Minimizing Churn in Distributed Systems

P. Brighten Godfrey, Scott Shenker, and Ion Stoica
UC Berkeley
Computer Science Division
{pbg,shenker,istoica}@cs.berkeley.edu

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.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; C.4 [**Performance of Systems**]: Fault tolerance; G.3 [**Probability and Statistics**]: Renewal theory

General Terms

Design, Theory, Reliability

Keywords

Churn, node selection, multicast, DHT

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 [21, 29]. 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 [27].

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 availability) 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, 18, 32].

SIGCOMM'06, September 11–15, 2006, Pisa, Italy.

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 a 1.3-5$\times$ reduction in churn over the best fixed strategy in the longer 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 performance of Random Replacement, showing that it is 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 is an easy way to reduce 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 strategies. Section 4 explores how the difference between RR and PL affects system design. We discuss why one would intentionally use RR 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 [34] — 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 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. We count a failure, and the selector's response to that failure, as separate events. So in a run of length T, if we begin with k nodes in use, two nodes fail simultaneously, and the selector 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 $i3$ [33]. We study 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 [14]). The complexity arises from the property that the cost of a failure depends on the number of nodes in use at the time.

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 after 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 replacement strategy (see [14]).

Agnostic Replacement strategies.

- *Random Replacement (RR)*: Pick k random initial nodes. After one fails, replace it with a uniform-random available node.
- *Passive Preference List:* Given a ranking of the nodes, after an in-use node fails, replace it with the most preferable available node.
- *Active Preference List:* Given a ranking of the nodes, after 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.

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
Skype	25	710	11.5 hours
Gnutella	2.5	1846	1.8 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.

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 [35]: 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 [7]: 51,662 desktop PCs within Microsoft Corporation were pinged every hour for 35 days beginning July 6, 1999.

Skype superpeers [15]: A set of 4000 nodes participating in the Skype superpeer network were sent an application-level ping every 30 minutes for about 25 days beginning September 12, 2005. As in the web sites trace, there are a number of short periods when many nodes appear to fail, due to network problems near the measurement site.

Gnutella peers [31]: 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, indicating that the Gnutella application was running. The majority of those hosts were usually down (see Table 1).

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

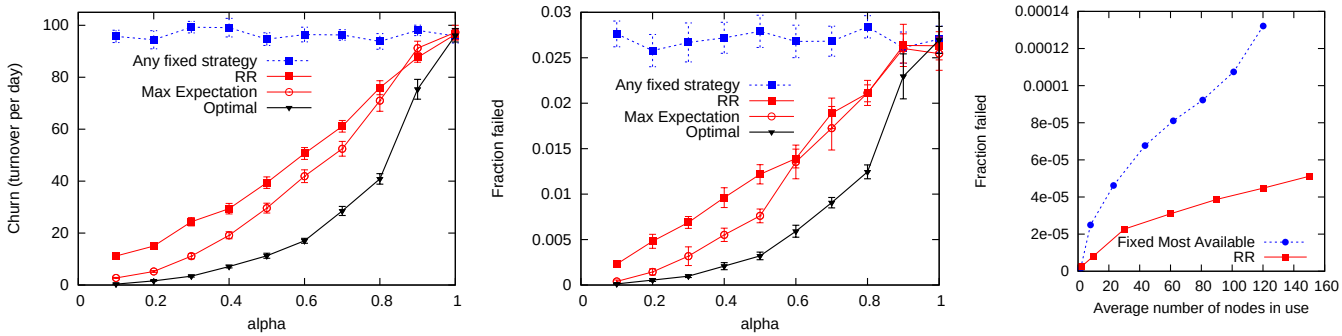


Figure 1: Churn (left) and fraction of requests failed in Chord (center) for varying α, with fixed $k = 50$ nodes in use and the synthetic Pareto lifetimes. Right: Chord in the PlanetLab trace (one trial per data point).

