Gossip-based Peer Sampling

MÁRK JELASITY
University of Szeged and Hungarian Academy of Sciences, Hungary
SPYROS VOULGARIS
ETH Zurich, Switzerland
RACHID GUERRAOUI
EPFL, Lausanne, Switzerland
ANNE-MARIE KERMARREC
INRIA, Rennes, France
MAARTEN VAN STEEN
Vrije Universiteit, Amsterdam, The Netherlands

Gossip-based communication protocols are appealing in large-scale distributed applications such as information dissemination, aggregation, and overlay topology management. This paper factors out a fundamental mechanism at the heart of all these protocols: the *peer-sampling* service. In short, this service provides every node with peers to gossip with. We promote this service to the level of a first-class abstraction of a large-scale distributed system, similar to a name service being a first-class abstraction of a local-area system. We present a generic framework to implement a peer-sampling service in a decentralized manner by constructing and maintaining *dynamic unstructured* overlays through gossiping membership information itself. Our framework generalizes existing approaches and makes it easy to discover new ones. We use this framework to empirically explore and compare several implementations of the peer-sampling service. Through extensive simulation experiments we show that—although all protocols provide a good quality uniform random stream of peers to each node locally—traditional theoretical assumptions about the randomness of the unstructured overlays as a whole do not hold in any of the instances. We also show that different design decisions result in severe differences from the point of view of two crucial aspects: load balancing and fault tolerance. Our simulations are validated by means of a wide-area implementation.

Authors' addresses: M. Jelasity, USZ HAS Research Group on Artificial Intelligence, University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1., Hungary; email: jelasity@inf.u-szeged.hu; S. Voulgaris, Institute for Pervasive Computing, ETH Zentrum (IFW B47.2), CH-8092 Zurich, Switzerland; email: spyros@inf.ethz.ch; R. Guerraoui, LPD (Station 14), I&C, EPFL, CH-1015 Lausanne, Switzerland; email: Rachid.Guerraoui@epfl.ch; A.-M. Kermarrec, IRISA/INRIA-Rennes, Campus Universitaire de Beaulieu, F-35043 Rennes Cedex, France; email: Anne-Marie.Kermarrec@inria.fr; M. van Steen, Department of Computer Science, Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam, The Netherlands; email: steen@cs.vu.nl.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; C.4 [**Performance of Systems**]: Fault Tolerance

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: gossip-based protocols, epidemic protocols, peer sampling service

1. INTRODUCTION

Gossip-based protocols, also called epidemic protocols, are appealing in large-scale distributed applications. The popularity of these protocols stems from their ability to reliably pass information among a large set of interconnected nodes, even if the nodes regularly join and leave the system (either purposefully or on account of failures), and the underlying network suffers from broken or slow links.

In a gossip-based protocol, each node in the system periodically exchanges information with a subset of its peers. The choice of this subset is crucial to the wide dissemination of the gossip. Ideally, any given node should exchange information with peers that are selected following a uniform random sample of *all nodes* currently in the system [Demers et al. 1987; van Renesse et al. 1998; Birman et al. 1999; Karp et al. 2000; Sun and Sturman 2000; Kowalczyk and Vlassis 2005]. This assumption has led to rigorously establish many desirable features of gossip-based protocols like scalability, reliability, and efficiency (see, e.g., [Pittel 1987] in the case of information dissemination, or [Kempe et al. 2003; Jelasity et al. 2005] for aggregation).

In practice, enforcing this assumption would require to develop applications where each node may be assumed to know every other node in the system [Birman et al. 1999; Gupta et al. 2002; Kermarrec et al. 2003]. However, providing each node with a complete membership table from which a random sample can be drawn, is unrealistic in a large-scale dynamic system, for maintaining such tables in the presence of joining and leaving nodes (referred to as *churn*) incurs considerable synchronization costs. In particular, measurement studies on various peer-to-peer networks indicate that an individual node may often be connected in the order of only a few minutes to an hour (see, e.g. [Bhagwan et al. 2003; Saroiu et al. 2003; Sen and Wang 2004]).

Clearly, decentralized schemes to maintain membership information are crucial to the deployment of gossip-based protocols. This paper factors out the very abstraction of a *peer-sampling service* and presents a generic, yet simple, gossip-based framework to implement it.

The peer-sampling service is singled-out from the application using it and, abstractly speaking, the same service can be used in different settings: information dissemination [Demers et al. 1987; Eugster et al. 2004], aggregation [Kempe et al. 2003; Jelasity et al. 2005; Jelasity et al. 2004; Montresor et al. 2004], load balancing [Jelasity et al. 2004], and network management [Voulgaris and van Steen 2003; Voulgaris et al. 2005]. The service is promoted as a first class abstraction of a large-scale distributed system. In a sense, it plays the role of a naming service in a traditional LAN-oriented distributed system as it provides each node with other nodes to interact with.

The basic general principle underlying the framework we propose to implement the peer-sampling service, is itself based on a gossip paradigm. In short, every node (1) maintains a relatively small local membership table that provides a *partial view* on the complete set of nodes and (2) periodically refreshes the table using a gossiping procedure. The framework is generic and can be used to instantiate known [Eugster et al. 2003; Jelasity et al. 2003; Voulgaris et al. 2005] and novel gossip-based membership implementations. In fact, our framework captures many possible variants of gossip-based membership dissemination. These variants mainly differ in the way the membership table is updated at a given node after the exchange of tables in a gossip round. We use this framework to experimentally evaluate various implementations and identify key design parameters in practical settings. Our experimentation covers both extensive simulations and emulations on a wide-area cluster.

We consider many dimensions when identifying qualitative differences between the variants we examine. These dimensions include the randomness of selecting a peer as perceived by a single node, the accuracy of the current membership view, the distribution of the load incurred on each node, as well as the robustness in the presence of failures and churn.

Maybe not surprisingly, we show that communication should rather be bidirectional: it should follow the push-pull model. Adhering to a push-only or pull-only approach can easily lead to (irrecoverable) partitioning of the set of nodes. Another finding is that robustness against failing nodes or churn can be enhanced if old table entries are dropped when exchanging membership information.

However, as we shall also see, no single implementation outperforms the others along all dimensions. In this study we identify these tradeoffs when selecting an implementation of the peer-sampling service for a given application. For example, to achieve good load balancing, table entries should rather be *swapped* between two peers. However, this strategy is less robust against failures and churn than non-swapping ones.

The paper is organized as follows. Section 2 presents the interface and generic implementation of our peer-sampling service. Section 3 characterizes *local* randomness: that is, the randomness of the samples as seen by a fixed participating node. In Section 4 we analyze *global* randomness in a graph-theoretic framework. Robustness to failures and churn is discussed in Section 5. The simulations are validated through a wide-area experimentation described in Section 6. Sections 7, 8 and 9 present the discussion, related work and conclusions, respectively.

2. PEER-SAMPLING SERVICE

The peer-sampling service is implemented over a set of nodes (a group) wishing to execute one or more protocols that require random samples from the group. The task of the service is to provide a participating node with a random subset of peers from the group.

2.1 API

The API of the peer-sampling service simply consists of two methods: init and getPeer. It would be technically straightforward to provide a framework for a multipleapplication interface and architecture. For a better focus and simplicity of notations we assume, however, that there is only one application. The specification of these methods is as follows.

init(). Initializes the service on a given node if this has not been done before. The actual initialization procedure is implementation dependent.

getPeer(). Returns a peer address if the group contains more than one node. The returned address is a sample drawn from the group. Ideally, this sample should be an independent unbiased random sample. The exact characteristics of this sample (e.g., its randomness or correlation in time and with other peers) is affected by the implementation.

The focus of the research presented in this paper is to give accurate information about the behavior of the getPeer() method in the case of a class of gossip-based implementations. Applications requiring more than one peer simply invoke this method repeatedly.

Note that we do not define a stop method. In other words, graceful leaves are handled as crashes. The reason is to ease the burden on applications by delegating the responsibility of removing inactive nodes to the service layer.

2.2 Generic Protocol Description

We consider a set of nodes connected in a network. A node has an address that is needed for sending a message to that node. Each node maintains a membership table representing its (partial) knowledge of the global membership. Traditionally, if this knowledge is complete, the table is called the *global view* or simply the *view*. However, in our case each node knows only a limited subset of the system, so the table is consequently called a *partial view*. The partial view is a list of c *node descriptors*. Parameter c represents the size of the list and is the same for all nodes.

A node descriptor contains a network address (such as an IP address) and an *age* that represents the freshness of the given node descriptor. The partial view is a list data structure, and accordingly, the usual list operations are defined on it. Most importantly, this means that the order of elements in the view is not changed unless some specific method (for example, permute, which randomly reorders the list elements) explicitly changes it. The protocol also ensures that there is at most one descriptor for the same address in every view.

The purpose of the gossiping algorithm, executed periodically on each node and resulting in two peers exchanging their membership information, is to make sure that the partial views contain descriptors of a continuously changing random subset of the nodes and (in the presence of failure and joining and leaving nodes) to make sure the partial views reflect the dynamics of the system. We assume that each node executes the same protocol of which the skeleton is shown in Figure 1.

The protocol consists of two threads: an active (client) thread initiating communication with other nodes, and a passive (server) thread waiting for and answering these requests.

Although the protocol is not synchronous, it is often convenient to refer to *cycles* of the protocol. We define a cycle to be a time interval of T time units where T is the parameter of the protocol in Figure 1. During a cycle, each node initiates one view exchange.

```
do forever
  wait(T time units)
  $p \leftarrow$ view.selectPeer()
  if push then
    // 0 is the initial age
    buffer $\leftarrow$ ((MyAddress,0))
    view.permute()
    move oldest H items to end of view
    buffer.append(view.head(c/2-1))
    send buffer to $p$
  else // empty view to trigger response
    send (null) to $p$
  if pull then
    receive buffer$_p$ from $p$
    view.select(c,H,S,buffer$_p$)
  view.increaseAge()
```

(a) active thread

```
do forever
  receive buffer$_p$ from $p$
  if pull then
    // 0 is the initial age
    buffer $\leftarrow$ ((MyAddress,0))
    view.permute()
    move oldest H items to end of view
    buffer.append(view.head(c/2-1))
    send buffer to $p$
  view.select(c,H,S,buffer$_p$)
  view.increaseAge()
```

(b) passive thread

```
method view.select(c,H,S,buffer$_p$)
  view.append(buffer$_p$)
  view.removeDuplicates()
  view.removeOldItems( min(H,view.size-c) )
  view.removeHead( min(S,view.size-c) )
  view.removeAtRandom(view.size-c)
```

(c) method view.select(c,H,S,buffer$_p$)

Fig. 1. The skeleton of a gossip-based implementation of a peer sampling service.

We now describe the behavior of the active thread. The passive thread just mirrors the same steps. The active thread gets activated in each T time units exactly once. Three globally known system-wide parameters are used in this algorithm: parameters c, the size of the partial view of each node, H and S. For the sake of clarity, we leave the details of the meaning and impact of H and S until the end of this section.

(1) First, a peer node is selected to exchange membership information with. This selection is implemented by the method selectPeer that returns the address of a *live* node. This method is a parameter of the generic protocol. We discuss the implementations of selectPeer in Section 2.3.

(2) Subsequently, if the information has to be pushed (boolean parameter push is true), then a buffer is initialized with a fresh descriptor of the node running the thread. Then, $c/2-1$ elements are appended to the buffer. The implementa-

tion ensures that these elements are selected randomly from the view without replacement, ignoring the oldest H elements (as defined by the age stored in the descriptors). If there are not enough elements in the view, then the oldest H elements are also sampled to fill in any remaining slots. As a side-effect of permuting the view to select the $c/2-1$ random elements without replacement, the view will have exactly those elements as first items (i.e., in the list head) that are being sent in the buffer. This fact will play a key role in the interpretation of parameter S as we explain later. Parameter H is guaranteed to be less than or equal to $c/2$. The buffer created this way is sent to the selected peer.

