

Measuring the Mixing Time of a Network

Xenofon Foukas

School of Informatics

University of Edinburgh

Edinburgh, United Kingdom

Email: x.foukas@sms.ed.ac.uk

Antonio Carzaniga

Faculty of Informatics

University of Lugano

Lugano, Switzerland

Email: antonio.carzaniga@usi.ch

Alexander L. Wolf

Department of Computing

Imperial College London

London, United Kingdom

Email: a.wolf@imperial.ac.uk

Abstract—Mixing time is a global property of a network that indicates how fast a random walk gains independence from its starting point. Mixing time is an essential parameter for many distributed algorithms, but especially those based on gossip. We design, implement, and evaluate a distributed protocol to measure mixing time. The protocol extends an existing algorithm that models the diffusion of information seen from each node in the network as the impulse response of a particular dynamic system. In its original formulation, the algorithm was susceptible to topology changes (or “churn”) and was evaluated only in simulation. Here we present a concrete implementation of an enhanced version of the algorithm that exploits multiple parallel runs to obtain a robust measurement, and evaluate it using a network testbed (Emulab) in combination with a peer-to-peer system (FreePastry) to assess both its performance and its ability to deal with network churn.

I. INTRODUCTION

The performance of distributed algorithms often depends on global properties of the network, where the term “network” here refers broadly to a generic interconnection of computational nodes, including wide-area networks, data-center networks, ad hoc networks, and many kinds of overlay networks. One of these critically important properties is the *mixing time* of the network, a global property that indicates how fast a random walk gains independence from its starting point [1].

Gossip algorithms and decentralized measurement algorithms, which are used for such things as sampling or for computing aggregate node metrics, perform better on fast-mixing networks than on slow-mixing networks. Mixing time can also play a role in security and privacy: botnet peer-to-peer overlays tend to be fast mixers, which allows their presence to be distinguished from the underlying network’s background traffic [7]; in social networks, honest friends tend to exhibit fast mixing times, whereas the presence of malicious users perpetrating a Sybil attack through multiple fake identities tend to slow the mixing time of the network [9]. It is therefore useful to have an effective and efficient general-purpose facility to measure mixing time.

With full knowledge of the network graph, it is possible to measure the mixing time exactly and with a simple local computation. In particular, the mixing time can be computed from the second largest eigenvalue of the adjacency matrix of the network graph. However, such global knowledge is expensive to obtain and maintain in precisely the cases where distributed algorithms are most useful. We therefore propose a

mechanism that provides a good *estimate* of the mixing time in a decentralized and efficient manner.

The mechanism we propose is based on a recent spectral estimation algorithm due to Carzaniga, Hall, and Papalini [2]. In essence, their algorithm computes a surrogate of the adjacency matrix using minimal state and limited local communications, and then uses the largest eigenvalues of the surrogate matrix as an estimate of the global matrix. However, the accuracy of their algorithm is negatively affected by changes in the network, or “churn” events. Furthermore, their algorithm has been evaluated only in simulation.

We make the following novel contributions. First, we extend the basic scheme introduced by Carzaniga, Hall, and Papalini to provide a churn-resilient *mixing-time measurement service*. At a high-level, our idea is to combine parallel, independent estimates that would individually be susceptible to the same churn events, but in different ways, collectively resulting in a more accurate estimate. Second, we design and develop a protocol that can be used to realize the service. Third, we give a paradigmatic implementation of this protocol for use with a peer-to-peer overlay network, FreePastry.¹

We evaluate our design and implementation on the Emulab network testbed, obtaining three important results. First, we confirm the basic simulation results of Carzaniga, Hall, and Papalini in a concrete implementation on an emulated network. Second, we validate the extensions to the scheme, showing that it achieves good estimates even in the presence of churn. Third, we demonstrate the service by applying it in the well-known distributed measurement algorithm *push-sum* [4].

II. BACKGROUND

We consider a general, multi-hop network in which each node connects to a set of neighbors. This set may be more or less stable, a node may or may not maintain direct network connections to its neighbors (i.e., nodes may be neighbors in an overlay), and those direct connections may be uni- or bi-directional. Thus, we model the network as a generic directed graph that can represent static networks or highly dynamic networks such as peer-to-peer systems. Our only assumption about the network graph is that it is *ergodic*. This is reasonable, since all connected networks of practical interest (other than, say, a bipartite network) are ergodic.

¹<http://www.freepastry.org/>

Given this model, we consider the problem of measuring the mixing time, which is essentially a metric for how quickly a random propagation of information stabilizes throughout a network. A bit more formally, and to better define our objective, we consider the case of a random walk. Since the network is ergodic, the probability $x_v(t)$ that a random walk visits a node v at hop t , as t approaches infinity, converges to a value π_v independently of the starting node of the walk. The vector of probabilities π is called the *asymptotic (steady state) distribution* of the walk. In this case, given a distance ε , the mixing time τ_ε is the minimal number of hops after which the visiting probabilities $x(t > \tau_\varepsilon)$ approximate the asymptotic distribution π by at most ε . The mixing time thus depends on the structure of the network and of the random walk. Specifically, the mixing time can be computed from the second largest eigenvalue of the transition matrix of the walk $A = (a_{uv})$, where a_{uv} is the probability that a walk would hop from node v to node u (and, therefore, $a_{uv} > 0$ only when there is an edge from v to u in the network graph).

The structure of random walks is representative of several randomized gossip algorithms [1] whose performance depends on the second largest eigenvalue of a characteristic matrix defined by the algorithm and related to the adjacency matrix of the network graph. We therefore consider a network system or algorithm \mathcal{A} in which each node v defines a column of weights $(a_{.v})$, associated with v 's outgoing edges, that collectively define the characteristic matrix $A = (a_{uv})$.