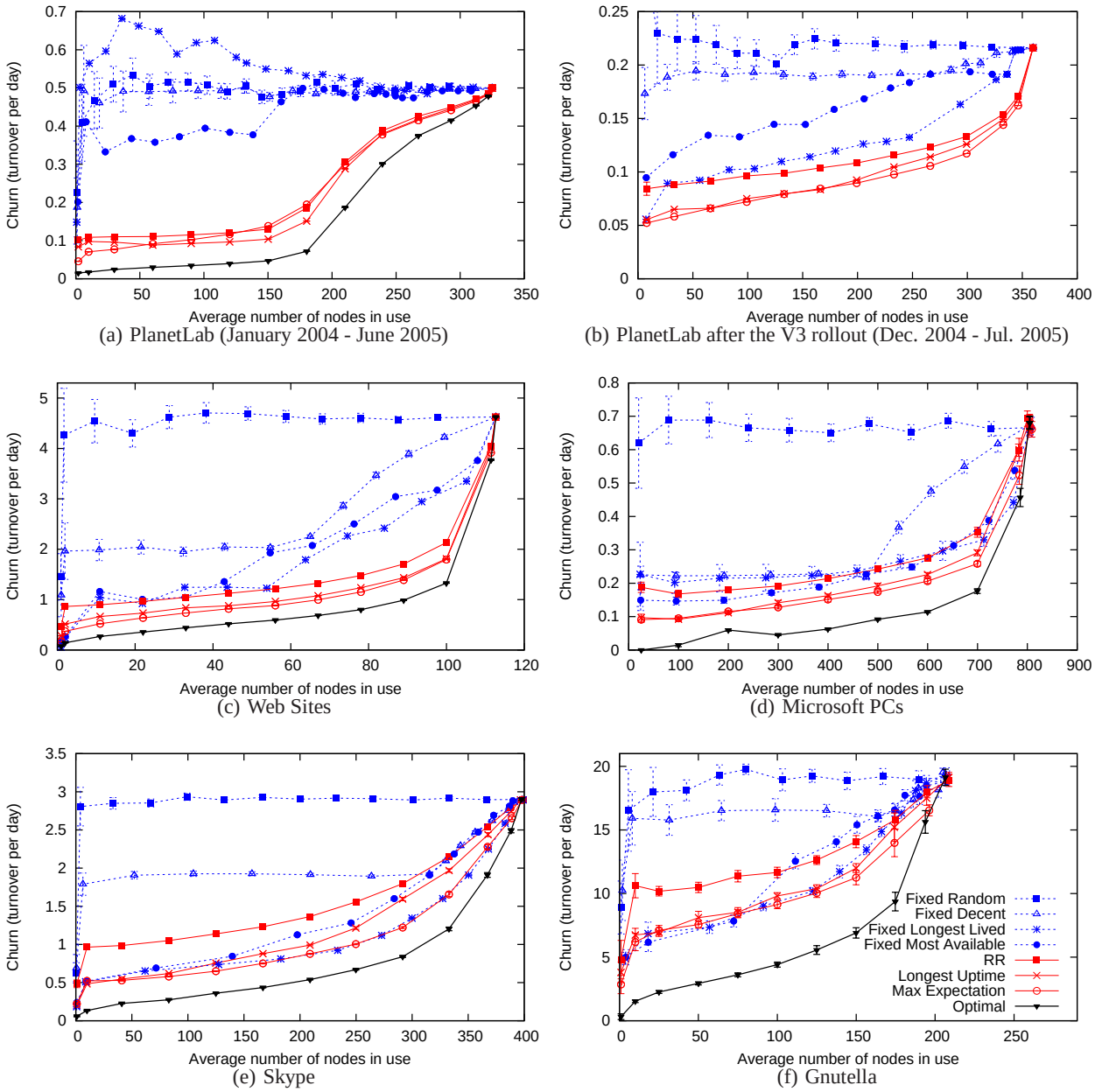
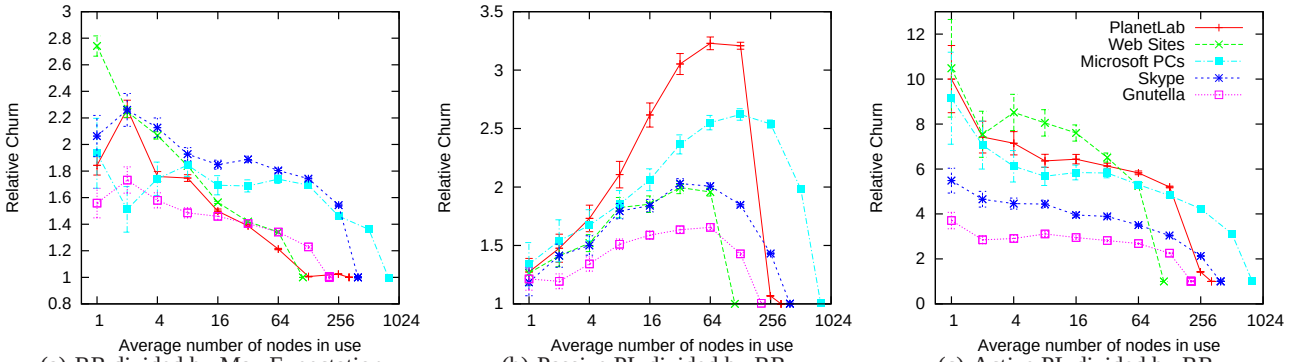