(3) If a reply is expected (boolean parameter pull is true) then the received buffer is passed to method select(c,H,S,buffer), which creates the new view based on the listed parameters, and the current view, making sure the size of the new view does not decrease and is at most c.
Method select(c,H,S,buffer) creates the new view based on the listed parameters, and the current view as follows: After appending the received buffer to the view, it keeps only the freshest entry for each address, eliminating duplicate entries. After this operation, there is at most one descriptor for each address. At this point, the size of the view is guaranteed to be at least the original size, since in the original view each address was included also at most once. Subsequently, the method performs a number of removal steps to decrease the size of the view to c. The parameters of the removal methods are calculated in such a way that the view size never drops below c. First, the oldest items are removed, as defined by their age, and parameter H. The name H comes from *healing*, that is, this parameter defines how aggressive the protocol should be when it comes to removing links that potentially point to faulty nodes (dead links). Note that in this way self-healing is implemented without actually checking if a node is alive or not. If a node is not alive, then its descriptors will never get refreshed (and thus become old), and therefore sooner or later they will get removed. The larger H, the sooner older items will be removed from views.

(4) After removing the oldest items, the S first items are removed from the view. Recall that it is exactly these items that were sent to the peer previously. As a result, parameter S controls the priority that is given to the addresses received from the peer. If S is high, then the received items will have a higher probability to be included in the new view. Since the same algorithm is run on the receiver side, this mechanism in fact controls the number of items that are *swapped* between the two peers, hence the name S for the parameter. This parameter controls the diversity of the union of the two new views (on the passive and active side). If S is low then both parties will keep many of their exchanged elements, effectively increasing the similarity between the two respective views. As a result, more unique addresses will be removed from the system. In contrast, if S is high, then the number of unique addresses that are lost from both views is lower. The last step removes random items to reduce the size of the view back to c.

This framework captures the essential behavior of many existing gossip membership protocols (although exact matches often require small changes). As such, the framework serves two purposes: (1) we can use it to compare and evaluate a wide range of different gossip membership protocols by changing parameter values, and (2) it can serve as a unifying implementation for a large class of protocols. As a next step, we will explore the design space of our framework, forming the basis for an extensive protocol comparison.

2.3 Design Space

In this section we describe a set of specific instances of our generic protocol by specifying the values of the key parameters. These instances will be analyzed in the rest of the paper.

2.3.1 *Peer Selection.* As described before, peer selection is implemented by selectPeer() that returns the address of a *live* node as found in the caller's current view. In this study, we consider the following *peer selection* policies:

rand	Uniform randomly select an available node from the view
tail	Select the node with the *highest* age

Note that the third logical possibility of selecting the node with the *lowest* age is not included since this choice is not relevant. It is immediately clear from simply considering the protocol scheme that node descriptors with a low age refer to neighbors that have a view that is strongly correlated with the node's own view. More specifically, the node descriptor with the lowest age always refers exactly to the last neighbor the node communicated with. As a result, contacting this node offers little possibility to update the view with unknown entries, so the resulting overlay will be very static. Our preliminary experiments fully confirm this simple reasoning. Since the goal of peer sampling is to provide uncorrelated random peers continuously, it makes no sense to consider any policies with a bias towards low age, and thus protocols that follow such a policy.

2.3.2 *View propagation.* Once a peer has been chosen, the peers may exchange information in various ways. We consider the following two *view propagation* policies:

push	The node sends descriptors to the selected peer
pushpull	The node and selected peer exchange descriptors

Like in the case of the view selection policies, one logical possibility: the *pull* strategy, is omitted. It is easy to see that the pull strategy cannot possibly provide satisfactory service. The most important flaw of the pull strategy is that a node cannot inject information about itself, except only when explicitly asked by another node. This means that if a node loses all its incoming connections (which might happen spontaneously even without any failures, and which is rather common as we shall see) there is no possibility to reconnect to the network.

2.3.3 *View selection.* The parameters that determine how view selection is performed are H, the self-healing parameter, and S, the swap parameter. Let us first note some properties of these parameters. First, assuming that c is even, all values of H for which $H > c/2$ are equivalent to $H = c/2$, because the protocol never decreases the view size to under c. For the same reason, all values of S for which

$S > c/2 - H$ are equivalent to $S = c/2 - H$. Furthermore, the last, random removal step of the view selection algorithm is executed only if $S < c/2 - H$. Keeping these in mind, we have a "triangle" of protocols with H ranging from 0 to $c/2$, and with S ranging from 0 to $c/2 - H$. In our analysis we will look at this triangle at different resolutions, depending on the scenarios in question. As a minimum, we will consider the three vertices of the triangle defined as follows.

blind	$H = 0, S = 0$	Keep blindly a random subset
healer	$H = c/2$	Keep the freshest entries
swapper	$H = 0, S = c/2$	Minimize loss of information

We must note here that even in the case of swapper, only at most $c/2-1$ descriptors can be swapped, because the first element of the received buffer of length $c/2$ is always a fresh descriptor of the sender node. This fresh descriptor is always added to the view of the recipient node if $H + S = c/2$, that is, when no random elements are removed. This detail is very important as it is the only way fresh information can enter the system.

2.4 Implementation

We now describe a possible implementation of the peer-sampling service API based on the framework presented in Section 2.2. We assume that the service forms a layer between the application and the unstructured overlay network.

2.4.1 *Initialization.* Method init() will cause the service to register itself with the gossiping protocol instance that maintains the overlay network. From that point, the service will be notified by this instance whenever the actual view is updated.

2.4.2 *Sampling.* As an answer to the getPeer call, the service returns an element from the *current* view. To increase the randomness of the returned peers, the service makes a best effort not to return the same element twice during the period while the given element is in the view: this would introduce an obvious bias that would damage the quality of the service. To achieve this, the service maintains a queue of elements that are currently in the view but have not been returned yet. Method getPeer returns the first element from the queue and subsequently it removes this element from the queue. When the service receives a notification on a view update, it removes those elements from the queue that are no longer in the current view, and appends the new elements that were not included in the previous view. If the queue becomes empty, the service falls back on returning random samples from the current view. In this case the service can set a warning flag that can be read by applications to indicate that the quality of the returned samples is no longer reliable.

In the following sections, we analyze the behavior of our framework in order to gradually come to various optimal settings of the parameters. Anticipating our discussion in Section 7, we will show that there are some parameter values that never lead to good results (such as selecting a peer from a fresh node descriptor). However, we will also show that no single combination of parameter values is always best and that, instead, tradeoffs need to be made.

3. LOCAL RANDOMNESS

Ideally, a peer-sampling service should return a series of unbiased independent random samples from the current group of peers. The assumption of such randomness has indeed led to rigorously establish many desirable features of gossip-based protocols like scalability, reliability, and efficiency [Pittel 1987].

When evaluating the quality of a particular implementation of the service, one faces the methodological problem of characterizing randomness. In this section we consider a fixed node and analyze the series of samples generated at that particular node.

There are essentially two ways of capturing randomness. The first approach is based on the notion of Kolmogorov complexity [Li and Vitányi 1997]. Roughly speaking, this approach considers as random any series that cannot be compressed. Pseudo random number generators are automatically excluded by this definition, since any generator, along with a random seed, is a compressed representation of a series of any length. Sometimes it can be proven that a series *can* be compressed, but in the general case, the approach is not practical to test randomness due to the difficulty of proving that a series *cannot* be compressed.

The second, more practical approach assumes that a series is random if any statistic computed over the series matches the theoretical value of the same statistic under the assumption of randomness. The theoretical value is computed in the framework of probability theory. This approach is essentially empirical, because it can never be mathematically proven that a given series is random. In fact, good pseudo random number generators pass most of the randomness tests that belong to this category.

Following the statistical approach, we view the peer-sampling service (as seen by a fixed node) as a random number generator, and we apply the same traditional methodology that is used for testing random number generators. We test our implementations with the "diehard battery of randomness tests" [Marsaglia 1995], the *de facto* standard in the field.

3.1 Experimental Settings

We have experimented our protocols using the PeerSim simulator [PeerSim]. All the simulation results throughout the paper were obtained using this implementation.

The diehard test suite requires as input a considerable number of 32-bit integers: the most expensive test needs $6 \cdot 10^7$ of them. To be able to generate this input, we assume that all nodes in the network are numbered from 0 to N. Node N executes the peer-sampling service, obtaining one number between 0 and $N-1$ each time it calls the service, thereby generating a sequence of integers. If N is of the form $N = 2^n + 1$, then the bits of the generated numbers form an unbiased random bit stream, provided the peer-sampling service returns random samples.

Due to the enormous cost of producing a large number of samples, we restricted the set of implementations of the view construction procedure to the three extreme points: blind, healer and shuffler. Peer selection was fixed to be tail and pushpull was fixed as the communication model. Furthermore, the network size was fixed to be $2^{10} + 1 = 1025$, and the view size was $c = 20$. These settings allowed us to complete $2 \cdot 10^7$ cycles for all the three protocol implementations. In each case, node

Birthday Spacings	The k-bit random numbers are interpreted as "birthdays" in a "year" of 2^k days. We take m birthdays and list the spacings between the consecutive birthdays. The statistic is the number of values that occur more than once in that list.
Greatest Common Divisor	We run Euclid's algorithm on consecutive pairs of random integers. The number of steps Euclid's algorithm needs to find the greatest common divisor (GCD) of these consecutive integers in the random series, and the GCD itself are the statistics used to test randomness.
Permutation	Tests the frequencies of the $5! = 120$ possible orderings of consecutive integers in the random stream.
Binary Matrix Rank	Tests the rank of binary matrices built from consecutive integers, interpreted as bit vectors.
Monkey	A set of tests for verifying the frequency of the occurrences of "words" interpreting the random series as the output of a monkey typing on a typewriter. The random number series is interpreted as a bit stream. The "letters" that form the words are given by consecutive groups of bits (e.g., for 2 bits there are 4 letters, etc).
Count the 1-s	A set of tests for verifying the number of 1-s in the bit stream.
Parking Lot	Numbers define locations for "cars." We continuously "park cars" and test the number of successful and unsuccessful attempts to place a car at the next location defined by the random stream. An attempt is unsuccessful if the location is already occupied (the two cars would overlap).
Minimum Distance	Integers are mapped to two or three dimensional coordinates and the minimal distance among thousands of consecutive points is used as a statistic.
Squeeze	After mapping the random integers to the interval $[0, 1)$, we test how many consecutive values have to be multiplied to get a value smaller than a given threshold. This number is used as a statistic.
Overlapping Sums	The sum of 100 consecutive values is used as a statistic.
Runs Up and Down	The frequencies of the lengths of monotonously decreasing or increasing sequences are tested.
Craps	200,000 games of craps are played and the number of throws and wins are counted. The random integers are mapped to the integers $1, \ldots, 6$ to model the dice.

Table I. Summary of the basic idea behind the classes of tests in the diehard test suite for random number generators. In all cases tests are run with several parameter settings. For a complete description we refer to [Marsaglia 1995].

N generated four samples in each cycle, thereby generating four 10-bit numbers. Ignoring two bits out of these ten, we generated one 32-bit integer for each cycle.

Experiments convey the following facts. No matter which two bits are ignored, it does not affect the results, so we consider this as a noncritical decision. Note that we could have generated 40 bits per cycle as well. However, since many tests in the diehard suit do respect the 32-bit boundaries of the integers, we did not want to artificially diminish any potential periodic behavior in terms of the cycles.

3.2 Test Results

A detailed description of the tests in the diehard benchmark is out of the scope of the paper. In Table I we summarize the basic ideas behind each class of tests. In general, the three random number sequences pass all the tests, including the most difficult ones [Marsaglia and Tsang 2002], with one exception. Before discussing the one exception in more detail, note that for two tests we did not have enough 32-bit integers, yet we could still apply them. The first case is the permutation test, which is concerned with the frequencies of the possible orderings of 5-tuples of subsequent random numbers. The test requires $5 \cdot 10^7$ 32-bit integers. However, we applied the test using the original 10-bit integers returned by the sampling service, and the random sequences passed. The reason is that ordering is not sensitive to the actual range of the values, as long as the range is not extremely small. The second case is the so called "gorilla" test, which is a strong instance of the class of the monkey tests [Marsaglia and Tsang 2002]. It requires $6.7 \cdot 10^7$ 32-bit integers. In this case we concatenated the output of the three protocols and executed the test on this sequence, with a positive result. The intuitive reasoning behind this approach is that if any of the protocols produces a nonrandom pattern, then the entire sequence is supposed to fail the test, especially given that this test is claimed to be extremely difficult to pass.

