AVMON: Optimal and Scalable Discovery of Consistent Availability Monitoring Overlays for Distributed Systems*

Ramsés Morales and Indranil Gupta
Dept. of Computer Science, University of Illinois at Urbana-Champaign
{rvmorale,indy}@cs.uiuc.edu

Abstract

This paper addresses the problem of selection and discovery of a consistent availability monitoring overlay for computer hosts in a large-scale distributed application, where hosts may be selfish or colluding. We motivate six significant goals for the problem - consistency, verifiability, and randomness, in selecting the availability monitors of nodes, as well as discoverability, load-balancing, and scalability in finding these monitors. We then present a new system, called AVMON, that is the first to satisfy these six requirements. The core algorithmic contribution of this paper is a protocol for discovering the availability monitoring overlay in a scalable and efficient manner, given any arbitrary monitor selection scheme that is consistent and verifiable. We mathematically analyze the performance of AVMON's discovery protocols, and derive an optimal variant that minimizes memory, bandwidth, computation, and discovery time of monitors. Our experimental evaluations of AVMON use three types of availability traces - synthetic, from PlanetLab, and from a peer-to-peer system (Overnet) - and demonstrate that AVMON works well in a variety of distributed systems.

Keywords: *Churn, Availability, Monitoring, Overlay, Consistency, Scalability, Optimality.*
Technical Areas: *Fault-Tolerance and Dependability, Peer-to-Peer.*

1. Introduction

Large-scale distributed applications running atop PlanetLab [11] and Enterprise Grids [18], as well as on top of peer-to-peer (p2p) systems, have to deal with the phenomenon of *churn*. Churn refers to rapid and continuous arrival and departure, failure, and birth and death, of computer hosts (nodes) in the system. Such availability variation across nodes and across time, has recently led to the design of many *availability-aware* strategies for distributed computing problems such as replication, multicast, etc.

*This work was supported in part by NSF CAREER grant CNS-0448246.

However, such availability-aware strategies necessarily rely on the presence of an underlying *availability monitoring service*. The high-level goal of an availability monitoring service is to maintain *long-term* availability information for *each host* (i.e., for each node) in the system. While a few availability monitoring solutions have been proposed in the literature (e.g., [3, 4, 12]), the generic availability monitoring problem has not been addressed as yet. This paper is the first to explicitly define goals for the availability monitoring problem, to address these goals with a general and overlay-independent solution, and to explore the optimality of discovery protocols for the overlay.

The problem challenge in the availability monitoring overlay problem comes from the fact that nodes may be selfish or colluding, thus reporting higher-than-measured availabilities for themselves and their "friend" nodes (we will elaborate on this soon). Yet, there are many applications that rely on such a service. Examples include availability-based replica selection [3, 4, 7, 14], availability-based parent selection in overlay multicast trees [7], and implementation of availability-based reliability predicates for multicast [12]. In fact, Godfrey et al recently showed in [7] that with detailed availability history about each node in the system, one can design "smart" node selection strategies for replication of a service or a file, and that these outperform availability-agnostic strategies. Finally, availability histories of nodes can even be used to predict availability of individual nodes in the future, e.g., [9].

Concretely, this availability monitoring problem consists of two orthogonal sub-problems: I. Selection and Discovery of the *Availability Monitoring Overlay:* for each node x, select and discover a list of nodes who monitor node x, and II. *Availability History Maintenance:* what is the exact mechanism used by a monitor of a given node x to store x's availability history. While several different techniques have been proposed for the sub-problem II, i.e., how a monitor maintains history (see raw and aged techniques in, e.g., [3, 9]), the solution space for the sub-problem I is relatively less-explored.

Our focus in this paper is only on the more challenging

sub-problem I above: selection and discovery of the Availability Monitoring Overlay. Formally, this problem can be stated as follows (following the notation of [12]). For each node x, select and discover a small subset of nodes to monitor x. Denote this monitoring set of x as $PS(x)$, called the *pinging set of node* x. Each of the nodes in $PS(x)$ is responsible for monitoring node x's long-term availability history. Similarly, node x might in turn be asked to monitor the availability of a small set of other nodes - this is called $TS(x)$, or the *target set of* x. The TS and PS relationships are inverses of each other.

