two strategies differ only during $[t_1, t_2)$, during which v incurs at least the cost of one leave, while w incurs at most the cost of one failure. Case (2): node v^i_j is used again during $[t_1, t_2)$. If it is used again by chain i, we need only delete any intervening nodes in the chain to form w. Otherwise, if it is used by some other chain ℓ, we can swap the chains as follows: chain i stays on node v^i_j until it fails, and then continues following chain ℓ's selections from time t_2 onward. Chain ℓ, rather than switching onto node v^i_j, follows chain i's former selections. Clearly constructing strategy w in this way can only reduce the total number of failures and reselections. Iterating this argument completes the proof. ■

Fact 2 *With full knowledge of the future, the Optimal strategy of Section 2.2 is optimal.*

Proof: Let v^i_j be a node which is selected at time t_1 and fails at some future time $t_2 > t_1$, and at the time it is selected, there is an available node u which next fails at time $t_3 > t_2$. If u is never in use during $[t_1, t_3)$ then clearly we can use u instead of v^i_j, and resume following chain v beginning at time t_3. Otherwise, suppose u is used during $[t'_1, t_3)$ for some $t'_1 > t_1$ by some chain v^ℓ. Then we can modify chain v^i to use u from t_1 until it fails at time t_3, and thereafter follow the former chain v^ℓ; and we can modify chain v^ℓ to follow the former chain v^i beginning at time t'_1. This does not introduce any new failures. Iterating the argument completes the proof. ■

A.4 Facts concerning fixed strategies

Definition 4 *The decision problem* Best Fixed Strategy *(BFS) is as follows:*

- ***Instance:*** *A set V of n nodes; for each node $v \in V$ a sequence of failure and recovery times $f_1 < r_1 < f_2 < r_2 \ldots$; an integer k; and a rational c.*
- ***Question:*** *Does there exist a set $S \subseteq V$ of $\geq k$ nodes such that, when using S under the given pattern of failures, the churn incurred is $\leq c$?*

One might object that this definition is not realistic, since k does not directly control the number of nodes in use: for example, the definition allows picking a set of k nodes that are always down. But the proofs that follow do not make use of such pathological cases, and transfer directly to the variant of the problem where $|S|$ is unconstrained but we are required to have an average of $\geq k$ nodes in use over time.

Fact 3 Best Fixed Strategy *is **NP**-complete.*

Proof: Clearly the problem is in **NP**. To show **NP**-hardness, we reduce from Max Clique, an instance of which consists of a graph $G = (V, E)$ and a clique size s. First assume that *exactly* k nodes are required by BFS. We reduce the Max Clique instance to an instance of BFS as follows:

Set $k = s$. There are $n = |V|$ nodes in the BFS instance identified with the n nodes in G. All nodes are up all the time, except as we will specify. For each pair of nodes $v, w \in V$, we set aside a period of time P_{vw} in the trace during which each node other than v and w fails and recovers, all at independent times. Let $[t_1, t_4]$ be some arbitrary subinterval of P_{vw} during which there are no failures. Then if $(v, w) \in E$, we have v and w fail and recover at independent times during $[t_1, t_4]$. Otherwise, they are down during overlapping periods, according to the following sequence of events, where $t_1 < t_2 < t_3 < t_4$:

- t_1: v fails;
- t_2: w fails;

- t_3: v recovers;
- t_4: w recovers.

Now suppose we pick some set $S \subseteq V$ of k nodes to use. Note that if both v and w are in S and $(v, w) \in E$, or if one of v or w is not in S, then the churn during P_{vw} is $n \cdot 2/k$ since each node fails and recovers at independent times (so each event costs $1/k$). However, if $v, w \in S$ and $(v, w) \notin E$, then v and w have overlapping failures and the churn is $(n-1) \cdot \frac{2}{k} + \frac{2}{k-1} > 2n/k$. Summing over all $n(n-1)$ periods, we have that if S corresponds to a k-clique in G, then the churn is $n(n-1)n\frac{2}{k}$, but otherwise the churn is strictly greater. Thus, asking for a k-clique in G is equivalent to asking whether there exists a fixed set of k nodes such that the churn incurred when using S is $\leq n(n-1)n\frac{2}{k}$.

To handle the case that $> k$ nodes are permissible, we can construct additional failures such that the number of nodes chosen will dominate the churn, forcing $|S| = k$. To do this, in a "fresh" time period with no other failures, we have all the nodes fail sequentially, and then recover sequentially in the reverse order. Regardless of which nodes are chosen, the same pattern arises among the chosen nodes, for a churn during this period of $\Theta(\log |S|)$. Repeating this pattern $\Theta(n(n-1)n\frac{2}{k})$ times is sufficient to ensure that regardless of the pattern of failures representing the graph structure, a smaller S has lower churn, so the optimal S has $|S| = k$. ■

Fact 4 *Picking the k nodes with fewest failure and recovery events is a k-approximation for* Best Fixed Strategy.

Proof: Each event costs ≤ 1, while the optimal strategy must pick a set of nodes with at least as many events, each with cost $\geq \frac{1}{k}$. ■