Consider now the test that proved to be difficult to pass. This test was an instance of the class of binary matrix rank tests. In this instance, we take 6 consecutive 32-bit integers, and select the same (consecutive) 8 bits from each of the 6 integers forming a 6×8 binary matrix whose rank is determined. That rank can be from 0 to 6. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks smaller or equal to 4, and for ranks 5 and 6.

When the selected byte coincides with the byte contributed by one call to the peer-sampling service (bits 0-7, 8-15, etc), protocols blind and swapper fail the test. To better see why, consider the basic functioning of the rank test. In most of the cases, the rank of the matrix is 5 or 6. If it is 5, it typically means that the same 8-bit entry is copied twice into the matrix. Our implementation of the peer-sampling service explicitly ensures that the diversity of the returned elements is maximized in the short run (see Section 2.4). As a consequence, rank 6 occurs relatively more often than in the case of a true random sequence. Note that for many applications this property is actually an advantage. However, healer passes the test. The reason of this will become clearer in the remaining parts of the paper. As we will see, in the case of healer the view of a node changes faster and therefore the queue of the samples to be returned is frequently flushed, so the diversity-maximizing effect is less significant.

The picture changes if we consider only every 4th sample in the random sequence generated by the protocols. In that case, blind and swapper pass the test, but healer fails. In this case, the reason of the failure of healer is exactly the opposite: there are relatively too many repetitions in the sequence. Taking only every 8th sample, all protocols pass the test.

Finally, note that even in the case of "failures," the numeric deviation from random behavior is rather small. The expected occurrences of ranks of ≤ 4, 5, and 6 are 0.94%, 21.74%, and 77.31%, respectively. In the first type of failure, when there are too many occurrences of rank 6, a typical failed test gives percentages 0.88%, 21.36%, and 77.68%. When ranks are too small, a typical failure is, for example, 1.05%, 21.89%, and 77.06%.

3.3 Conclusions

The results of the randomness tests suggest that the stream of nodes returned by the peer-sampling service is close to uniform random for all the protocol instances examined. Given that some widely used pseudo-random number generators fail at least some of these tests, this is a highly encouraging result regarding the quality of the randomness provided by this class of sampling protocols.

Based on these experiments we cannot, however, conclude on global randomness of the resulting graphs. Local randomness, evaluated from a peer's point of view is important, however, in a complex large-scale distributed system, where the stream of random nodes returned by the nodes might have complicated correlations, merely looking at local behavior does not reveal some key characteristics such as load balancing (existence of bottlenecks) and fault tolerance. In Section 4 we present a detailed analysis of the global properties of our protocols.

4. GLOBAL RANDOMNESS

In Section 3 we have seen that from a local point of view all implementations produce good quality random samples. However, statistical tests for randomness and independence tend to hide important *structural* properties of the system *as a whole.* To capture these global correlations, in this section we switch to a graph theoretical framework. To translate the problem into a graph theoretical language, we consider the *communication topology* or *overlay topology* defined by the set of nodes and their views (recall that getPeer() returns samples from the view). In this framework the directed edges of the communication graph are defined as follows. If node a stores the descriptor of node b in its view then there is a directed edge (a, b) from a to b. In the language of graphs, the question is how similar this overlay topology is to a random graph in which the descriptors in each view represent a uniform independent random sample of the whole node set?

In this section we consider graph-theoretic properties of the overlay graphs. An important example of such properties is the *degree distribution.* The indegree of node i is defined as the number of nodes that have i in their views. The outdegree is constant and equal to the view size c for all nodes. Degree distribution has many significant effects. Most importantly, degree distribution determines whether there are hot spots and bottlenecks from the point of view of communication costs. In other words, *load balancing* is determined by the degree distribution. It also has a direct relationship with reliability to different patterns of node failures [Albert et al. 2000], and has an effect on the exact way epidemics are spread [Pastor-Satorras and Vespignani 2001]. Apart from the degree distribution we also analyze the clustering coefficient and average path length, as described and motivated in Section 4.2.

The main goal of this paper is to explore the different design choices in the protocol space described in Section 2.2. More specifically, we want to assess the impact of the peer selection, view selection, and view propagation parameters. Accordingly, we chose to fix the network size to $N = 10^4$ and the maximal view size to $c = 30$. The results presented in this section were obtained using the PeerSim simulation environment [PeerSim].

4.1 Properties of Degree Distribution

The first and most fundamental question is whether, for a particular protocol implementation, the communication graph has some stable properties, which it maintains during the execution of the protocol. In other words, we are interested in the *convergence behavior* of the protocols. We can expect several sorts of dynamics which include chaotic behavior, oscillations, and convergence. In case of convergence the resulting state may or may not depend on the initial configuration of the system. In the case of overlay networks we obviously prefer to have convergence towards a state that is independent of the initial configuration. This property is called *self-organization*. In our case it is essential that in a wide range of scenarios the protocol instances should automatically produce consistent and predictable behavior. Section 4.1.1 examines this question.

A related question is whether there is convergence and what kind of communication graph a protocol instance converges to. In particular, as mentioned earlier, we are interested in what sense overlay topologies deviate from certain random graph models. We discuss this issue in Section 4.1.2.

Finally, we are interested in looking at *local dynamic* properties along with *globally stable* degree distributions. That is, it is possible that while the overall degree distribution and its global properties such as maximum, variance, average, etc., do not change, the degree of the individual nodes does. This is preferable because in this case even if there are always bottlenecks in the network, the bottleneck will not be the same node all the time which greatly increases robustness and improves load balancing. Section 4.1.3 is concerned with these questions.

4.1.1 *Convergence.* We now present experimental results that illustrate the convergence properties of the protocols in three different bootstrapping scenarios:

Growing. In this scenario, the overlay network initially contains only one node. At the beginning of each cycle, 500 new nodes are added to the network until the maximal size is reached in cycle 20. The view of these nodes is initialized with only a single node descriptor, which belongs to the oldest, initial node. This scenario is the most pessimistic one for bootstrapping the overlays. It would be straightforward to improve it by using more contact nodes, which can come from a fixed list or which can be obtained using inexpensive local random walks on the existing overlay. However, in our discussion we intentionally avoid such optimizations to allow a better focus on the core protocols and their differences.

Lattice. In this scenario, the initial topology of the overlay is a ring lattice, a structured topology. We build the ring lattice as follows. The nodes are first connected into a ring in which each node has a descriptor in its view that belongs to its two neighbors in the ring. Subsequently, for each node, additional descriptors of the nearest nodes are added in the ring until the view is filled.

Random. In this scenario the initial topology is defined as a random graph, in which the views of the nodes were initialized by a uniform random sample of the peer nodes.

As we focus on the dynamic properties of the protocols, we did not wish to average out interesting patterns, so in all cases the result of a single run is shown in the plots. Nevertheless, we ran all the scenarios 100 times to gain data on the stability of the

protocol	partitioned runs	average number of clusters	average largest cluster
(rand,healer,push)	100%	22.28	9124.48
(rand,swapper,push)	0%	n.a.	n.a.
(rand,blind,push)	18%	2.06	9851.11
(tail,healer,push)	29%	2.17	9945.21
(tail,swapper,push)	97%	4.07	9808.04
(tail,blind,push)	10%	2.00	9936.20

Table II. Partitioning of the push protocols in the growing overlay scenario. Data corresponds to cycle 300. Cluster statistics are over the partitioned runs only.

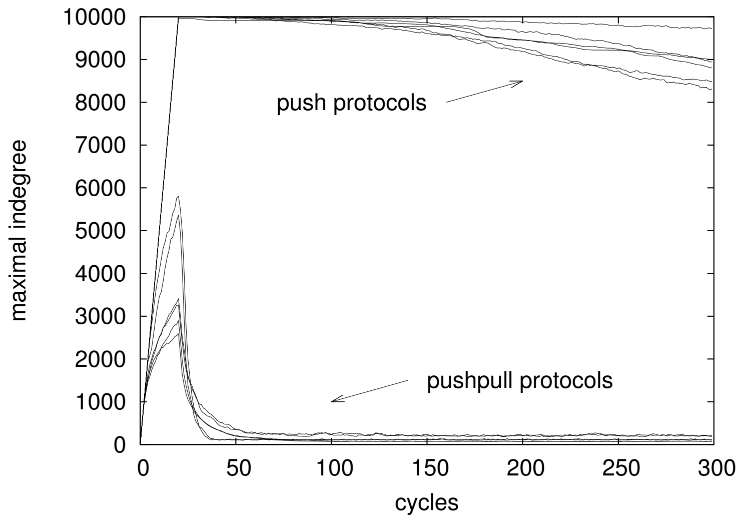

Fig. 2. Evolution of maximal indegree in the growing scenario (recall that growing stops in cycle 20). The runs of the following protocols are shown: peer selection is either rand or tail, view selection is blind, healer or swapper, and view propagation is push or pushpull.

protocols with respect to the connectivity of the overlay. Connectivity is a crucial feature, a minimal requirement for all applications. The results of these runs show that in all scenarios, every protocol under examination creates a connected overlay network in 100% of the runs (as observed in cycle 300). The only exceptions were detected during the growing overlay scenario. Table II shows the push protocols. With the pushpull scheme we have not observed any partitioning.

The push versions of the protocols perform very poorly in the growing scenario in general. Figure 2 illustrates the evolution of the maximal indegree. The maximal indegree belongs to the central contact node that is used to bootstrap the network. After growing is finished in cycle 20, the pushpull protocols almost instantly balance the degree distribution thereby removing the bottleneck. The push versions, however, get stuck in this unbalanced state.

This is not surprising, because when a new node joins the network and gets an initial contact node to start with, the only way it can get an updated view is if

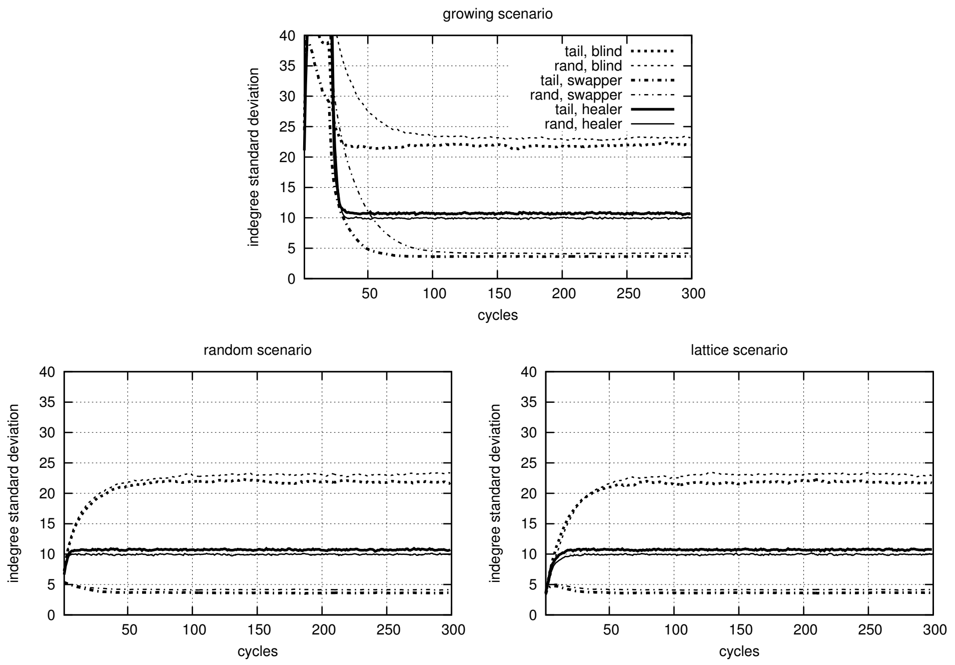

Fig. 3. Evolution of standard deviation of indegree in all scenarios of pushpull protocols.

some other node contacts it actively. This, however, is very unlikely. Because all new nodes have the same contact, the view at the contact node gets updated extremely frequently causing all the joining nodes to be quickly forgotten. A node has to push its own descriptor many times until some other node actually contacts it. This also means that if the network topology moves towards the shape of a star, then the push protocols have extreme difficulty balancing this degree-distribution state again towards a random one.

We conclude that this lack of adaptivity and robustness effectively renders push-only protocols useless. In the remaining part of the paper we therefore consider only the pushpull model.