We assume a system model whereby nodes may be selfish, and colluding, i.e., have a constant number of colluding friends that misreport its availability[1]. A selfish node that reports higher than actual availability[2] for itself could manage to obtain higher reliability from multicast [7, 12], or cause service outage for systems that are rely on high availability nodes [3, 4, 7, 14]. This makes the design of an availability monitoring service challenging, and none of the existing solutions in literature appear to solve this problem.

Design Goals: To address the above challenges, we specify six goals for our problem. The first three goals (*consistency*, *verifiability*, and *randomness*) are for the selection of the pinging set of a node. We also desire quick *discoverability* of a node's pinging and target sets, in a manner that is *load-balanced* and *scalable*. We state these concretely below.

1. *Consistency:* Given two nodes x, y, the relationship of whether or not $y \in PS(x)$, should be consistent, i.e., this relationship should not change due to any factors such as joining and leaving of nodes in the system, the size of the system, etc. This ensures that each node will always be monitored by a consistent set of other nodes, regardless of whatever else happens in the system. This is useful for maintaining long-term availability history, avoiding transferring histories on churn, and avoiding pollution of monitoring sets with colluders.

2. *Verifiability:* Given two nodes x, y, any third node should be able to correctly verify whether $y \in PS(x)$ or not. This is an important requirement as this prevents selfish nodes from selecting and advertising its colluding nodes as being in its $PS(x)$.

3. *Randomness:* This requirement stands for uniform randomness, and says that given a node x, $PS(x)$ should contain other nodes picked uniformly at random, in an: (a) in an identically distributed fashion, and (b) independently of one another. Condition (a) both reduces the chances of a node's colluder being one of its monitors, and helps in load-balancing. Condition (b) avoids having groups of nodes being correlated in being present together in several pinging sets. If malicious, such correlated nodes can jeopardize availability calculation for all nodes they are monitoring.

4. *Discoverability:* Any node x should be able to discover its $PS(x)$ and $TS(x)$ quickly. Further, any other node y should be able to locate at least a constant number of (any) given node x's $PS(x)$. This enables protocols using the availability service, e.g., [3, 4, 7, 9, 12, 14], to gather information about individual nodes' availability[3].

5. *Load Balancing:* For discovery of pinging sets, the message overhead, memory overhead, and computational overhead, should each be uniformly distributed across all nodes.

6. *Scalability:* For discovery of pinging sets, the per-node message overhead, computational overhead, and memory overhead should each be low and scalable.

Existing Solutions: Existing availability monitoring schemes in literature fall into three categories: (1) [Self-reporting] relies on a node reporting its own availability (i.e., $PS(x) = \{x\}$). (2) [Central] uses a central availability monitor (i.e., $PS(x) = y_0$, where y_0 is a specific node or a small fixed subset of nodes). (3) [DHT-based] uses a p2p DHT (distributed hash table, e.g., [13]) overlay to decide the monitoring set for a node, e.g., akin to [3, 4, 16].

Each of these schemes has disadvantages, and none of these schemes satisfies all of the conditions (1)-(6) above. Self-reporting allows nodes to lie about their own availability. Central monitoring is neither load-balanced, nor scalable. Finally, the DHT-based approach typically decides the $PS(x)$ as the set of neighboring K nodes with id's around a hash of the nodeID of x (i.e., a "replica set" around this hashed value). This approach does not satisfy either consistency or randomness, and has problems w.r.t. verifiability[4]. Consistency may be violated when there is churn, e.g., the set of nodes around the hash of nodeID of x can change as nodes join and leave in that region. This causes frequent transfers of node x's availability history across churned nodes. Randomness is violated because the condition 3(b) above is not satisfied - two nodes y, z that are in a $PS(x)$ are likely to also appear together in other pinging sets (of other nodes whose nodeIDs hash nearby). Finally, verifiability could be expensive under node churn - churn in the DHT ring region where a node's pinging set lies is expensive for other nodes to keep track of.

In-Brief Contributions of this Paper: This paper presents AVMON, the first complete system for selection and discovery of an availability monitoring overlay, in order to satisfy all the six properties of consistency, verifiability, randomness, discoverability, load-balance, and scalability. The two core algorithmic contributions of the current paper are:

[1] Our goal is to select $PS(x)$ in a manner that reduces the effect of existing colluders of x; thus we do not target situations where nodes become compromised *after* they have a monitoring role assigned to them.