Figure 2: Churn with varying average number of nodes in use traces. The key at lower right applies to all six plots.



(a) RR divided by Max Expectation (b) Passive PL divided by RR (c) Active PL divided by RR

Figure 3: Churn of Random Replacement relative to other strategies. The key at right applies to all three plots.

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 $i3$ [33] codebase. Events are node joins and failures and datagrams being sent and 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. For the traces with more than 1000 nodes, 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 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

In the two peer-to-peer traces, the best fixed strategies match the performance of the best replacement strategies, perhaps since these traces are 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$\times$ improvement over the best fixed strategy, depending on k and 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(b)), 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 of 2 of Max Expectation.

Figure 4 illustrates the general behavior of the Preference List strategies via the PlanetLab trace. Active PL 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 PL pays to add preferred nodes as soon as they recover. Passive PL 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

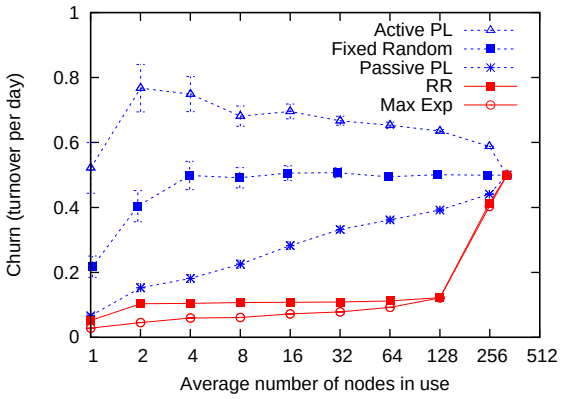


Figure 4: Preference List strategies in the PlanetLab trace. Note the log-scale x axis.

and pays for most of the failures.

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

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

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 $T \to \infty$. (If instantaneous recovery seems unrealistic, we note that the analysis of RR is identical in the model that each node has only a single session, and the total number of nodes is held constant by introducing a fresh node after each failure.)

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 PL 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 PL 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 $\frac{1}{kT}$ to switch back to v after its recovery, yielding churn $3/\mu$.

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 biased 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*

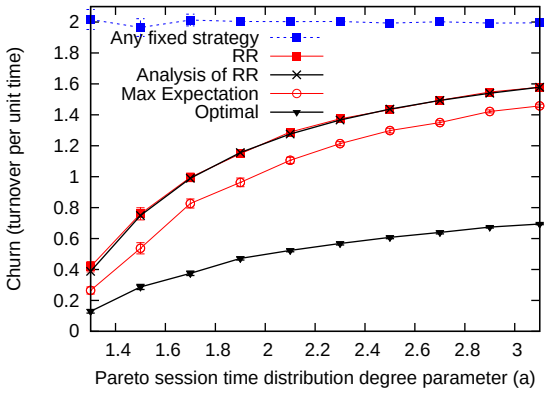


Figure 5: Simulation and analysis of churn with varying session time distribution, $n = 20$, and $\alpha = \frac{1}{2}$.

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 [14]. □

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 pick b so that $\mu = 1$. Even though the analysis assumes large n, it differs from the simulation by $\leq 1.5\%$ for $a \geq 1.5$. As a approached 1, convergence time in the simulation became impractical. For $a \leq 1$, $f(x)$ has infinite variance and does not satisfy the conditions of Theorem 1.

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 $y \in [0,1]$,*

$$E\left[X' \mid X' \geq x'\right] \quad \geq \quad E\left[X \mid X \geq x\right],$$

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

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. The Lorenz partial order is consistent with variance, in the sense that $X' \succeq X$ implies $\text{var}(X') \geq \text{var}(X)$.