Figure 3 illustrates the convergence of the pushpull protocols. Note that the average indegree is always the view size c. We can observe that in all scenarios the protocols quickly converge to the same value, even in the case of the growing scenario, in which the initial degree distribution is rather skewed. Other properties not directly related to degree distribution also show convergence, as discussed in Section 4.2.

4.1.2 *Static Properties.* In this section we examine the converged degree distributions generated by the different protocols. Figure 4 shows the converged standard deviation of the degree distribution. We observe that increasing both H and S results in a lower—and therefore more desirable—standard deviation. The reason is different for these two cases. With a large S, links to a node come to existence only

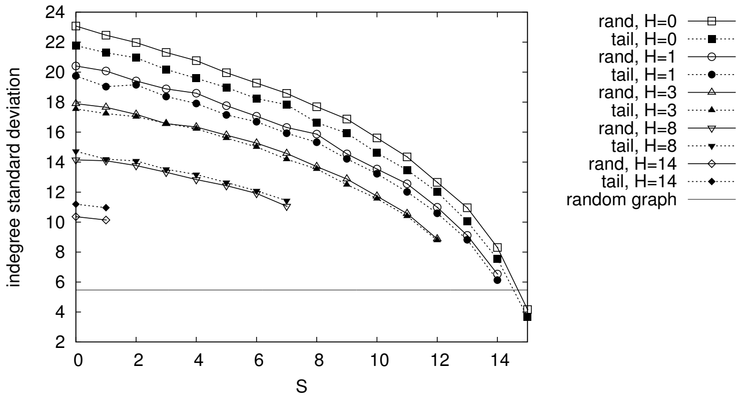

Fig. 4. Converged values of indegree standard deviation.

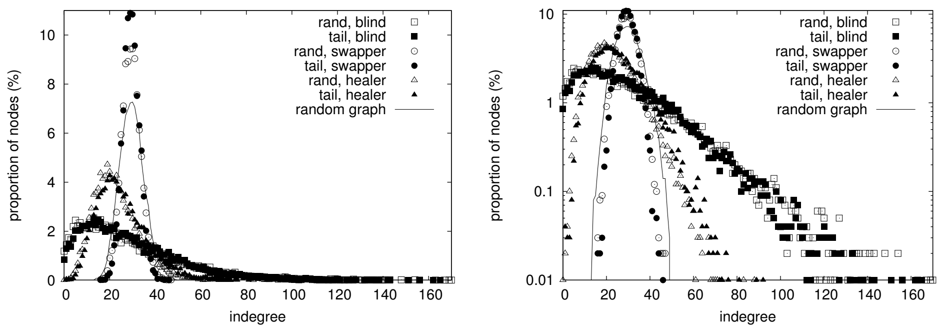

Fig. 5. Converged indegree distributions on linear and logarithmic scales.

in a very controlled way. Essentially, new incoming links to a node are created only when the node itself injects its own fresh node descriptor during communication. On the other hand, with a large H, the situation is the opposite. When a node injects a new descriptor about itself, this descriptor is (exponentially often) copied to other nodes for a few cycles. However, one or two cycles later all copies are removed because they are pushed out by new links (i.e., descriptors) injected in the meantime. So the effect that reduces variance is the short lifetime of the copies of a given link.

Figure 5 shows the entire degree distribution for the three vertices of the design space triangle. We observe that the distribution of swapper is narrower than that of the random graph, while blind has a rather heavy tail and also a large number of nodes with zero or very few nodes pointing to them, which is not desirable from the point of view of load balancing.

4.1.3 *Dynamic Properties.* Although the distribution itself does not change over time during the continuous execution of the protocols, the behavior of a single node still needs to be determined. More specifically, we are interested in whether a given

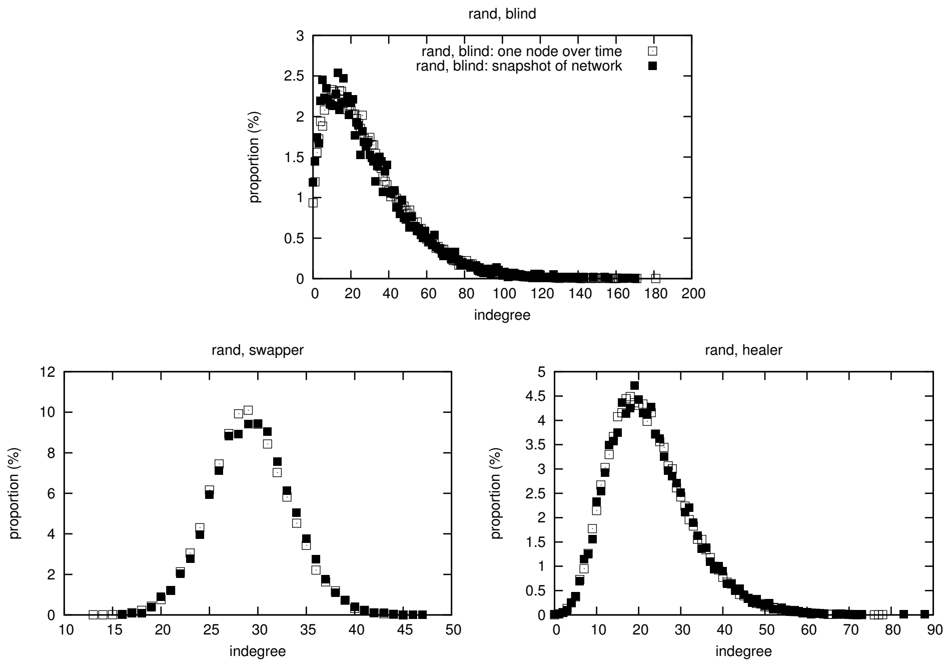

Fig. 6. Comparison of the converged indegree distribution over the network at a fixed time point and the indegree distribution of a fixed node during an interval of 50,000 cycles. The vertical axis represents the proportion of nodes and cycles, respectively.

fixed node has a variable indegree or whether the degree changes very slowly. The latter case would be undesirable because an unlucky node having above-average degree would continuously receive above-average traffic while others would receive less, which results in inefficient load balancing.

Figure 6 compares the degree distribution of a node over time, and the entire network at a fixed time point. The figure shows only the distribution for one node and only the random peer-selection protocols, but the same result holds for tail peer selection and for all the 100 other nodes we have observed. From the fact that these two distributions are very similar, we can conclude that all nodes take all possible values at some point in time, which indicates that the degree of a node is not static.

However, it is still interesting to characterize *how quickly* the degree changes, and whether this change is predictable or random. To this end, we present autocorrelation data of the degree time-series of fixed nodes in Figure 7. The band indicates a 99% confidence interval assuming the data is random. Only one node is shown, but all the 100 nodes we traced show very similar behavior. Let the series $d_1, \ldots d_K$ denote the indegree of a fixed node in consecutive cycles, and $\overline{d}$ the average of this series. The autocorrelation of the series $d_1, \ldots d_K$ for a given time lag k is defined

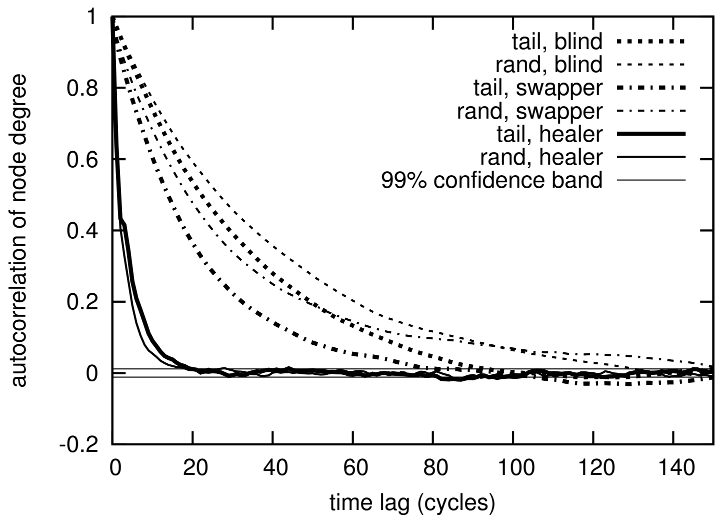

Fig. 7. Autocorrelation of indegree of a fixed node over 50,000 cycles. Confidence band corresponds to the randomness assumption: a random series produces correlations within this band with 99% probability.

as

$$r_k = \frac{\sum_{j=1}^{K-k}(d_j - \overline{d})(d_{j+k} - \overline{d})}{\sum_{j=1}^{K}(d_j - \overline{d})^2},$$

which expresses the correlation of pairs of degree values separated by k cycles.

We observe that in the case of healer it is impossible to make any prediction for a degree of a node 20 cycles later, knowing the current degree. However, for the rest of the protocols, the degree changes much slower, resulting in correlation in the distance of 80-100 cycles, which is not optimal from the point of view of load balancing.

4.2 Clustering and Path Lengths

Degree distribution is an important property of random graphs. However, there are other equally important characteristics of networks that are independent of degree distribution. In this section we consider the *average path length* and the *clustering coefficient* as two such characteristics. The clustering coefficient is defined over undirected graphs (see below). Therefore, we consider the undirected version of the overlay after removing the orientation of the edges.

4.2.1 *Average path length.* The shortest path length between node a and b is the minimal number of edges required to traverse in the graph in order to reach b from a. The average path length is the average of the shortest path lengths over all pairs of nodes in the graph. The motivation of looking at this property is that, in any information dissemination scenario, the shortest path length defines a lower bound on the time and costs of reaching a peer. For the sake of scalability a small

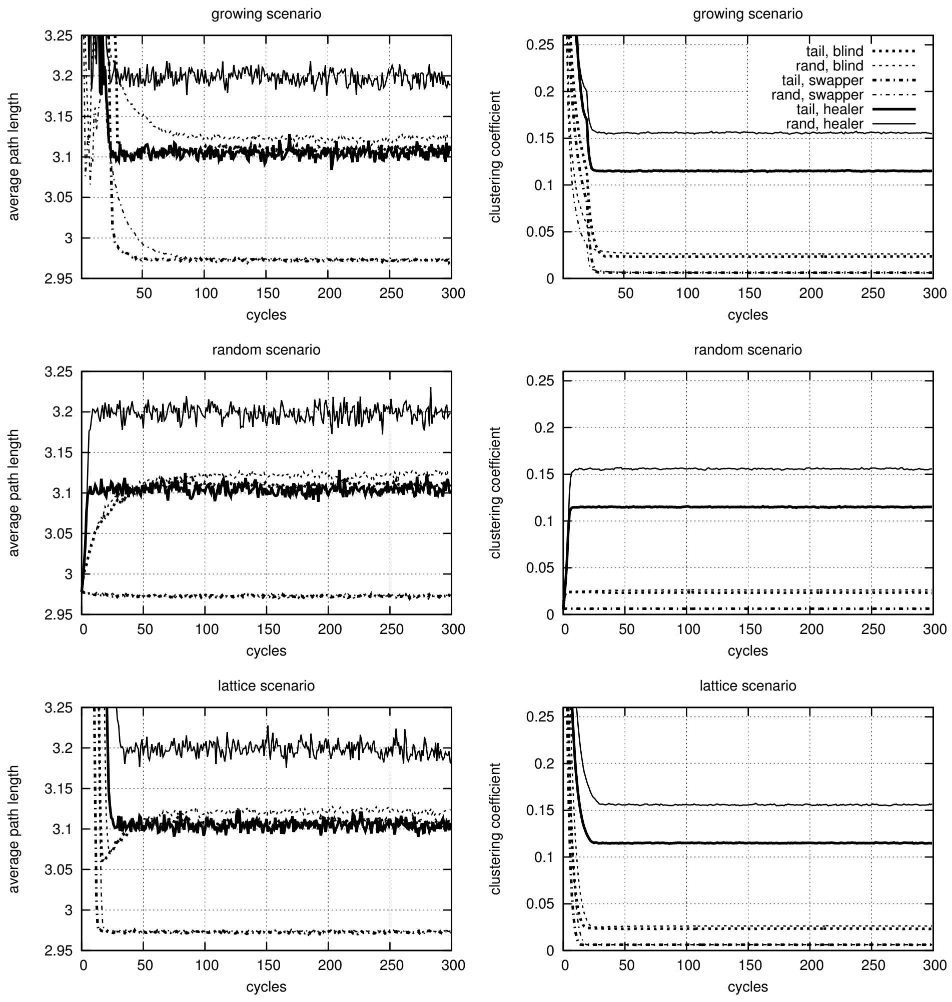

Fig. 8. Evolution of the average path length and the clustering coefficient in all scenarios.

average path length is essential. In Figure 8, especially in the growing and lattice scenarios, we verify that the path length converges rapidly. Figure 9 shows the converged values of the average path length for the design space triangle defined by H and S. We observe that all protocols result in a very low path length. Large S values are the closest to the random graph.