Our goal, then, is to provide a decentralized and efficient mechanism to measure the second largest eigenvalue of A .

A. Spectral Estimation

The spectral analysis of graphs has many interesting applications. The best-known example is probably the PageRank algorithm, but there are also applications in the field of networking with corresponding algorithms to compute the spectral properties of a network in a decentralized manner. For example, EigenTrust [3] is a reputation management algorithm for peer-to-peer networks that computes the principal eigenvector of a matrix containing trust values of the nodes using a distributed power method. Kempe and McSherry [5] propose a generic and decentralized algorithm to compute the principal eigenvectors of a symmetric weighted adjacency matrix of the network. Notice, however, that all these algorithms compute the top *eigenvectors*, which contain more information, and require significantly more memory and communication resources, than what we need for the estimation of the mixing time.

We build upon a simpler spectral estimation algorithm developed by Carzaniga, Hall, and Papalini [2] that computes the top *eigenvalues* of the adjacency matrix of the network. The algorithm views the network as a dynamic system defined by the following state-space equations:

$$x(t+1) = Ax(t) + Bu(t) \quad (1)$$

$$y(t) = Cx(t) \quad (2)$$

This is a discrete-time, linear, time-invariant, single-input, single-output, deterministic system. That is, the state of the system $x(t) \in \mathbb{R}^n$ evolves in steps $t = 1, 2, \dots$ (*discrete time*) through a transformation defined by a matrix $A \in \mathbb{R}^{n \times n}$ (*linear*) that is immutable over time (*time invariant*). The system is stimulated by a scalar input signal $u(t) \in \mathbb{R}$ (*single input*) that feeds into the state through a vector $B \in \mathbb{R}^n$, and produces a scalar output signal $y(t) \in \mathbb{R}$ (*single output*) as a linear combination of the state, with coefficients $C \in \mathbb{R}^n$. The system is not subject to error signals (*deterministic*).

We denote by $h(t)$, for $t = 1, 2, \dots$, the *impulse response* of the system starting from the quiescent state $x(0) = 0$. Thus, $h(t)$ is the output of the system when the input is the unit impulse ($u(0) = 1$ and $u(t) = 0$ for $t > 0$).

The estimation can be thought of as a synchronous distributed algorithm in which each node v holds a scalar value $x_v(t)$ corresponding to a component of the state of a dynamic system, and executes the following steps:

- 1) *Initialization*: v initializes its state variable x_v to either 0 or 1, chosen uniformly at random, and then records the initial value $h_v(1) \leftarrow x_v$.
- 2) *Distributed computation of the impulse response*: for $k - 1$ rounds $t \leftarrow 2 \dots k$, v sends value $w_u = x_v a_{uv}$ to each out-neighbor u , updates its state $x_v \leftarrow \sum w_v$ with the sum of all values w_v received from its in-neighbors, and then records each new value $h_v(t) \leftarrow x_v$. Effectively, v computes k values of the impulse response $h_v(t)$ of a system $x(t+1) = Ax(t) + Bu(t)$; $y(t) = Cx(t)$, where A is the matrix of system \mathcal{A} , B corresponds to the global state of the system initialized in step 1, and C_v is a row vector of all zeroes except for a 1 in position v .
- 3) *Realization of a surrogate system*: v uses $h_v(t)$ with Kung's algorithm [6] to compute a matrix \hat{A}_v that defines an approximate realization of the system.
- 4) *Eigenvalues of the surrogate system*: v computes the second largest eigenvalue $\hat{\lambda}_v$ of \hat{A}_v using a standard numeric algorithm.
- 5) *Neighborhood gossip round*: v exchanges its estimate $\hat{\lambda}_v$ with its neighbors and uses the median value $\bar{\lambda}$ of its estimate plus all the estimates it receives from its neighbors.

B. Advantages and Limitations

Notice that the algorithm is completely decentralized: node v holds the column $(a_{.v})$ of the original matrix A , which corresponds to v 's local view of the whole system \mathcal{A} , but v does not know the rest of A . And yet v can compute an estimate $\bar{\lambda}$ that, as it turns out, approximates very well the second largest eigenvalue of the actual (global) matrix A . Notice also that each node v uses $k \ll n$ rounds of local communication for a network of size n . In other words, the Carzaniga, Hall, and Papalini algorithm is quite efficient and precise.

The flaw in this algorithm, however, is that it assumes the global matrix A remains unchanged during the k rounds of the computation of the impulse response. This means that the algorithm does not account for the practical problem of

network churn. Churn events, particularly a node leaving the network, effectively change the matrix A , negatively impacting the quality of the estimation. We elaborate on this flaw, and discuss how we address it, in the next section.

III. COMPENSATING FOR NETWORK CHURN

Carzaniga, Hall, and Papalini observed the significant inaccuracy of their algorithm in the presence of network churn, but offered no solution to this problem [2]. Experimentally, we observe (Section VI) that churn events are particularly disruptive when they occur close to the end of the computation of the impulse response. This is intuitive, since a change during the early rounds of the computation would likely result in a spectral estimation that contains enough information about the new topology to wash out the effects of the change, whereas a change during the later rounds might not be fully observed from some locations in the network, resulting in poor local estimates at those locations based on outdated matrices. Obviously, the uncertainty that churn introduces to the accuracy of the estimate greatly hinders the usefulness of a measurement service in realistic settings.