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 [14].

4. APPLICATIONS

We have seen that Random Replacement consistently outperforms Preference List strategies (Section 2.5) essentially because it takes advantage of skewed 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. We also study how quickly RR converges to its steady-state churn rate.

In Section 4.2 we discuss how two classes of proposed DHT topologies behave like Active PL and RR, and show that randomizing the Chord topology decreases the fraction of failed lookups by 29% in the Gnutella trace.

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 [32], that a random parent selection algorithm was better than a certain longest-uptime heuristic.

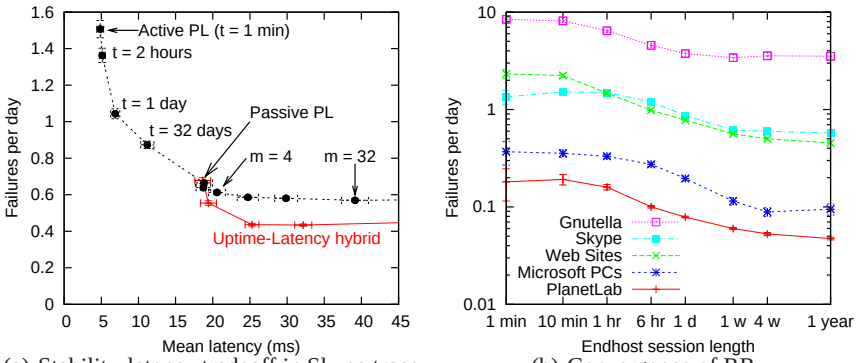
Finally, Section 4.4 explores two strategies for placing replicas in DHTs. Although a difference in their associated maintenance bandwidth had been previously observed, we show that part of the performance difference is due to behavior close to RR in one, and PL in the other.

4.1 Anycast

To give a simple instantiation of preference list and RR 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 obtains a list of the m servers to which it has lowest latency, perhaps by utilizing an anycast service such as [3, 12, 38], and connects to a random one of these m. Additionally, the endhost periodically probes for a closer server that may have newly joined, switching to such a server after some random delay in $[0, t]$ following the join.

We simulated the resulting number of failures in a simple simulator with events at the level of node joins and failures, as in Section 2. We do not count a switch as a failure. Latencies were obtained from a synthetic edge network delay space generator of Zhang et al [39], which is modeled on measurements of latency between DNS servers. The large availability traces were sampled down to 2000 nodes.

Figure 6(a) depicts the tradeoff between server failure rate and latency that results from various choices of the parameters m and t in the Skype trace. The upper-left point has $t = 1$ minute and $m = 1$, and corresponds to an Active PL strategy. As $t \to \infty$, we



(a) Stability-latency tradeoff in Skype trace (b) Convergence of RR

Figure 6: Anycast simulation results.

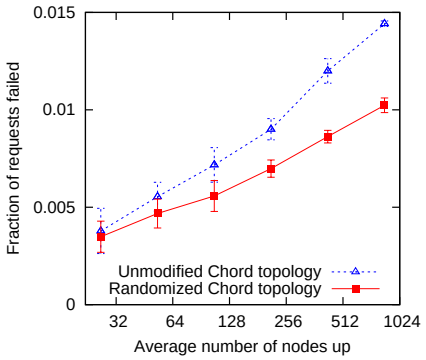


Figure 7: DHT neighbor selection simulation in Gnutella trace.

move to a Passive PL strategy, and failure rate decreases by roughly 56% (46-72% in the other traces). Increasing m results in an RR strategy and decreases failure rate by a further 13% (13-21% in the other traces). This latter decrease is modest since we are only selecting one node at a time (compare with Figure 3(b) with $k = 1$ nodes in use). However, this may be useful if, for example, a mean latency of 40 ms were acceptable to the application in question. We also simulated a hybrid strategy which used $t = \infty$ and selected the replacement server which minimized $w \cdot latency - (1 - w) \cdot uptime$. As w decreases from 1 to 0, the strategy moves from Passive PL to Longest Uptime. In the Skype trace, this additional uptime information reduces failure rate by about 24% below the randomized strategy with $m = 32$.

Of course, 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.