4.2.2 *Clustering coefficient.* The clustering coefficient of a node a is defined as the number of edges between the neighbors of a divided by the number of all possible edges between those neighbors. Intuitively, this coefficient indicates the extent to which the neighbors of a are also neighbors of each other. The clustering coefficient of a graph is the average of the clustering coefficients of its nodes, and always lies between 0 and 1. For a complete graph, it is 1, for a tree it is 0. The motivation for

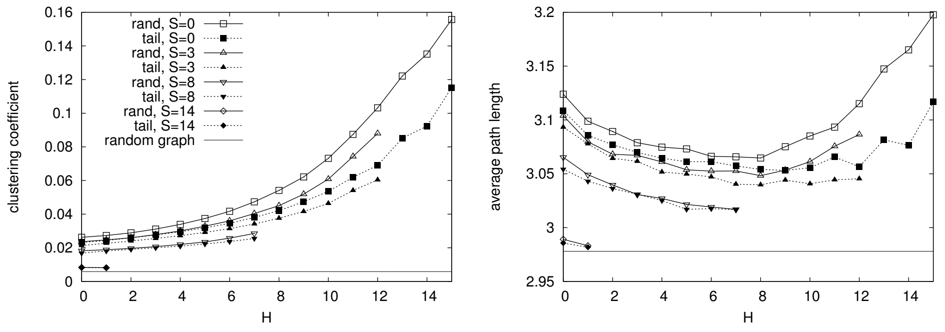

Fig. 9. Converged values of clustering coefficient and average path length.

analyzing this property is that a high clustering coefficient has potentially damaging effects on both information dissemination (by increasing the number of redundant messages) and also on the self-healing capacity by weakening the connection of a cluster to the rest of the graph thereby increasing the probability of partitioning. Furthermore, it provides an interesting possibility to draw parallels with research on complex networks where clustering is an important research topic (e.g., in social networks) [Watts and Strogatz 1998].

Like average path length, the clustering coefficient also converges (see Figure 8); Figure 9 shows the converged values. It is clear that clustering is controlled mainly by H. The largest values of H result in rather significant clustering, where the deviation from the random graph is large. The reason is that if H is large, then a large part of the views of any two communicating nodes will overlap right after communication, since both keep the same freshest entries. For the largest values of S, clustering is close to random. This is not surprising either because S controls exactly the diversity of views.

5. FAULT TOLERANCE

In large-scale, dynamic, wide-area distributed systems it is essential that a protocol is capable of maintaining an acceptable quality of service under a wide range of severe failure scenarios. In this section we present simulation results on two classes of such scenarios: *catastrophic failure*, where a significant portion of the system fails at the same time, and heavy *churn*, where nodes join and leave the system continuously.

5.1 Catastrophic Failure

As in the case of the degree distribution, the response of the protocols to a massive failure has a static and a dynamic aspect. In the static setting we are interested in the self-healing capacity of the converged overlays to a (potentially massive) node failure, as a function of the number of failing nodes. Removing a large number of nodes will inevitably cause some serious structural changes in the overlay even if it otherwise remains connected. In the dynamic case we would like to learn to what extent the protocols can repair the overlay after a severe damage.

The effect of a massive node failure on connectivity is shown in Figure 10. In

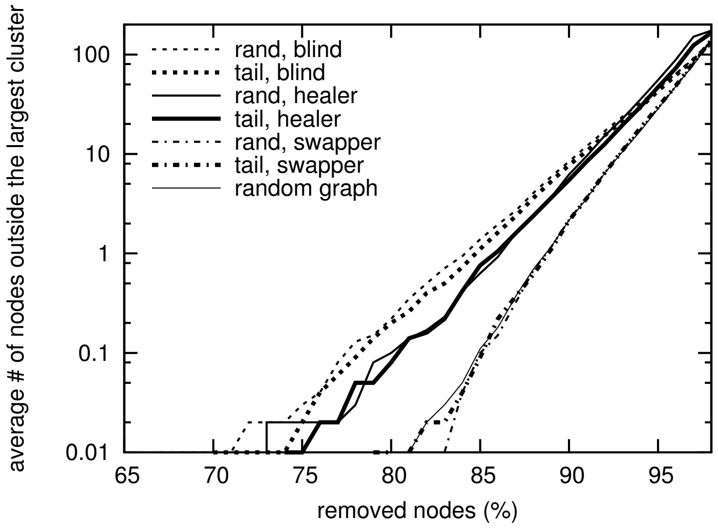

Fig. 10. The number of nodes that do not belong to the largest connected cluster. The average of 100 experiments is shown. The random graph almost completely overlaps with the swapper protocols.

this setting the overlay in cycle 300 of the random initialization scenario was used as converged topology. From this topology, random nodes were removed and the connectivity of the remaining nodes was analyzed. In all of the $100 \times 6 = 600$ experiments performed we did not observe partitioning until removing 67% of the nodes. The figure depicts the number of the nodes outside the largest connected cluster. We observe consistent partitioning behavior over all protocol instances (with swapper being particularly close to the random graph): even when partitioning occurs, most of the nodes form a single large connected cluster. Note that this phenomenon is well known for traditional random graphs [Newman 2002].

In the dynamic scenario we made 50% of the nodes fail in cycle 300 of the random initialization scenario and we then continued running the protocols on the damaged overlay. The damage is expressed by the fact that, on average, half of the view of each node consists of descriptors that belong to nodes that are no longer in the network. We call these descriptors dead links. Figure 11 shows how fast the protocols repair the overlay, that is, remove dead links from the views. Based on the static node failure experiment it was expected that the remaining 50% of the overlay is not partitioned and indeed, we did not observe partitioning with any of the protocols. Self-healing performance is fully controlled by the healing parameter H, with $H = 15$ resulting in fully repairing the network in as little as 5 cycles (not shown).

5.2 Churn

To examine the effect of churn, we define an artificial scenario in which a given proportion of the nodes crash and are subsequently replaced by new nodes in each cycle. This scenario is a *worst case* scenario because the new nodes are assumed to

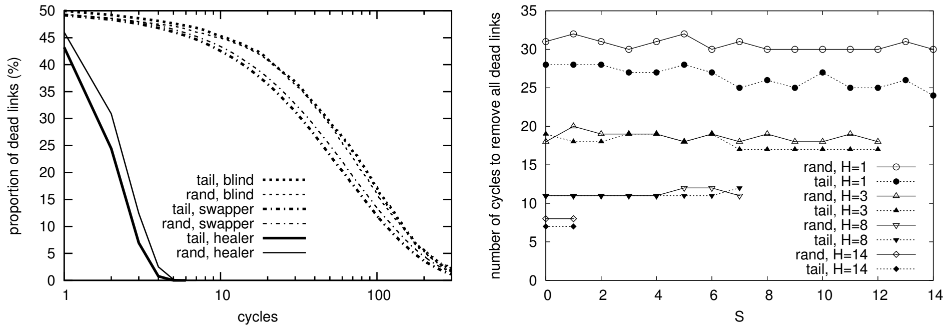

Fig. 11. Removing dead links following the failure of 50% of the nodes in cycle 300.

join the system for the first time, therefore they have no information whatsoever about the system (their view is initially empty) and the crashed nodes are assumed never to join the system again, so the links pointing to them will never become valid again. A more realistic trace-based scenario is also examined in Section 5.3 using the Gnutella trace described in [Saroiu et al. 2003].

We focus on two aspects: the churn rate, and the bootstrapping method. Churn rate defines the number of nodes that are replaced by new nodes in each cycle. We consider *realistic* churn rates (0.1% and 1%) and a *catastrophic* churn rate (30%). Since churn is defined in terms of cycles, in order to validate how realistic these settings are, we need to define the cycle length. With the very conservative setting of 10 seconds, which results in a very low load at each node, the trace described in [Saroiu et al. 2003] corresponds to 0.2% churn in each cycle. In this light, we consider 1% a comfortable upper bound of realistic churn, given also that the cycle length can easily be decreased as well to deal with even higher levels of churn.

We examine two bootstrapping methods. Both are rather unrealistic, but our goal here is not to suggest an optimal bootstrapping implementation, but to analyze our protocols under churn. The following two methods are suitable for this purpose because they represent two opposite ends of the design space:

Central. We assume that there exists a server that is known by every joining node, and that is stable: it is never removed due to churn or other failures. This server participates in the gossip membership protocol as an ordinary node. The new nodes use the server as their first contact. In other words, their view is initialized to contain the server.

Random. An oracle gives each new node a random live peer from the network as its first contact.

Realistic implementations could use a combination of these two approaches, where one or more servers serve random contact peers, using the peer-sampling service itself. Any such implementation can reasonably be expected to result in a behavior in between the two extremes described above.

Simulation experiments were run initializing the network with random links and subsequently running the protocols under the given amount of churn until the observed properties reached a stable level (300 cycles). The experimental results

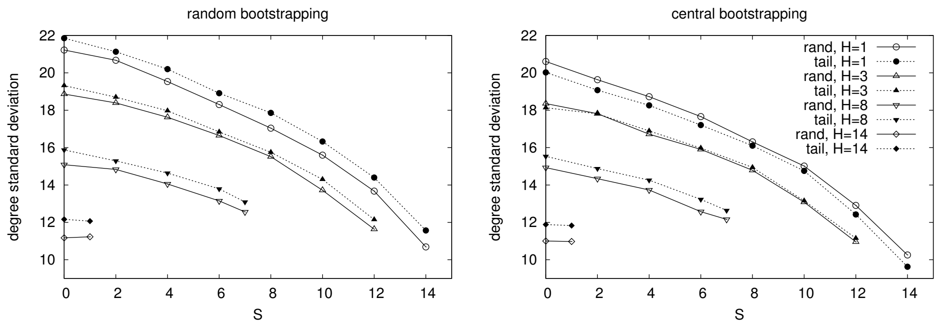

Fig. 12. Standard deviation of node degree with churn rate 1%. Node degree is defined over the *undirected* version of the subgraph of *live* nodes. The $H = 0$ case is not comparable to the shown cases; due to reduced self-healing, nodes have much fewer live neighbors (see Figure 13) which causes relatively low variance.

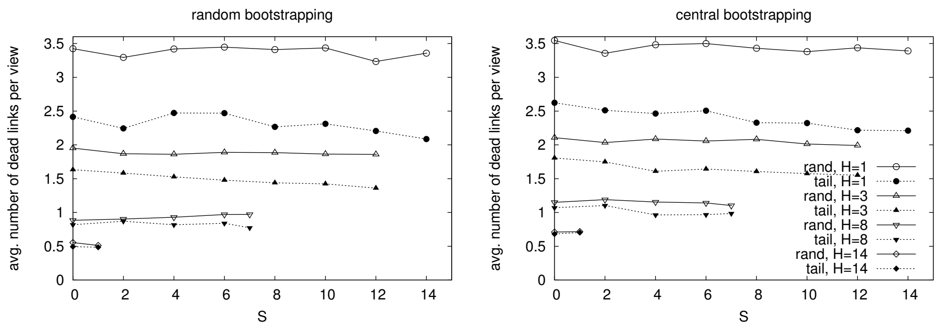

Fig. 13. Average number of dead links in a view with churn rate 1%. The $H = 0$ case is not shown; it results in more than 11 dead links per view on average, for all settings.

reveal that for realistic churn rates (0.1% and 1%) all the protocols are robust to the bootstrapping method and the properties of the overlay are very close to those without churn. Figure 12 illustrates this by showing the standard deviation of the node degrees in both scenarios, for the higher churn rate 1%. Observe the close correspondence with Figure 4. The clustering coefficient and average path length show the same robustness to bootstrapping, and the observed values are almost identical to the case without churn (not shown).