A simplistic way to approach the challenge of network churn would be to assume the availability of a global “sanity check” based on some external knowledge of the network’s properties that would indicate whether or not a given estimate is acceptable. If the estimate turned out to be bad, the whole process could be repeated. However, introducing such an oracle amounts to assuming that a churn event does not violate those network properties and/or making conservative judgements on the viability of the estimate. The latter, in particular, implies possibly unnecessary, sequential re-executions of the algorithm, thereby leading to wasted network traffic and excessive delays in obtaining a result; such delays would also result from simply requiring that each execution always consists conservatively of a large number of rounds. None of these approaches is practical in realistic networks of any reasonable scale.

Our approach, instead, is to take advantage of the fact that otherwise inaccurate estimates might still spread some useful information through the network. Further, we make use of multiple measurement runs, but execute them roughly in parallel rather than sequentially, and combine them to produce a superior result. We refer to an execution of the basic spectral estimation algorithm as a *line* and illustrate its use in Figure 1. Each of the three horizontal bars in the figure represents one line, initiated at three different times and formed from a series of rounds, ending with the computation of an estimate as described in Section II. The shaded rounds contribute to the impulse response computation and, therefore, heavily influence the accuracy of the mixing-time estimate. A light shade indicates that the round contributes to a good estimate, whereas a dark shade indicates the opposite.

At some point in this example, a churn event occurs (e.g., a failure in the network or the appearance of a new peer node), resulting in a change to the network topology. In the case of the first line, the event occurs late in the line and, therefore,

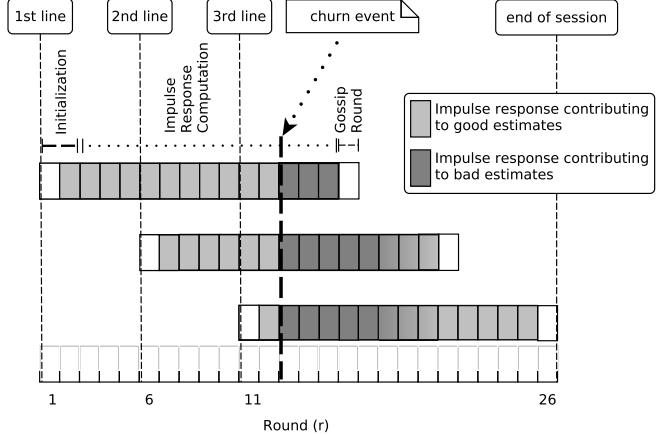


Fig. 1: Execution of $l = 3$ parallel, partially overlapping lines, each consisting of a total of 16 rounds ($k = 15$).

there is no time for it to acknowledge the effect of the event, thereby resulting in a bad estimate. The other two lines, while affected by the event, nevertheless manage to recover before terminating and result in a good estimate.

But the question remains, how can we know which of the estimates produced by the individual lines is acceptable? The answer is to compare their results using, for example, an approximate voting scheme. This would reveal the presence of an inconsistency, if it existed, effectively serving as a kind of distributed failure detector for the measurement facility (and, likewise, a detector of churn events). Moreover, it could reveal whether the effects of a churn event had eventually been overcome.

Our main idea is therefore to compensate for network churn by computing a series of partially (but not completely) overlapping impulse responses. This way, if a churn event occurs, a few of the impulse responses—those that terminate right after the churn event—might result in a faulty measurement, but the majority of them probabilistically will be accurate enough to obtain a good estimate. It should be noted that this idea does not always guarantee the accuracy of the final estimate, but as the experiments in Section VI demonstrate, it can be effective in many cases.

The tuning parameters of our approach involve the number of parallel instances and their length in rounds. Moreover, we must invent a means to manage the parallel instances and merge their results. We discuss the design of a distributed protocol for realizing the mixing-time measurement service in the next section.

IV. PROTOCOL

Our protocol defines the local state maintained by each node in the network, and the messages exchanged to evolve and share that state among the nodes. The protocol builds upon the spectral estimation algorithm described in Section II-A as extended above to compensate for churn.

A. General Design

We refer to the process of obtaining a measurement of the network's mixing time, including all the actions taking place from when a service request is made until the measurement actually becomes available, as a *session*. A session is composed of l lines (i.e., instances of the spectral estimation algorithm) that execute in parallel and are initiated at different points in time so as to only partially overlap. Each line consists of $k+1$ rounds (k rounds for steps 1 and 2 of the algorithm plus 1 for the gossip round of step 5). Both l and k serve as session parameters. The total number of rounds in a session, from the initiation of the first line to the termination of the last line, is denoted by r , and since lines partially overlap, it follows that $r < l(k+1)$.

Lines are staggered and evenly spread within a session, separated by a fixed interval of $\lfloor k/l \rfloor$ rounds. This leads to a session length of $r = (l-1)\lfloor k/l \rfloor + k+1$ rounds. Using the example of Figure 1, we can see that for $l = 3$ and $k = 15$ a new line is introduced every 5 rounds (rounds 1, 6 and 11) and hence r is 26.

Each line is composed of three phases: *Initialization*, *Impulse Response Computation*, and *Gossip Round*, derived from the steps in the spectral estimation algorithm. The Initialization phase is composed of two rounds (corresponding to step 1 and the first round of step 2). In the first (local) round, each node v chooses a value $x_v(1)$ uniformly at random and sets it as its initial impulse response, $h_v(1)$. Then for the second round, it sends the value $x_v(1)a_{uv}$ to all its out-neighbors u , and gathers the corresponding values sent by adjacent nodes. The sum $x_v(2)$ of the received values is then computed and is set as the second impulse response, $h_v(2)$.

For each round $i = 3..k$ in the Impulse Response Computation phase (corresponding to the remaining rounds of step 2), each node v sends the value $x_v(i-1)a_{uv}$ to its out-neighbors and waits to receive the corresponding values of its in-neighbors u . It uses the received values to compute the new impulse response as the sum of those values. Once this phase terminates, the impulse responses gathered are used to produce an estimate of the network's properties (steps 3 and 4 of the algorithm).