So far we have assumed an endhost which continually selected a server over the entire trace. Suppose now that the endhost arrives at a random time, uses RR server selection, and departs after a given session length ℓ. Figure 6(b) shows that when ℓ is small, the endhost experiences the mean server failure rate, as in Active PL. Intuitively, the endhost departs before it makes full use of the session of the server it selected. The failure rate converges as ℓ approaches the mean session length of a RR-selected server, decreasing by 2.3×-5.1× depending on the trace.

As an example, some Skype peers which are behind NATs select superpeers through which to relay voice calls. Since 90% of relayed Skype calls last less than 36 minutes [15], if the peers select relays randomly, these calls would see roughly the mean superpeer failure rate (one failure every 16 hours). However, one could imagine designing the superpeer network to maintain a set of randomly selected "super-superpeers" through which interruption-sensitive voice calls are routed when possible. Such a design would result in a failure rate similar to that of a persistent endhost session (one failure every 42 hours).

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. 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 two different ways of selecting long-distance neighbors.

Deterministic and randomized topologies

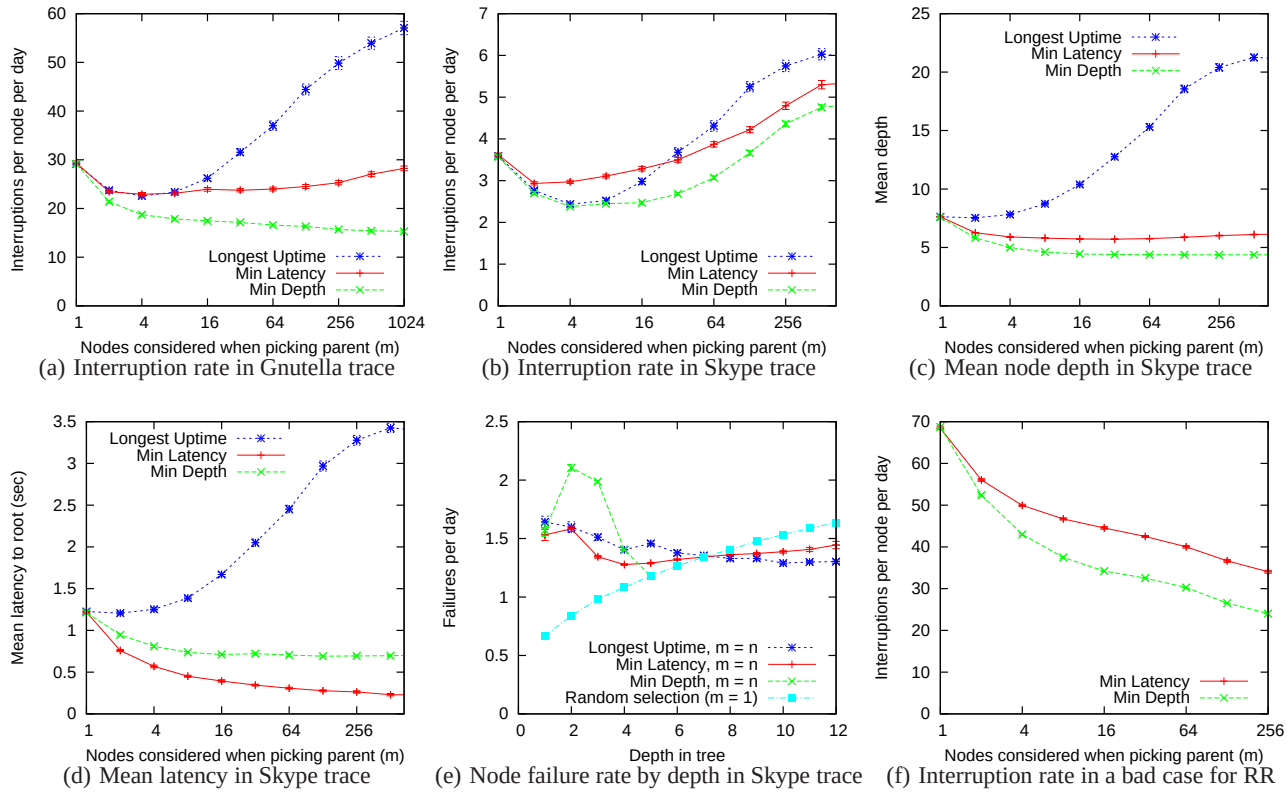
In the first class of topologies, used in Chord [34], CAN [26], and others [17, 25], each node v maintains links to the owners of certain other IDs which are a deterministic function of v's ID. For example, Chord's keyspace is $\{0, \ldots, N-1\}$, where $N = 2^{160}$, and node v maintains links called *fingers* to the owners of $id(v) + 2^i \pmod N$ for each $i \in \{0, \ldots, (\log_2 N) - 1\}$. 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, a key x is owned by the node whose ID most closely follows x in the (modular) keyspace. 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 $id(w)$.