Let us now consider the damage churn causes in the networks. First of all, for all protocols and scenarios the networks remain connected, even for $H = 0$. Still, a (low) number of dead links remain in the overlay. Figure 13 shows the average number of dead links in the views, again, only for the higher churn rate (1%). It is clear that the extent of the damage is fully controlled by the healing parameter H. Furthermore, it is clear that the protocols are robust to the bootstrapping scenario also in this case. If $H \geq 1$ then the *maximal* (not average) number of dead links in any view for the different protocol instances ranges from $5 - 13$ in the case of churn rate 1% and from $2 - 5$ for churn rate 0.1%, where the lowest value belongs to the highest H. If $H = 0$ then the number of dead links radically increases: it is at least 11 on average, and the maximal number of dead links ranges from 20-25 for the different settings. That is, in the presence of churn, it is essential for any implementation to set at least $H = 1$. We have already seen this effect in Section 5.1 concerning self-healing performance.

Although the server participates in the overlay, the plots showing results under the central bootstrapping scenario were calculated ignoring the server, because its properties sharply differ from the rest of the network. In particular, it has a high indegree, because all new nodes will have a fresh link to the server, and that link will stay in the view of joining nodes for a few more cycles, possibly replicated in the meantime. Indeed, we observe that for 1% churn, 12% − 28% of the nodes have a link to the server at any time, depending on H and S. However, if we assume that the server can handle the traffic generated by joining nodes, a high indegree is noncritical. The expected number of incoming messages due to indegree d is d/c (where c is the view size), with a very low variance. This means that the generated traffic is of the same order of magnitude as the traffic generated by the joining nodes. We note again, however, that we do not consider this simplistic server-based solution a practical approach; we treat it only as a worst-case scenario to help us evaluate the protocols.

So far we have been discussing realistic churn rates. However, it is of academic interest to examine the behavior under *extremely* difficult scenarios, where the network suffers a catastrophic damage *in each cycle.* The catastrophic churn rate of 30% combines the effects of catastrophic failure (see Section 5.1) and churn.

Unlike with realistic churn rates, in this case the bootstrapping method has a strong effect on the performance of the protocols and therefore becomes the major design decision, although the parameters H and S still have a very strong effect as well. Consequently, we need to analyze the interaction of the gossip membership protocol and the bootstrapping method. In the case of the server-based solution, the overlay evolves into a ring-like structure, with a few shortcut links. The reason is that the view of the server is predominantly filled with entries of the newly joined nodes, since each time a new node contacts the server it also places a fresh entry about itself in the view of the server. These entries are served to the subsequently joining nodes, thus forming a linear structure. This ring-like structure is rather robust: it remains connected (even after removing the server) for all protocols with $H >= 8$. However, it has a slightly higher diameter than that of the random graph (approximately 20-30 hops). For healer the average number of dead links per view is still as low as 10 and 9 for random and tail peer selection, respectively.

The random scenario is rather different. In particular, we lose connectivity for all the protocols, however, for large values of H the largest connected cluster almost reaches the size of the network (see Figure 14). Besides, the structure of the overlay is also different. As Figure 14 shows, tail peer selection results in a slightly more unbalanced degree distribution (note that the low deviation for low values of H is due to the low number of live nodes). The reason is that—also considering that tail peer selection picks the oldest *live* node—the nodes that stay in the overlay for somewhat longer will receive more incoming traffic because (due to the very high number of dead links in each view) they tend to be the oldest live node in most

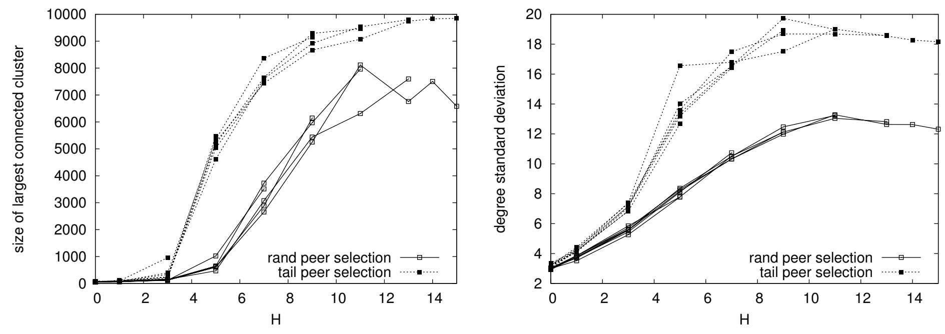

Fig. 14. Size of largest connected cluster and degree standard deviation under catastrophic churn rate (30%), with the random bootstrapping method. Individual curves belong to different values of S but the measures depend only on H, so we do not need to differentiate between them. Connectivity and node degree are defined over the *undirected* version of the subgraph of *live* nodes.

views they are in. For healer the average number of dead links per view is 11 and 9 for random and tail peer selection, respectively.

To summarize our findings: under realistic churn rates all the protocols perform very similarly to the case when there is no churn at all, independently of the bootstrapping method. Besides, some of the protocol instances, in particular, healer, can tolerate even catastrophic churn rates with a reasonable performance with both bootstrapping methods.

5.3 Trace-driven Churn Simulations

In Section 5.2 we analyzed our protocols under artificial churn scenarios. Here, we consider a realistic churn scenario using the, so called, *lifetime measurements* on Gnutella, carried out by Saroiu et al. [Saroiu et al. 2003]. These traces contain—among other information—the connection and disconnection times for a total of 17,125 nodes over a period of 60 hours. Throughout the trace, the number of connected nodes remains practically unchanged, in the order of 10^4 nodes.

We noticed a periodic pattern occurring every 404 seconds in the traces. In each 404-second interval, all connections and disconnections take place during the first 344 seconds, rendering the network static during the last 60 seconds. These recurring gaps would represent a positive bias for our churn simulations, as they periodically provide the overlay with some "breathing space" to process recent changes. However, these gaps are not realistic and are most probably an artifact of the logging mechanism. Therefore, we decided to eliminate them by linearly expanding each 344 second interval to cover the whole 404 seconds. Note that this transformation leaves the node uptimes practically unaltered.

We have taken the following two decisions with respect to the parameters in the experiments presented. First, peer selection is fixed to random. Section 5.2 showed that random is outperformed by tail peer selection in all cases. Therefore, random is a suitable choice for this section as the worst case peer selection policy. Second, the swap parameter, S, is fixed to 0. Section 5.2 showed that $S = 0$ results in the highest (therefore worst) degree deviation, while it does not affect the number of

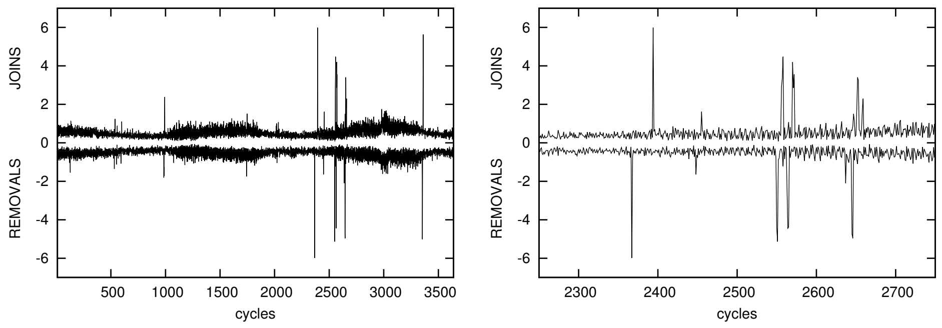

Fig. 15. Churn in the Saroiu traces. Full time span of 3600 one minute cycles and zoomed in to cycles 2250 to 2750.

dead links.

We apply two join methods: central and random, as defined in Section 5.2. The only difference is that a reconnecting node still remembers the links it previously had, some of which may be dead at reconnection time. This facilitates reconnection, but generally increases the total number of dead links.

The cycle length was chosen to be 1 minute. We anticipate that in reality the cycle length will be shorter, resulting in lower churn per cycle. The choice of a cycle length close to the upper end of realistic values is intentional, and is aimed at testing this specific gossip membership protocol under increased stress.

Figure 15 shows the node connections and disconnections as a percentage of the current network size. Connections are shown as positive points, whereas disconnections as negative. Although we ran the experiments for the whole trace, we focus on its most interesting part, namely cycles 2250 to 2750. Notice that at cycle 2367, around 450 nodes get disconnected at once and reconnect altogether 27 minutes later, at cycle 2394, probably due to a router failure. Similar temporary—but shorter—group disconnections are observed later on, around cycles 2450, 2550, and 2650, respectively.

Let us now examine the way the overlay is affected by those network changes. Figure 16 shows that the number of dead links is always kept at fairly small levels, especially when H is at least 1. As expected, the number of dead links peaks when there are massive node disconnections and gets back to normal quickly. However, it is not affected by the observed massive node reconnections, because these happen shortly after the respective disconnections, and the neighbors of the reconnected nodes are still alive.

Two observations regarding the effect of H can be made. First, higher values of H result in fewer dead links per view, validating the analysis in Section 5.2. Second, higher values of H trigger the *faster* elimination of dead links. The peaks caused by massive node disconnections are wider for low H values, and become sharper as H grows. In fact, these two observations are related to each other: in a persistently dynamic network, the converged average number of dead links depends on the rate at which the protocol disposes of them.

Figure 17 shows the evolution of the node degree deviation. It can be observed

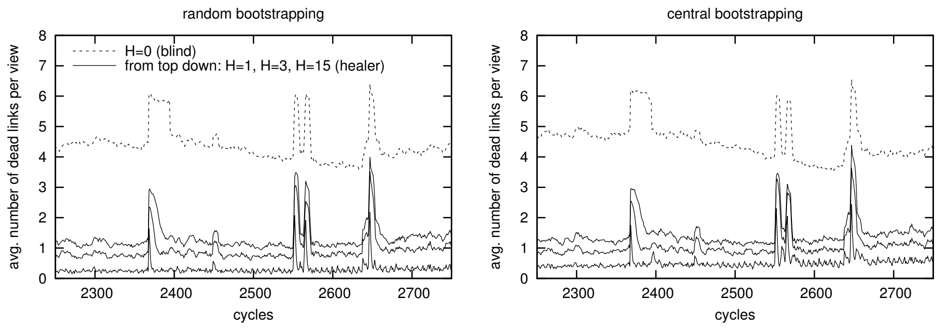

Fig. 16. Average number of dead links per view, based on the Saroiu Gnutella traces. All experiments use random peer selection and $S = 0$.

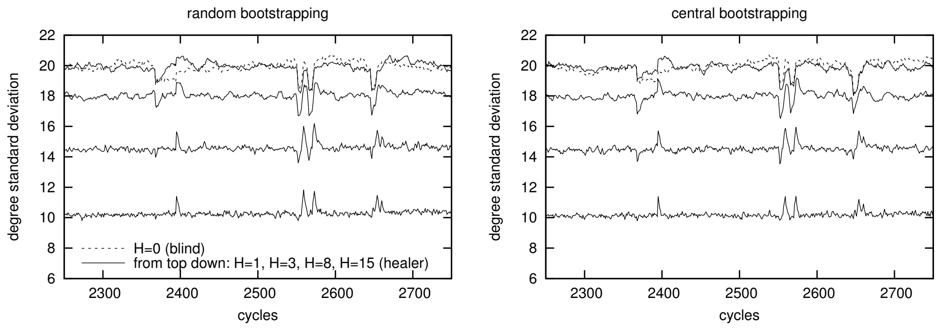

Fig. 17. Evolution of standard deviation of node degree based on the Saroiu Gnutella traces. All experiments use random peer selection and $S = 0$.

that for $H \geq 1$ the degree deviation under churn is very close to the corresponding converged values in a static network (see Figure 4). For $H = 0$ though, the higher number of pending dead links affects the degree distribution more. Note that both massive node disconnections *and* connections disturb the degree deviation, but in both cases a few cycles are sufficient to recover the original overlay properties.

To recap our analysis, we have shown that even with a pessimistic cycle length of 1 minute, all protocols for $H \geq 1$ perform very similarly to the case of a stable network, independently of the join method. Anomalies caused by massive node connections or disconnections are repaired quickly.