The estimate for each line at each node is obtained through a simple Gossip Round (step 5), in which neighbors exchange their estimates and then take the median of those estimates as the line's estimate.

Our protocol adds one additional step, also performed locally at each node. This step, the *Vote*, takes advantage of the multiple lines within a session to compute a median value representing the consensus among the lines. This value is then the final measurement reported locally from a node back to the hosted entity that issued the service request.

B. Protocol Messages

A session is initiated when a service request, containing the desired number of lines l and number of rounds k for each line, is made to one of the network's nodes. The *initiator*, the node that receives the service request, configures a new

session based on the two parameters, assigns the session a unique ID, and sets the first line to its Initialization phase. In an epidemic fashion, the *initiator* then sends a message of type INIT to all of its out-neighbors, containing the information required for them to locally initiate (i.e., participate in) the new session. The essential contents of the INIT message are shown in Figure 2. Each node receiving the INIT message performs the same set of actions as the *initiator*, until all the nodes are aware of the new session, using the session ID as a "mark" to terminate the process.

INIT	NEXT	GOSSIP
session ID	session ID	session ID
total lines (l)	line number	line number
line number	round number (i)	estimate ($\hat{\lambda}_v$)
rounds in line (k)		
$w_u = x_v(1)a_{uv}$		

Fig. 2: Protocol messages and their contents.

The INIT message is used not only to establish new sessions but also new lines within a session. When the time comes for a new line to be created, the *initiator*, as before, sets the new line to its Initialization phase and sends INIT messages to its out-neighbors using the same session ID, but with an incremented line number. If a node receives an INIT message bearing a known session ID, but an unknown line number, a new line is created locally within the context of that session. The process continues epidemically. Note, however, that a node will not participate in the current session if it joins the network any time after the Initialization phase of the first line completes. This is not a problem, since the presence of the node will be accounted for in a subsequent session.

During the Impulse Response Computation phase, a series of NEXT messages carry the values used to compute the next impulse response to out-neighbors, along with context information for the session, line number, and round. Similarly, GOSSIP messages are used in the Gossip Round to transport a list of computed eigenvalues to out-neighbors.

Therefore, the total number of messages exchanged in a network with m edges is $ml(k+1)$. In practice, this communication overhead could be greatly reduced, first by multiplexing the values of different rounds and lines within a session onto a single message, and further, since messages are very short, by attaching protocol messages as "piggyback" onto regular traffic.

C. Discovering and Managing In-Neighbors

One critical aspect of the spectral estimation algorithm is an assumed knowledge of the in-neighbor set. In particular, each node needs to know the number of expected incoming values in each round so that it can terminate that round and proceed to the next one. In practice, peer-to-peer and other distributed systems typically do not require nor record such information. Moreover, we do not wish to limit our service to work only for networks having bi-directional channels. Therefore, discovering and maintaining a list of in-neighbors becomes an integral protocol function for our purpose.

Under our protocol, a node discovers its in-neighbors for each line individually within a session during the Initialization phase. During this phase, each node chooses a value uniformly at random as its initial impulse response and sends an INIT message to its out-neighbors. Since each node receiving an INIT message for a new session with a particular ID will in turn epidemically send INIT messages to all its out-neighbors, the node will eventually receive corresponding INIT messages echoed back from all its in-neighbors.

Pragmatically, we need to use a timeout in order to locally terminate the discovery process for a node, since we do not assume global knowledge such as the network diameter. While this cannot guarantee that all in-neighbors will be discovered (e.g., due to delayed INIT messages), we must assume that a reasonable timer value can be found to yield a probabilistically accurate record of the in-neighbor set for a large portion of the network without unduly compromising performance.

We have another pragmatic need for timers in our design, namely to avoid deadlocks under failure. A deadlock can occur when a message needs to be received from some in-neighbor present during the Initialization phase, but in a failed state at some later round. To overcome this we employ a simple failure detector in which each node in the set is assigned a timer representing its liveness. Every time a message is received from a given in-neighbor, the corresponding timer value is reset. If a timeout occurs, the in-neighbor is probed for liveness. If the node responds to the probe, its timer is reset. Otherwise, the node is removed from the list of in-neighbors.

V. IMPLEMENTATION

We implemented our protocol in the form of a Java library.² The library is intended to be used in conjunction with any peer-to-peer overlay network (Figure 3).

The library provides applications with a service to measure the mixing time of the peer-to-peer network through a dedicated API, just like any other function of the peer-to-peer system. Internally, the library obtains local connectivity information from the peer-to-peer system and then uses the IP network directly for communication. In some cases, it might be preferable to delegate all communication to the peer-to-peer system, for example to exploit proxy mechanisms or already established connections into private networks, and the library can be easily modified to do that.

We designed the library to be portable to a variety of peer-to-peer systems. In particular, we modularized the interface to the underlying peer-to-peer system through an *Integration Layer* that wraps the peer-to-peer system and hides its peculiarities from the implementation of the protocol’s *Control Layer*. In essence, the Integration Layer is responsible for obtaining the local node ID and the set of out-neighbors, as well as for manipulating this set to remove failed neighbors. To support a new type of overlay network, one must only write the specific code to obtain this information from the new overlay, and present this information through the following simple Node interface.