In the second class of topologies, links are chosen randomly. Symphony [22] was the first design to explicitly choose random neighbors, but some other topologies have enough underlying flexibility [16] 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[1] 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.

Results and implications

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 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 $\tilde{n} \approx 850$, the randomized topology has 29% fewer failed requests due to the lower finger failure rate. The randomized topology also had very slightly longer routes (7.6% longer for $\tilde{n} \approx 27$ but decreasing to just 0.8% longer for $\tilde{n} \approx 850$).

[1]This topology was studied in [23] in the context of route length.



(a) Interruption rate in Gnutella trace (b) Interruption rate in Skype trace (c) Mean node depth in Skype trace

(d) Mean latency in Skype trace (e) Node failure rate by depth in Skype trace (f) Interruption rate in a bad case for RR

Figure 8: Multicast simulation results.

Leonard et al [19] analyzed the resilience of several P2P systems including Chord in a stochastic model, deriving the expected time until a node is disconnected from the network. The analysis assumed that the selection of a neighbor is independent of its age — essentially an RR strategy. We have now seen that Chord, as well as the other deterministic DHTs, in fact follow a PL strategy. Since time-until-disconnection depends superlinearly on finger failure rate, the assumption of [19] would result in a significant overestimate of the resilience of the standard Chord protocol, as well as the other deterministic DHTs.

Several advantages of flexible, non-deterministic topologies are well known, most notably the ability to use proximity neighbor selection to reduce latency [16] – which, depending on the implementation, may result in a latency-based PL strategy. In the work most similar to this section, Ledlie et al [18] 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 *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.

Simulation setup

We closely follow the simulation scenario of Sripanidkulchai et al [32]. We deal with a single-source multicast tree, whose 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 and has available bandwidth for another child. 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 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 with $m = 100$ were also simulated in [32], in traces which had peak sizes of $1,000$-$80,000$ nodes up.

Unfortunately, we could not test under the traces and node bandwidth bounds used in [32] since their data is not publicly available. Instead, we use the traces of Section 2.3, latencies from Zhang et al [39] as in Section 4.1, and uniform node capacities: each node accepts at most $d = 4$ children unless otherwise stated. In [32], after a node fails, its descendants which are contributing more resources are allowed to rejoin before freeriders. Since we use homogeneous capacities, we have nodes rejoin in random order. Finally, a minor difference is that 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 mean node depth (number of hops from each node to the root) and mean latency through the tree to the root. We take the mean of these metrics over all samples within each trial, and then over all trials.

Results

We begin by discussing the Min Latency strategy. We will then confirm and offer additional interpretation of two results of [32] regarding the Min Depth and Longest Uptime strategies.

Figures 8(a) and 8(b) show that optimizing latency both helps and hurts the number of interruptions. The case $m = 1$ is random parent selection. As we begin increasing m, latency to the root decreases (Figure 8(d)) but there is a side effect of reducing tree height (Figure 8(c)), which reduces the interruption rate (22% in Gnutella, 19% in Skype) because there are fewer opportunities for failure along a node's path to the root. But for $m \geq 4$ the mean node depth is essentially constant and the trees become less stable, with interruptions increasing 22% in Gnutella and 86% in Skype.

The interior structure of the trees reveals the proximate cause of this instability. Figure 8(e) 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 rejoin in random order after an ancestor fails. 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 RR performed *worse* than PL strategies, performance should *improve* as $m \to n$. By Corollary 1, such an (unrealistic) bad case is when session times are essentially constant, e.g. uniform in $[9, 11]$. Figure 8(f) shows that in this case, interruption rate is indeed a monotonically decreasing function of m.