6. WIDE-AREA-NETWORK EMULATION

Distributed protocols often exhibit unexpected behavior when deployed in the real world that cannot always be captured by simulation. Typically, this is due to unexpected message loss, network and scheduling delays, as well as events taking place in unpredictable, arbitrary order. In order to validate the correctness of our simulation results, we implemented our gossip membership protocols and deployed them on a wide-area network.

We utilized the DAS-2 wide-area cluster as our testbed [DAS2]. The DAS-2 cluster consists of 200 dual-processor nodes spread across 5 sites in the Netherlands. A total of 50 nodes were used for our emulations, 10 from each site. Each node was running a Java Virtual Machine emulating 200 peers, giving a total of 10,000 peers. Peers were running in separate threads.

Although 200 peers were running on each physical machine, communication within a machine accounted for only 2% of the total communication. Local-area and wide-area traffic accounted for 18% and 80% of the total, respectively. Clearly, most messages are transferred through wide area connections. Note that the intra-cluster and inter-cluster round-trip delays on the DAS-2 are in the orders of 0.15 and 2.5 milliseconds, respectively. In all emulations, the cycle length was set to 5 seconds.

In order to validate our simulation results, we repeated the experiments presented in Figures 3 and 8 of Section 4, using our real implementation. A centralized coordinator was used to initialize the node views according to the bootstrapping scenarios presented in Section 4.1.1, namely *growing*, *lattice*, and *random*.

The first run of the emulations produced graphs practically indistinguishable from the corresponding simulation graphs. Acknowledging the low round-trip delay on the DAS-2, we ran the experiments again, this time inducing a 50 msec delay in each message delivery, accounting for a round-trip delay of 100 msec on top of the actual one. The results presented in this section are all based on these experiments.

Figure 18 shows the evolution of the indegree standard deviation, clustering coefficient, and average path length for all experiments, using the same scales as Figures 3 and 8 to facilitate comparison. The very close match between simulation-based and real-world experiments for all three nodes of the design space triangle allows us to claim that our simulations represent a valid approximation of real-world behavior.

The small differences of the converged values with respect to the simulations are due to the induced round-trip delay. In a realistic environment, view exchanges are not atomic: they can be intercepted by other view exchanges. For instance, a node having initiated a view exchange and waiting for the corresponding reply, may in the meantime receive a view exchange request by a third node. However, the view updates performed by the active and passive thread of a node are not commutative. The results presented correspond to an implementation where we simply ignored this problem: all requests are served immediately regardless of the state of the serving node. This solution is extremely simple from a design point of view but may lead to corrupted views.

As an alternative, we devised and implemented three approaches to avoid corrupted views. In the first approach, a node's passive thread *drops incoming requests* while its active thread is waiting for a reply. In the second one, the node *queues*—instead of dropping—incoming requests until the awaited reply comes. As a third approach, a node's passive thread serves all incoming requests, but its active thread *drops a reply* if an incoming request intervened.

Apart from the added complexity that these solutions impose on our design, their benefit turned out to be difficult or impossible to notice. Moreover, undesirable situations may arise in the case of the first two: dropping or delaying a request from a third node may cause that node to drop or delay, in turn, requests it receives itself.

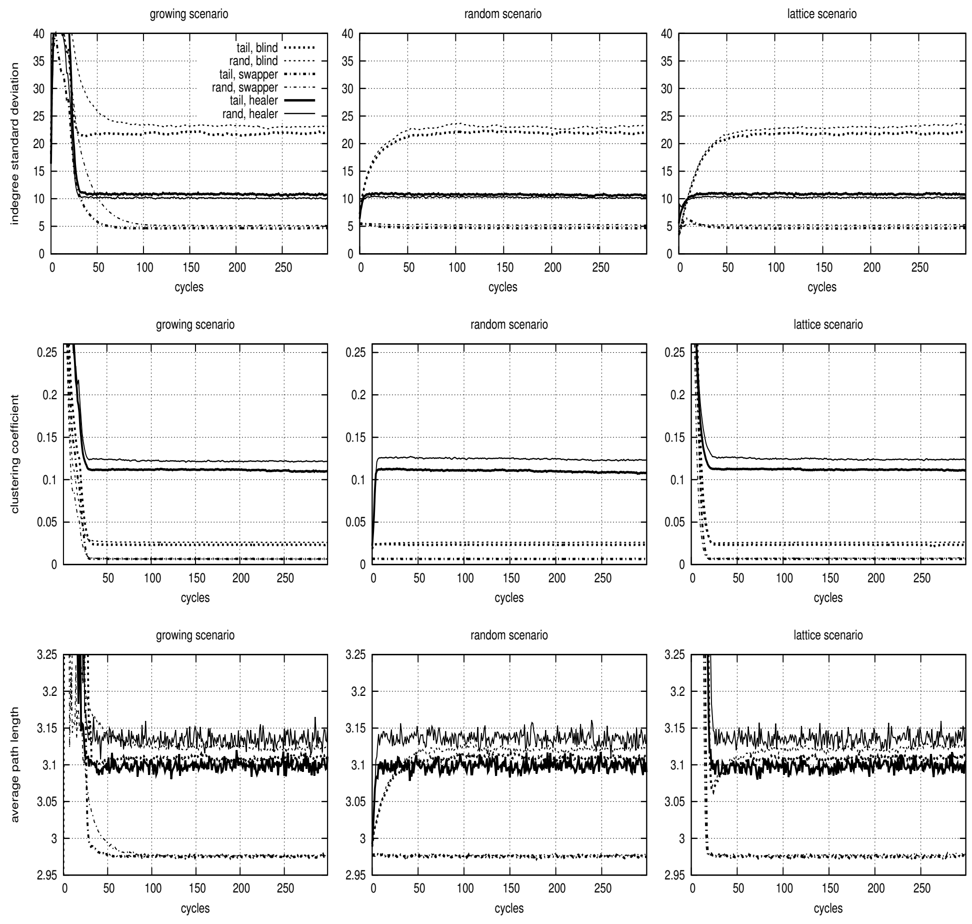

Fig. 18. Evolution of indegree standard deviation, clustering coefficient, and average path length in all scenarios for *real-world* experiments.

Chains of dependencies are formed this way, which can render parts of the network inactive for some periods. Given the questionable advantage these approaches can offer, and considering the design overhead they impose, we will not consider them further. Based on our experiments, the best strategy is simply ignoring the problem, which further underlines the exceptional robustness and simplicity of gossip-based design.

7. DISCUSSION

In this section we summarize and interpret the results presented so far. As stated in the introductory section, we were interested in determining the properties of various gossip membership protocols, in particular their randomness, load balancing and fault tolerance. In a sense, after we discussed in the last section why certain results were observed, we discuss here what the results imply.

7.1 Randomness

We have studied randomness from two points of view: local and global. Local randomness is based on the analogy between a pseudo random-number generator and the peer-sampling service as seen by a fixed node. We have seen that all protocols return a random sequence of peers at all nodes with a good approximation.

We have shown, however, that there are important correlations between the samples returned at different nodes, that is, the overlay graphs that the implementations are based upon are not random. Adopting a graph-theoretic approach, we have been able to identify important deviations from randomness that are different for the several instances of our framework.

In short, randomness is approached best by the view selection method swapper ($H = 0, S = c/2 = 15$), irrespective of the peer selection method. In general, increasing H increases the clustering coefficient. The average path length is close to the one of a random graph for all protocols we examined. Finally, with swapper the degree distribution has a smaller variance than that of the random graph. This property can often be considered "better than random" (e.g., from the point of view of load balancing).

Clearly, the randomness required by a given application depends on the very nature of that application. For example, the upper bound of the speed of reaching all nodes via flooding a network depends exclusively on the diameter of the network, while other aspects such as degree distribution or clustering coefficient are irrelevant for this specific question. Likewise, if the sampling service is used by a node to draw samples to calculate a local statistical estimate of some global property, such as network size or the availability of some resources, what is needed is that the local samples are uniformly distributed. However, it is *not* required that the samples are independent at different nodes, that is, we do not need global randomness at all; the unstructured overlay can have any degree distribution, diameter, clustering, etc.

7.1.1 *Load Balancing.* We consider the service to provide good load balancing if the nodes evenly share the cost of maintaining the service and the cost induced by the application of the service. Both are related to the degree distribution: if many nodes point to a certain node, this node will receive more sampling-service related gossip messages and most applications will induce more overhead on this node, resulting in poor load balancing. Since the unstructured overlays that implement the sampling service are dynamic, it is also important to note that nodes with a high indegree become a bottleneck only if they keep having a high indegree for a long time. In other words, a node is in fact allowed to have a high indegree temporarily, for a short time period.

We have seen that the blind view selection is inferior to the other alternatives. The degree distribution has a high variance (that is, there are nodes that have a large indegree) and on top of that, the degree distribution is relatively static, compared to the alternatives.

Clearly, the best choice to achieve good load balancing is the swapper view selection, which results in an even lower variance of indegree than in the uniform random graph. In general, the parameter S is strongly correlated with the variance of indegree: increasing S for a fixed H decreases the variance. The degree distribution is almost as static as in the case of healer, if $H = 0$. However, this is not a problem because the distribution has low variance.

Finally, healer also performs reasonably. Although the variance is somewhat higher than that of swapper, it is still much lower than blind. Besides, the degree distribution is highly dynamic, which means that the somewhat higher variance of the degree distribution does not result in bottlenecks because the indegree of the nodes change quickly. In general, increasing H for a fixed value of S also decreases the variance.

7.1.2 *Fault Tolerance.* We have studied both catastrophic and realistic scenarios. In the first category, catastrophic failure and catastrophic churn were analyzed. In these scenarios, the most important parameter turned out to be H: it is always best to set H as high as possible. One exception is the experiment with the removal of 50% of the nodes, where swapper performs slightly better. However, swapper is slow in removing dead links, so if failure can be expected, it is highly advisable to set $H \geq 1$.

In the case of realistic scenarios, such as the realistic (artificial) churn rates, and the trace-based simulations, we have seen that the damaging effect is minimal, and (as long as $H \geq 1$) the performance of the protocols is very similar to the case when there is no failure.

8. RELATED WORK

8.1 Gossip Membership Protocols

Most gossip protocols for implementing peer sampling are covered by our framework: we mentioned these in Section 2.2. One notable exception is [Allavena et al. 2005] that we address here in some more detail. The protocol is as follows. In each cycle, all nodes pull the full partial views from F randomly selected peers. In addition, they record the addresses of the peers initiating incoming pull requests during the given cycle. The old view is then discarded and a new view is generated from scratch. In the most practical version, the new view is generated by first adding the addresses of the incoming requests and subsequently filling the rest of the view with random samples from the union of the previously pulled F views without replacement.

Notice that there are two features that are incompatible with our framework: the application of $F \geq 1$ (in our case $F = 1$) and the asymmetry between push and pull, with pull having a bigger emphasis. Only one entry—the initiator peer's own entry—is pushed. It is common to allow for $F \geq 1$ also in other proposals (e.g., [Eugster et al. 2003]). In our framework, information exchange is symmetric, or fully asymmetric, without a finer tuning possibility.

To compare this protocol with our framework, we implemented it and ran simulations using the scenarios presented in this paper. The view size and network size were the same as in all simulations, and F was 1, 2, or 3. The main conclusions are summarized below. The protocol class presented in [Allavena et al. 2005] has some difficulty dealing with the scenarios when the initial network is not random (the growing and lattice initializations, see Section 4.1.1). For $F = 1$ we consistently observed partitioning in the lattice scenario (which was otherwise never observed in our framework). In the growing scenario—mostly for $F = 1$ but also for $F = 2$ and

$F = 3$—the protocols occasionally get stuck in a local attractor where there is a star subgraph: a node with a very high indegree, and a large number of nodes with zero indegree and 1 as outdegree. Apart from these issues, if we consider self-healing, load balancing and convergence properties, the protocols roughly behave as if they were instances in our framework using pushpull, with $0 \leq H \leq 1$ and $S = 0$, with increasing F tending towards $H = 1$. Since we have concluded that the "interesting" protocols in our space have either a high H or a high S value, based on the empirical evidence accumulated so far there is no urgent need to extend our framework to allow for $F > 1$ or asymmetric information exchange. However, studying these design choices in more detail is an interesting topic for future research.

In the following we summarize a number of other fields that are related to the research presented in this paper.