²<http://www.inf.usi.ch/carzaniga/p2impulse/>

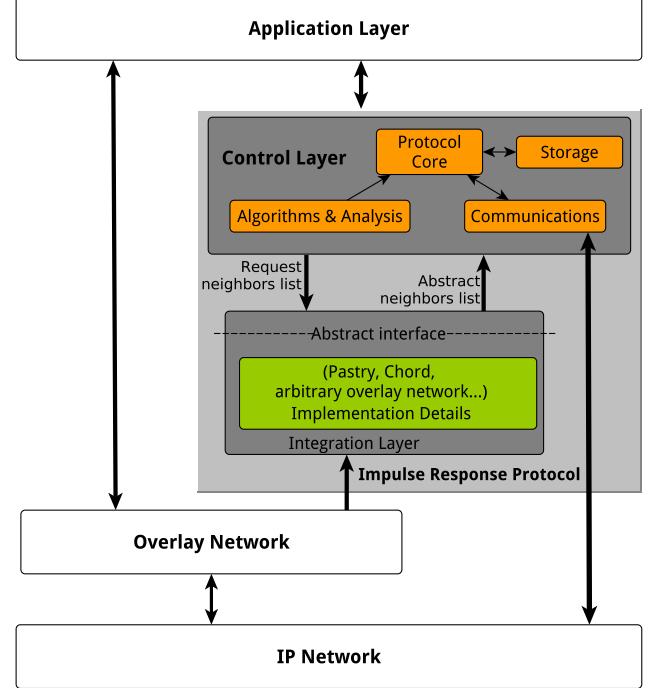


Fig. 3: Implementation architecture.

```

1 | public interface Node {
2 |     Id getLocalId();
3 |     Set<Neighbor> getOutNeighbors();
4 |     boolean removeOutNeighborNode(Id nodeId);
5 |

```

Our library currently provides implementations of the interface for both Pastry¹ and Chord.³

The Control Layer is responsible for implementing and coordinating the high-level protocol services, and consists of four components, as depicted in Figure 3. Most of the functionality is within the *Protocol Core* component, which is responsible for managing running sessions and their respective lines, as well as for managing their state. This state includes the phase they are in, when to advance to the next round, whether a line or a whole session should terminate, and the like. The *Protocol Core* also handles incoming protocol messages, deciding which impulse should be added to which line and whether a new message containing an impulse should be shared with the node’s out-neighbors. Yet another important role of this component is to run periodic maintenance tasks, such as discovering in-neighbors, maintaining the sets of in- and out-neighbors, and deciding whether to probe a remote node for liveness, as described in Section IV-C. These operations are performed by multiple threads running in parallel, each assigned to a different task.

The *Communications* component is responsible for the actual transmission and dispatching of protocol messages. The *Protocol Core* generates the required messages and then passes

³<http://sourceforge.net/projects/chordless/>

them to the Communications component, which transmits them asynchronously. Similarly, for incoming messages, the Communications component receives and dispatches messages to the Protocol Core. The design of the protocol messages (Figure 2) and of the Communications component allows for the multiplexing/demultiplexing of messages belonging to different lines or different rounds within the same session.

The Control Layer handles the state of, and computations associated with, the estimation service through the *Storage* and *Algorithms & Analysis* components, respectively. The Algorithms & Analysis component is responsible for performing all the algorithmic operations required by the protocol (Kung’s algorithm, eigenvalue computation, etc.). It is built on *jblas*, a linear-algebra library for Java that is itself based on the well-known *BLAS* and *LAPACK* libraries. The storage component uses a simple key/value database for network measurements.

Under our protocol, it is possible to have multiple sessions active simultaneously, initiated by different nodes originating from different service requests. However, the measurements provided by them would likely be quite similar and, therefore, the additional sessions would lead to increased network traffic without increased information.

In order to reduce unproductive traffic, the Control Layer makes use of a cache to store previous measurements. When a new service request is made, it checks whether a session is currently active. If no session is active, the cache is consulted for recent measurements and a new session created only if the stored values exceed some request-specific age threshold. On the other hand, if a session is currently active, then the entity issuing the service request can ask to wait for the output of that session or insist that a new session be initiated.

VI. EVALUATION

We now present the results of experiments to evaluate our mixing-time measurement service. The experiments are conducted using the Pastry overlay network deployed on the Emulab and PlanetLab network testbeds. We start by examining the case of an ideal network with no churn, where we validate the simulation results obtained by Carzaniga, Hall, and Papalini [2]. We then study the effects of churn on the estimated spectral values, and in particular the resiliency of our estimation for different protocol configurations. Finally, we demonstrate how the protocol could be used by a real application to provide some useful network-related measurements. In particular, we present an application based on the push-sum algorithm [4] operating on top of the Pastry overlay to compute the average number of files stored in the constituent nodes of the network.

A. Experimental Setup

To conduct our experimental study we developed an implementation of the generic Node interface for FreePastry, an open-source implementation of Pastry. FreePastry satisfies the requirements of the spectral estimation algorithm, since it builds and maintains a strongly connected and ergodic network, and also because, with minor changes to the source

code, it provides the local connectivity of each node required by the protocol (i.e., the list of out-neighbors). The topology of the Pastry overlay is also very similar to that of Chord, which is what Carzaniga, Hall, and Papalini used in their simulations [2].

We deployed the combined stack of our measurement service and FreePastry on two network testbeds, PlanetLab and Emulab. PlanetLab is a large research network spanning the world and running over the open Internet, which makes it a realistic testbed. On the other hand, this same realism may be limiting, in the sense that it is difficult, if not impossible, to conduct an experiment under controlled conditions. Therefore, we also deployed and tested the service on Emulab, which is a rack-based network testbed that allows the execution of repeatable experiments under controlled conditions. As it turns out, we obtain very similar and consistent results on both testbeds in all our experiments, so we report here only the specific results obtained on Emulab.