We now discuss two results of Sripanidkulchai et al [32]. First, in tests using a fixed m, they found that Min Depth best optimized stability among the strategies they tested, which is true in most cases we tested (e.g. all of Figure 8(a)). Interestingly, we find that even Min Depth can benefit from some randomization as well, with less than half the interruption rate at $m = 4$ than $m = n$ in Skype. This effect also appeared in the Microsoft PCs trace and to a lesser extent in PlanetLab, but not in Gnutella or Web Sites.

Second, Sripanidkulchai et al [32] 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,b) for sufficiently large m. However, we also find that using Longest Uptime, the nodes near the root are less stable than in the $m = 1$ case (Figure 8(e)).

In fact, neither Min Depth nor LU optimizes exactly the right metric. Min Depth ignores the stability of nodes on the path from the parent to the root, and LU ignores the length of that path and the stability of all ancestors except the parent. 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 the other heuristics.

4.4 DHT replica placement

In this section we compare two common strategies, *Root Set* and *Random*, for managing file replicas in distributed hash table-based storage systems. The metric we study is the rate at which new replicas are created, which directly affects maintenance bandwidth.

Replica management strategies

In DHT-based storage systems, nodes are assigned identifiers (IDs) in a keyspace. Each stored file or object o 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.

The *Root Set* strategy for placing those replicas is used in slightly varying forms by DHash [8], PAST [30], Bamboo [29], 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. In both cases a file transfer is not necessary if the node in question already has the file. This occurs when a node returns after a *transient* failure, such as a network outage, which does not affect files stored on disk.

The *Random* strategy is used by Pond [28], Total Recall [5], and Weatherspoon et al [37]. The root $r(o)$ stores a directory of all available replicas of o, which may be on any node in the system. (The directory is assumed to be small relative to the size of an object replica, so the cost of replicating it — with, for example, the Root Set strategy — is negligible.) Random has two parameters, k and $f \in (0, 1]$. Repair is only initiated when the number of available replicas falls below $\lceil f \cdot k \rceil$, at which point new replicas are created until k are available. This "lazy replication" provides a buffer so the system reacts to transient failures less frequently.

Simulation setup

It has been previously observed [5, 37] that Random significantly outperforms Root Set, and this has been attributed to a number of disadvantages of Root Set. Root Set might "forget" about replicas that end up outside the root set; it replicates when nodes arrive, rather than only in response to failures; and in some implementations it lacks the lazy replication threshold f.

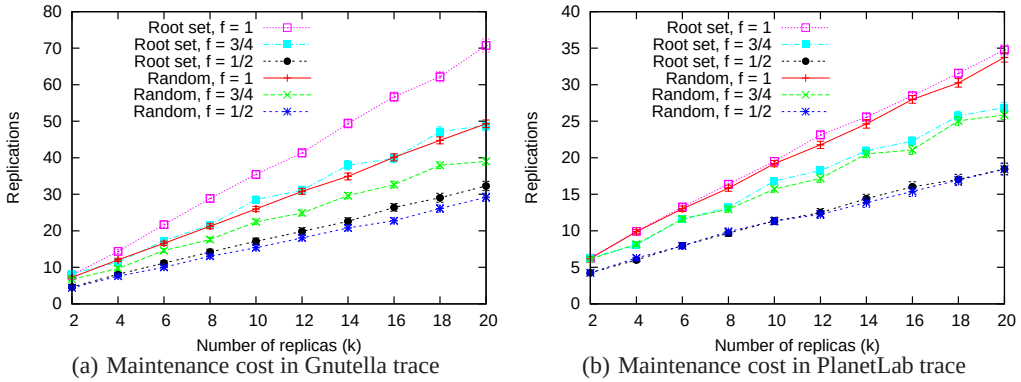
Our goal is to quantify the impact of another difference: the choice of node on which to place a replica once it is created. To compare the strategies on equal footing, we modify the Root Set strategy so that it monitors all replicas in the system, does not replicate in response to node joins, and uses the lazy replication threshold f. The remaining difference is that Root Set places each new replica on the first node available node in the root set (i.e., Passive PL) while Random follows RR node selection.

As before we use a simulator with events at the level of node joins, node failures, and file replications. Since our traces do not include information about data loss associated with failures, we assume no data loss, which provides a lower bound on the permanent failure rate. We assume files are written to the system at the beginning of each trial. We measure the mean number of replications used to maintain each file after the initial write.