8.2 Complex Networks

The assumption of uniform randomness has only fairly recently become subject to discussion when considering large complex networks such as the hyperlinked structure of the WWW, or the complex topology of the Internet. Like social and biological networks, the structures of the WWW and the Internet both follow the quite unbalanced power-law degree distribution, which deviates strongly from that of traditional random graphs. These new insights pose several interesting theoretical and practical problems [Barabási 2002]. Several dynamic complex networks have also been studied and models have been suggested for explaining phenomena related to what we have described in the present paper [Dorogovtsev and Mendes 2002]. This related work suggests an interesting line of future theoretical research seeking to explain our experimental results in a rigorous manner.

8.3 Unstructured Overlays

There are a number of protocols that are not gossip-based but that are potentially useful for implementing peer sampling. An example is the Scamp protocol [Ganesh et al. 2003]. While this protocol is reactive and so less dynamic, an explicit attempt is made towards the construction of a (static) random graph topology. Randomness has been evaluated in the context of information dissemination, and it appears that reliability properties come close to what one would see in random graphs. Some other protocols have also been proposed to achieve randomness [Law and Siu 2003; Pandurangan et al. 2003], although not having the specific requirements of the peer-sampling service in mind. Finally, random walks on arbitrary (hence, also unstructured) networks offer a powerful tool to obtain random samples, where even the sampling distribution can be adjusted [Zhong et al. 2005]. These protocols, however, have a significantly higher overhead if many samples are required. This overhead and the convergence time also depend on the structure of the overlay network the random walk operates on.

8.4 Structured Overlays

In a sense, structured overlays have also been considered as a basic middleware service to applications [Dabek et al. 2003]. However, a structured overlay [Rowstron and Druschel 2001; Ratnasamy et al. 2001; Stoica et al. 2001] is by definition not dynamic. Hence utilizing it for implementing the peer-sampling service requires additional techniques such as random walks [Zhong et al. 2005; King and Saia 2004]. Another example of this approach is a method assuming a tree overlay [Kostić et al. 2003]. It is unclear whether a competitive implementation can be given considering also the cost of maintaining the respective overlay structure.

Another issue in common with our own work is that graph-theoretic approaches have been developed for further analysis [Loguinov et al. 2003]. Astrolabe [van Renesse et al. 2003] also needs to be mentioned as a hierarchical (and therefore structured) overlay, which, although applying (nonuniform) gossip to increase robustness and to achieve self-healing properties, does not even attempt to implement or apply a uniform peer-sampling service. It was designed to support hierarchical information aggregation and dissemination.

9. CONCLUDING REMARKS

Gossip protocols have recently generated a lot of interest in the research community. The overlays that result from these protocols are highly resilient to failures and high churn rates. The underlying paradigm is clearly appealing to build large-scale distributed applications

The contribution of this paper is to factor out the abstraction implemented by the membership mechanism underlying gossip protocols: the peer-sampling service. The service provides every peer with (local) knowledge of the rest of system, which is key to have the system converge as a whole towards global properties using only local information.

We described a framework to implement a reliable and efficient peer-sampling service. The framework itself is based on gossiping. This framework is generic enough to be instantiated with most current gossip membership protocols [Eugster et al. 2003; Jelasity et al. 2003; Stavrou et al. 2004; Voulgaris et al. 2005]. We used this framework to empirically compare the range of protocols through simulations based on synthetic and realistic traces as well as implementations. We point out the very fact that these protocols ensure local randomness from each peer's point of view. We also observed that as far as the global properties are concerned, the average path length is close to the one in random graphs and that clustering properties are controlled by (and grow with) the parameter H. With respect to fault tolerance, we observe a high resilience to high churn rate and particularly good self-healing properties, again mostly controlled by the parameter H. In addition, these properties mostly remain independent of the bootstrapping approach chosen.

In general, when designing gossip membership protocols that aim at randomness, following a push-only or pull-only approach is not a good choice. Instead, only the combination results in desirable properties. Likewise, it makes sense to build in robustness by purposefully removing old links when exchanging views with a peer. This situation corresponds in our framework to a choice for $H > 0$.

Regarding other parameter settings, it is much more difficult to come to general conclusions. As it turns out, tradeoffs between, for example, load balancing and fault tolerance will need to be made. When focusing on swapping links with a selected peer, the price to pay is lower robustness against node failures and churn. On the other hand, making a protocol extremely robust will lead to skewed indegree distributions, affecting load balancing.

To conclude, we demonstrated in this extensive study that gossip membership protocols can be tuned to both support high churn rates and provide graph-theoretic properties (both local and global) close to those of random graphs so as to support a wide range of applications. A complementary research direction would be to explore this spectrum theoretically.

REFERENCES

Albert, R., Jeong, H., and Barabási, A.-L. 2000. Error and attack tolerance of complex networks. *Nature 406*, 378–382.

Allavena, A., Demers, A., and Hopcroft, J. E. 2005. Correctness of a gossip based membership protocol. In *Proceedings of the 24th annual ACM symposium on principles of distributed computing (PODC'05)*. ACM Press, Las Vegas, Nevada, USA.

Barabási, A.-L. 2002. *Linked: the new science of networks*. Perseus, Cambridge, Mass.

Bhagwan, R., Savage, S., and Voelker, G. 2003. Understanding Availability. In *2nd International Workshop on Peer-to-Peer Systems*. Lecture Notes on Computer Science. Springer-Verlag, Berlin.

Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y. 1999. Bimodal multicast. *ACM Transactions on Computer Systems 17,* 2 (May), 41–88.

Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. 2003. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*. Berkeley, CA, USA.

DAS2. The distributed ASCI supercomputer 2 (DAS-2). `http://www.cs.vu.nl/das2/`.

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. ACM Press, Vancouver, British Columbia, Canada, 1–12.

Dorogovtsev, S. N. and Mendes, J. F. F. 2002. Evolution of networks. *Advances in Physics 51,* 1079–1187.

Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kermarrec, A.-M., and Kouznetsov, P. 2003. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems 21,* 4, 341–374.

Eugster, P. T., Guerraoui, R., Kermarrec, A.-M., and Massoulié, L. 2004. Epidemic information dissemination in distributed systems. *IEEE Computer 37,* 5 (May), 60–67.

Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. 2003. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers 52,* 2 (February).

Gupta, I., Birman, K. P., and van Renesse, R. 2002. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International 18,* 3, 165–184.

Jelasity, M., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. 2004. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *Middleware 2004*, H.-A. Jacobsen, Ed. Lecture Notes in Computer Science, vol. 3231. Springer-Verlag, 79–98.

Jelasity, M., Kowalczyk, W., and van Steen, M. 2003. Newscast computing. Tech. Rep. IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands. November.

Jelasity, M., Kowalczyk, W., and van Steen, M. 2004. An approach to massively distributed aggregate computing on peer-to-peer networks. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*. IEEE Computer Society, A Coruna, Spain, 200–207.

Jelasity, M., Montresor, A., and Babaoglu, O. 2004. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organising Systems*, G. Di Marzo Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, Eds. Lecture Notes in Artificial Intelligence, vol. 2977. Springer, 265–282. invited paper.

JELASITY, M., MONTRESOR, A., AND BABAOGLU, O. 2005. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems 23,* 2 (May).

KARP, R. M., SCHINDELHAUER, C., SHENKER, S., AND VÖCKING, B. 2000. Randomized Rumor Spreading. In *14th Symposium on the Foundations of Computer Science.* IEEE, IEEE Computer Society Press, Los Alamitos, CA., 565–574.

KEMPE, D., DOBRA, A., AND GEHRKE, J. 2003. Gossip-based computation of aggregate information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03).* IEEE Computer Society, 482–491.

KERMARREC, A.-M., MASSOULIÉ, L., AND GANESH, A. J. 2003. Probablistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems 14,* 3 (March).

KING, V. AND SAIA, J. 2004. Choosing a random peer. In *Proceedings of the 23rd annual ACM symposium on principles of distributed computing (PODC'04).* ACM Press, 125–130.

KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., BHIRUD, A., AND VAHDAT, A. 2003. Using random subsets to build scalable network services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS 2003).*

KOWALCZYK, W. AND VLASSIS, N. 2005. Newscast EM. In *17th Advances in Neural Information Processing Systems (NIPS)*, L. K. Saul, Y. Weiss, and L. Bottou, Eds. MIT Press, Cambridge, MA, 713–720.

LAW, C. AND SIU, K.-Y. 2003. Distributed construction of random expander graphs. In *Proceedings of The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'2003).* San Francisco, California, USA.

LI, M. AND VITÁNYI, P. 1997. *An Introduction to Kolmogorov Complexity and its Applications*, 2nd ed. Springer Verlag.

LOGUINOV, D., KUMAR, A., RAI, V., AND GANESH, S. 2003. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *Proceedings of ACM SIGCOMM 2003.* ACM Press, 395–406.

MARSAGLIA, G. 1995. *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness.* Florida State University. online at `http://www.stat.fsu.edu/pub/diehard`.

MARSAGLIA, G. AND TSANG, W. W. 2002. Some difficult-to-pass tests of randomness. *Journal of Statistical Software 7,* 3, 1–8.

MONTRESOR, A., JELASITY, M., AND BABAOGLU, O. 2004. Robust aggregation protocols for large-scale overlay networks. In *Proceedings of The 2004 International Conference on Dependable Systems and Networks (DSN).* IEEE Computer Society, Florence, Italy, 19–28.

NEWMAN, M. E. J. 2002. Random graphs as models of networks. In *Handbook of Graphs and Networks: From the Genome to the Internet*, S. Bornholdt and H. G. Schuster, Eds. John Wiley, New York, NY, Chapter 2.

PANDURANGAN, G., RAGHAVAN, P., AND UPFAL, E. 2003. Building low-diameter peer-to-peer networks. *IEEE Journal on Selected Areas in Communications (JSAC) 21,* 6 (August), 995–1002.

PASTOR-SATORRAS, R. AND VESPIGNANI, A. 2001. Epidemic dynamics and endemic states in complex networks. *Physical Review E 63*, 066117.

PeerSim. PeerSim. `http://peersim.sourceforge.net/`.

PITTEL, B. 1987. On spreading a rumor. *SIAM Journal on Applied Mathematics 47,* 1 (February), 213–223.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM).* ACM, ACM Press, San Diego, CA, 161–172.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware 2001*, R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 2218. Springer-Verlag, 329–350.

SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. 2003. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems Journal 9,* 2 (August), 170–184.

Sen, S. and Wang, J. 2004. Analyzing Peer-to-Peer Traffic Across Large Networks. *IEEE/ACM Transactions on Networking 12,* 2 (April), 219–232.

Stavrou, A., Rubenstein, D., and Sahu, S. 2004. A Lightweight, Robust P2P System to Handle Flash Crowds. *IEEE Journal on Selected Areas in Communication 22,* 1 (January), 6–17.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM).* ACM, ACM Press, San Diego, CA, 149–160.

Sun, Q. and Sturman, D. 2000. A Gossip-Based Reliable Multicast for Large-Scale High-Throughput Applications. In *International Conference on Dependable Systems and Networks.* IEEE, IEEE Computer Society Press, Los Alamitos, CA., 347–358.

van Renesse, R., Birman, K. P., and Vogels, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems 21,* 2 (May), 164–206.

van Renesse, R., Minsky, Y., and Hayden, M. 1998. A gossip-style failure detection service. In *Middleware '98.* IFIP, The Lake District, England, 55–70.

Voulgaris, S., Gavidia, D., and van Steen., M. 2005. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management 13,* 2, 197–217.

Voulgaris, S. and van Steen, M. 2003. An Epidemic Protocol for Managing Routing Tables in very large Peer-to-Peer Networks. In *14th Workshop on Distributed Systems: Operations and Management.* Lecture Notes on Computer Science, vol. 2867. IFIP/IEEE, Springer-Verlag, Berlin, 41–54.

Watts, D. J. and Strogatz, S. H. 1998. Collective dynamics of 'small-world' networks. *Nature 393*, 440–442.

Zhong, M., Shen, K., and Seiferas, J. 2005. Non-uniform random membership management in peer-to-peer networks. In *Proc. of the IEEE INFOCOM.* Miami, FL.

Received Month Year; revised Month Year; accepted Month Year