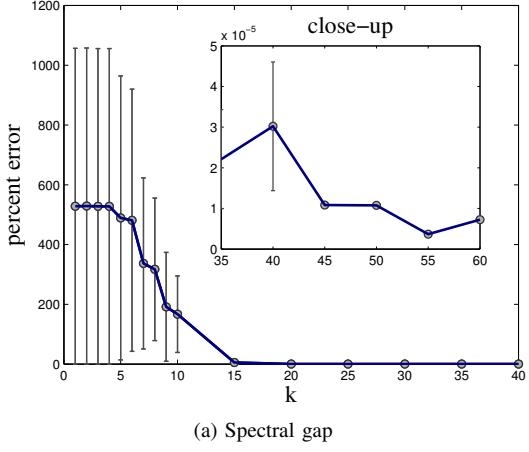
We conduct all the experiments on a peer-to-peer system of 100 nodes. In the case of Emulab we configure the network using a “big switch” topology with all links having a latency of 200ms. In the case of PlanetLab, we select nodes at random, obtaining the natural variability of levels of service (latency, bandwidth, and memory) offered by the testbed. Notice that we did not choose to select only highly performing nodes, which is an option of PlanetLab.

To replicate the results of Carzaniga, Hall, and Papalini, we measure both the spectral gap and the mixing time of the network. In particular, to compare the measurements provided by the service to the actual properties of the network, we also implemented an “all knowing” evaluator node to which every node in the network sends their local connectivity information. We then use the global properties computed (locally) by the evaluator node as the ground truth in the evaluation.

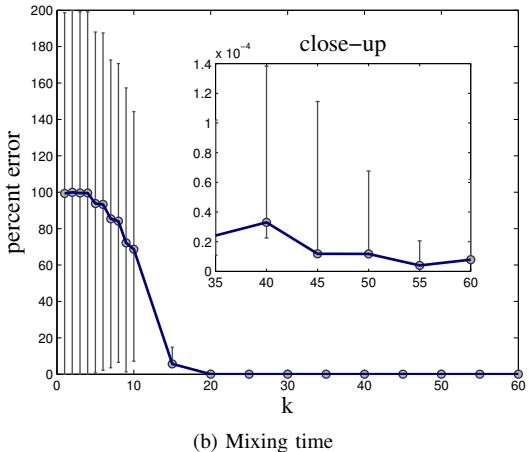
B. Validation of Simulation Results

In the first experiment we evaluate the accuracy of the estimation in the absence of churn by measuring the percent error in the mixing time and the spectral gap estimates. For this experiment we configure the protocol with sessions composed of a single line, since multiple lines would not change the results in any way without churn. We run multiple sessions with different numbers of rounds $k = 1 \dots 60$. Figure 4 displays the results. In particular, Figure 4a shows the spectral-gap error and Figure 4b the mixing-time error for the 10th, 50th, and 90th percentile of the values estimated by all the nodes in the network. The inner plot in the upper right corner of each graph focuses on the results for $k = 35$ up to $k = 60$.

We observe that, while the error can be quite high with only a few rounds, the accuracy of the estimate improves very quickly as the number of rounds increases. This is in accordance with the findings of Carzaniga, Hall, and Papalini. One noticeable difference is that the error drops faster in our experiments than in theirs. However, this too is to be expected, since we use a much smaller network (100 nodes



(a) Spectral gap



(b) Mixing time

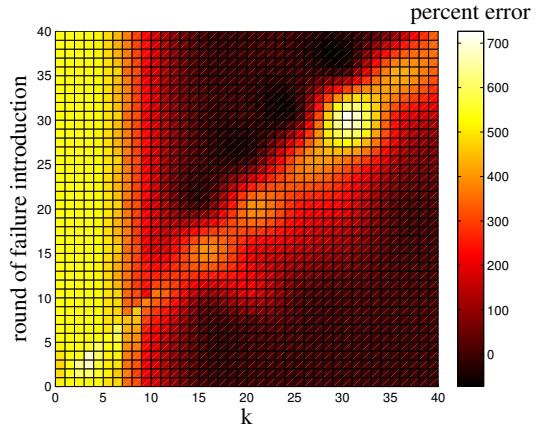
Fig. 4: Estimation error in the absence of churn.

in our real deployment versus 10000 in their simulation). We also observe that, although the error vanishes (see close-up graphs), the estimate is never completely accurate. We do not try to explain these small fluctuations, which might be related to numerical imprecision or other aspects of the estimation algorithm, including perhaps the variability in the initialization and in what is observable by each node in the network.

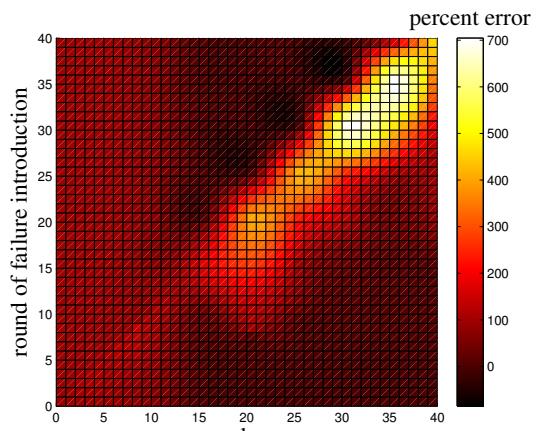
C. Resilience to Churn

Carzaniga, Hall, and Papalini measured the accuracy of the estimation in the presence of churn by introducing a single node failure in different rounds throughout the execution of their algorithm. As a result, they found that the errors in both the spectral gap and the mixing time could be as high as 200%, even for executions of the algorithm with several rounds. Moreover, the later the failure was introduced, the more the estimates diverged. As we did for the case of an ideal network, we validate these simulation results in our real deployments using sessions consisting of a single line (same case as the original estimation algorithm). Figures 5a and 5b present these results for the spectral gap and the mixing time, respectively. (These figures are best viewed in color.)