Results

Figure 9 compares the two strategies with $f \in \{1, \frac{3}{4}, \frac{1}{2}\}$ in two representative traces, PlanetLab and Gnutella, for $k \in \{2, \ldots, 20\}$. (Note that each "replica" may be an erasure-coded fragment of the file, so $k = 20$ is reasonable; in fact, Pond uses $k = 32$.) At $f = 1$ and $k = 20$ in Gnutella, Random requires 30% fewer replications than Root Set, and in fact Random with $f = 1$ is as good as Root Set with $f = \frac{3}{4}$. However, this difference diminishes as f is decreased, and the strategies differ little in PlanetLab.

Several limitations of the traces likely underestimate the long-term benefit of Random over Root Set. Once transient failures are largely masked, the strategies compete at the level of permanent failures. However, none of the traces is long relative to the perma-



(a) Maintenance cost in Gnutella trace (b) Maintenance cost in PlanetLab trace

Figure 9: Replica placement simulation results.

nent failure rate. For example, among Gnutella nodes that were up at some point in the first half of the trace, only 33% were absent in the second half, and that fraction was smaller in the other traces. Additionally, we have underestimated the permanent failure rate by assuming no data loss. As a consequence, it is likely that Random has not yet converged in these simulations (see Section 4.1).

Recently, Tati and Voelker [36] observed an effect of the Random strategy: nodes with higher average availability will be selected to receive objects more frequently, and will also likely have higher average availability in the future. This effect is closely related to RR (compare with the intuition in Section 3.3) and undoubtedly contributes to the difference between strategies that we have observed. Separating these effects, as well as obtaining better data on permanent failures, remains an interesting area of future research.

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 one 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 misbehavior 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 when the most stable nodes are sought after simultaneously by multiple agents, such as peers in a P2P system or users in a shared infrastructure like PlanetLab?

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 correspondence 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 known by the same name.

There has been a substantial amount of work on analysis of page replacement algorithms including LRU and RR; see e.g. [9–11] and the discussion in [10]. 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 in which node session times are exponentially distributed (with possibly unequal means). Thus a major difference is that our model analyzed in Section 3 is not limited to memoryless session times.

Longest Uptime is a common heuristic which has been studied in contexts including DHT neighbor selection [18], selecting superpeers [13], and selecting parents in an overlay multicast tree [32]. The Accordion DHT [20] 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 [24] used sophisticated statistical techniques to predict future node uptime, and experimented with placing file replicas in Chord on successors with greatest predicted time to live.

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 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 future work is to demonstrate these differences in a deployment of a large distributed system.

Acknowledgements

We thank the authors of [2, 7, 15, 31, 35] 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 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 by NSF grants ANI-0133811, ANI-0085879, ANI-0205519 and ANI-0225560, a NSF Graduate Research Fellowship, and British Telecom.

8. 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. ACM 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*, 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] 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.

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

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

[10] 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.

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

[12] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. Oasis: Anycast for any service. In *NSDI*, 2006.

[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)*, 2003.

[14] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. Technical Report EECS-2006-25, EECS Department, University of California, Berkeley, 2006.

[15] S. Guha, N. Daswani, and R. Jain. An experimental study of the Skype peer-to-peer VoIP system. In *IPTPS*, 2006.

[16] 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.

[17] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proc. IPTPS*, 2003.

[18] 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.

[19] D. Leonard, V. Rai, and D. Loguinov. On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. In *SIGMETRICS*, 2005.

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

[21] 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.

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

[23] G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized P2P networks. In *STOC*, 2004.

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

[25] M. Naor and U. Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *Proc. SPAA*, 2003.

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

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

[28] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. USENIX File and Storage Technologies (FAST)*, 2003.

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

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

[31] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. MMCN*, 2002.

[32] 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. ACM SIGCOMM*, 2004.

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

[34] 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. SIGCOMM*, 2001.

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

[36] K. Tati and G. M. Voelker. On object maintenance in peer-to-peer systems. In *Proc. IPTPS*, 2006.

[37] 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.

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

[39] B. Zhang, T. S. Ng, A. Nandi, R. Riedi, P. Druschel, and G. Wang. Measurement-based analysis, modeling, and synthesis of the Internet delay space, 2006. http://www.cs.rice.edu/ bozhang/synthesizer.html.