For this experiment we set the number of rounds to a range



(a) Spectral gap error



(b) Mixing time error

Fig. 5: Spectral gap and mixing time estimation error for a session with a single line.

of $k = 1 \dots 40$ and we plot a heat map in which the X-axis indicates the number of rounds, the Y-axis indicates the round in which the churn event occurs, and the heat color represents the median percent error of the estimate. The brighter the color of a point, the higher the percent error of the estimate.

We observe a number of features in both heat maps. First, we observe that the left sides of the maps are characterized by a lighter vertical band. This indicates that no matter where the churn event occurs, the accuracy is low for low values of k , which once again is to be expected, as shown in Section VI-B. Second, we observe a higher percent error close to the diagonal. This means that the estimates are inaccurate when a churn event occurs close to the end of the computation of the impulse response. Once more, this is in accordance with the original simulation results, with the only difference being that the percent error seems to be higher in the worst case for our implementation as compared to their simulated algorithm. We speculate that this inconsistency might be due to the slight difference between the FreePastry network used in our deployment and the Chord topology used in their

simulation. Third, we observe that churn events are more disruptive when they occur at the end of *longer* lines. This is also consistent with the simulation results and with our intuition that using multiple parallel runs can help to wash out erroneous estimates.

Having evaluated the accuracy of the estimation under churn with sessions consisting of just one line, we now turn to the more interesting case of *multiple* lines. Figures 6a and 6b present this case. We use the same parameters of Figure 5, except that now we use sessions of $l = 5$ lines, and again display the results using heat maps. However, notice two differences. First, the Y-axis extends to 72 rounds. This is because, while each line has the same length as in the experiments of Figure 5 (with $k = 1 \dots 40$), now the length of the whole *session* is $r = 72$ and, therefore, we study the impact of a churn event occurring at any point in the session. Second, the X-axis begins from $k = 5$ instead of $k = 1$. This is due to the way the partially overlapping lines are introduced, as explained in Section IV. With $l = 5$ lines and with $k < 5$ rounds in each line, there would be multiple lines completely overlapping and giving exactly the same results. In other words, the configurations in which $k < l$ amount to degenerate cases that we do not consider.

The results of Figure 6 demonstrate the resiliency provided by our enhanced approach, exhibiting a significant reduction in the percent error compared to the corresponding results of Figure 5, except for very low values of k . Notice also that, in order to make the error visible, we adjusted the “heat” scale. So, considering the mixing time, for example, the brightest color with $l = 1$ line per session (Figure 5b) indicates an error of 700%, while the same color in the case with $l = 5$ lines (Figure 6b) indicates an error of only 140%.

Once again we observe the bright vertical band on the left side of the map that indicates poor estimates for low values of k , as expected. We also notice that the error is most noticeable along the diagonal (which is lower, due to the different scales of the X and Y axes) similarly to the one-line experiments. However, the interpretation of this error is quite different. Here the diagonal corresponds to occurrences of churn events that affect most of the lines in a session, and therefore that lead to higher overall errors. Fortunately, however, this error vanishes for higher values of k . In essence, this is because the worst-case overall estimate corresponds to the estimate of the middle line, since all lines are affected, but earlier lines will be affected more and newer lines will be affected less. In other words, the overall estimate is never worse than the estimate of a line that is hit by a churn event near its $k/2$ round, and this estimate always improves with more rounds (i.e., higher k).

We conclude that using multiple, partially overlapping lines can be an effective approach to increasing the accuracy of the estimation in the presence of churn.

D. Demonstration

In this last part of the evaluation we demonstrate our service in a realistic setting by using it in support of a sampling application that computes the average number of files stored

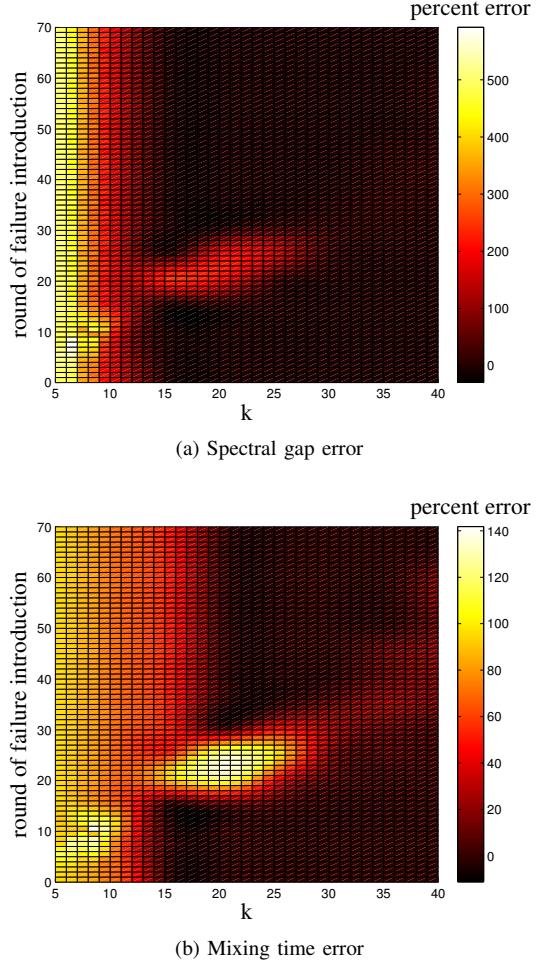


Fig. 6: Spectral gap and mixing time estimation error for a session with five lines.

in the nodes of a peer-to-peer network. We implemented this application on top of the FreePastry overlay, and deployed it over 100 nodes on the Emulab testbed.

Our sampling application implements the *push-sum* distributed gossip algorithm [4]. In essence, push-sum assumes that each node holds an estimate of the average, plus its weight. Initially the estimate is the number of files stored in the node and the weight is one. Then, for a number of rounds, each node chooses one of its neighbors uniformly at random, sends the information it currently holds, and then updates its current average and weight according to the information it receives from others. The idea is that after the algorithm terminates, the information of all the participating nodes will have been diffused across the network and, therefore, each node will have an accurate (and consistent) estimate of the average value.

One practical issue in implementing and configuring this algorithm is how to choose an appropriate number of rounds, since that has a fundamental effect on cost and precision. A small number of rounds will lead to inaccurate results, while a large number of rounds will result in an unnecessary increase of network traffic. One typical approach is to set

the number of rounds to a value approximate to the upper bound $O(\log n - \log \epsilon - \log \delta)$, where n is the number of nodes, $\epsilon < 1$ is the relative error in the approximation of the average, and $1 - \delta$ is the desired probability of obtaining such an approximation [4]. As it turns out, this heuristic approach usually yields a good estimate of the average. For our implementation we used a similar approach [8], but also takes into consideration the mixing time τ_{mix} , setting the number of rounds based on the alternative upper bound $O(\log n + \tau_{mix})$.

We implemented our application to use periodic estimates of the mixing time τ_{mix} provided by our measurement service. For the second parameter n (the size of the network) we exploit a particular property of Pastry, namely that the size of the routing table held by each node is $O(\log n)$. Therefore, we use the size of this table as a direct approximation for $\log n$. Concretely, we set the number of rounds for executing the push-sum algorithm to the size of the Pastry routing table plus the estimated τ_{mix} . In the particular setting we used, for 100 nodes, this results in 78 push-sum rounds.

To evaluate the effectiveness of our mixing-time measurement service, we collect the average values computed by push-sum throughout its execution, which we compare to the actual average (known to us, since we set up the experiment). Figure 7 shows the result. We plot the percent error of the target metric computed by push-sum (average number of files) as a function of the number of push-sum rounds. We plot the mean error, showing the minimum and maximum errors over all the values computed by the nodes in the network.

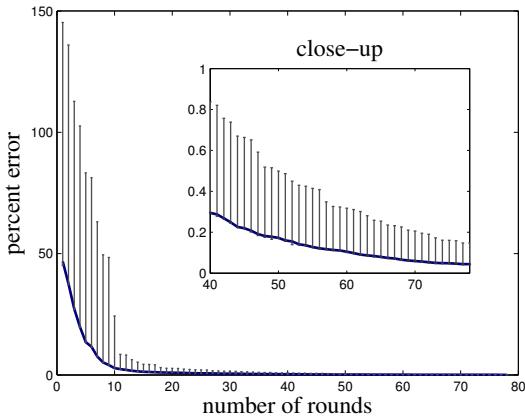


Fig. 7: Mean error in average number of files per node.

As we can see from the plot, while the average error is minimized quickly (at about 20 rounds), the maximum error requires more rounds to drop. This means that there are some nodes that converge later. However, in the final round, the average and the maximum error have a difference of less than 0.2%, meaning that almost all nodes converge to an accurate measurement.

We conclude that the mixing time provided by our measurement service can help to better tune push-sum to obtain the desired result.

VII. CONCLUSION

For some distributed algorithms, especially gossip algorithms, it is important to know whether or not they are running on a fast-mixing network. In particular, the mixing time often serves as a crucial performance parameter. For other algorithms, such as in the setting of security and privacy, the ability to uncover the mixing time of a network can itself serve as a key piece of valuable information.

Unfortunately, mixing time is a global property that cannot be immediately derived from the local information available to any single node in the network. On the other hand, the mixing time can be estimated with an efficient distributed algorithm. We extended and improved a recent and otherwise successful algorithm by making it resilient to network churn. We also designed and developed the first concrete implementation of the algorithm, providing it as a general, network-agnostic utility library. Our evaluation demonstrates that the design is robust, providing results with high accuracy.

One way to build upon our work would be to make the service resilient also to byzantine behavior. While it would be easy to discard forged or corrupt messages, it would be more interesting to defend against the faulty or malicious behavior of legitimate nodes. Ideally this would be an inherent stability property of the service. At a more practical level, it would be interesting to extend and engineer the implementation to support a wider range of applications and networks.

REFERENCES

- [1] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE/ACM Transactions on Networking*, 14(SI):2508–2530, June 2006.
- [2] A. Carzaniga, C. Hall, and M. Papalini. Fully decentralized estimation of some global properties of a network. In *Proceedings of IEEE INFOCOM*, pages 630–638, Orlando, Florida, Mar. 2012.
- [3] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003.
- [4] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Sciences*, pages 482–491, 2003.
- [5] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, Chicago, IL, USA, June 2004.
- [6] S. Kung. A new identification and model reduction algorithm via singular value decomposition. In *Proceedings of the 12th Asilomar Conference on Circuits, Systems and Computers*, Nov. 1978.
- [7] S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding P2P bots with structured graph analysis. In *Proceedings of the 19th USENIX Conference on Security*, Washington, DC, 2010.
- [8] T. Sauerwald. On mixing and edge expansion properties in randomized broadcasting. In *Algorithms and Computation*, pages 196–207. Springer, 2007.
- [9] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending against sybil attacks via social networks. In *Proceedings of ACM SIGCOMM*, pages 267–278, Pisa, Italy, Sept. 2006.