[2] Avoiding availability under-reporting requires tracking nodes' application activity, and thus an application-specific solution. Our goal is only application-independent solutions.

[3] We do not consider the problem of aggregating node availability histories in this paper.

[4] Other hash-based approaches relying on a DHT have similar drawbacks.

(i) a distributed, efficient, scalable, and load-balanced algorithm for *discovery* of monitors according to *any* consistent and verifiable selection scheme, and (ii) derivation of an optimal variant of this discovery protocol, in order to optimize memory and communication bandwidth, discovery time, and computation complexity. The AVMON system leverages the consistent hash-based pinging set selection from [12]. The AVMON system itself also includes practical optimizations for our algorithms in order to address high-churn systems. We have implemented AVMON, and our evaluation using three types of churn traces - synthetic, from PlanetLab, and from a p2p system (Overnet) - shows its usefulness in a variety of distributed systems.

2. Other Related Work

Distributed membership maintenance protocols have been the focus of several researchers. The goal of these protocols is to have each node maintain a *neighbor list*, which then defines a membership graph in the system. Full neighbor lists, resulting in a complete membership graph, are maintained by approaches such as SWIM [5] and via gossip [15]. Others address maintenance of partial membership lists at nodes, creating a system-wide membership graph that is random (but neither consistent nor verifiable). SCAMP [6] works by having each joining node initiate several joining requests which then undergo random walks and probabilistic inclusion in the membership lists of recipient nodes. CYCLON [17] works by having each node periodically exchange its neighbor lists with a random neighbor, and pick a new neighbor list from the union of these lists. T-Man [8] supports a generic class of membership graph predicates, including random membership graphs.

However, none of the above systems addresses availability monitoring as a first class problem. While they target randomness, they do not tackle consistency or verifiability. Yet, AVMON's mechanisms bear some similarities to these systems - like CYCLON, we continuously change neighbor lists, and like SCAMP, we spread joining node information to a small subset of nodes.

Finally, the hash-based monitoring relationship in this paper is borrowed from our previous work [12]. However, that paper [12] did not address scalable discovery of monitors (as is the focus of the current paper). Instead, [12] had each node broadcast a message (to everyone in the system) whenever it joined the system, thus resulting in a linear and unscalable bandwidth.

3. AVMON Design

This section first discusses the system model, then an overview of AVMON, and finally its core algorithms.

System Model and Problem: We assume a distributed system where each node may leave, fail in a crash-stop manner, and rejoin the system, at any time. In addition, nodes may be born (i.e., join for the first time), and may also die (i.e., leave the system for good). Deaths are silent and not explicit, i.e., a node may leave or fail for the last time without specifying it was a death. We assume that communication between pairs of nodes is reliable and timely if both nodes are currently alive.

Nodes are assumed to have persistent storage that can be retrieved after a failure or a rejoin - this will be used to store availability information about other monitored nodes. Finally, we also assume that the system has a stable system size (i.e., number of alive nodes), and this is known a priori as N. This assumption is true in practice, as even high-churn p2p systems have a stable system size; the actual system size varies always within a constant factor of the stable value [2].

AVMON Overview: First (Section 3.1), AVMON relies on a hash-based implementation of the monitoring relationship. This hash-based function can be executed by any node in the system, and determines if a given arbitrary pair of nodes x, y is related by $y \in PS(x)$ (or vice-versa). We chose a hash-based implementation because it is consistent, verifiable, and random. Second (Section 3.2), in order to discover *any* consistent and verifiable monitoring relationships (such as the one in Section 3.1), AVMON maintains and uses a *coarse overlay*. Each node maintains a fixed-size neighbor list, called the *coarse view*, that is a random subset of the remaining nodes in the system. This coarse overlay is used by nodes to discover monitoring relationships between other pairs of nodes and inform the relevant nodes of such discovery. The protocol for coarse overlay maintenance and neighbor discovery is scalable, and load-balanced. Finally (Section 3.3), the $PS(.)$ sets are used to monitor other nodes in a scalable manner, with optimizations.

3.1. Consistent Monitor Selection

In order to decide when a node y is a monitor for a node x, in a consistent, verifiable, and random manner, AVMON leverages a hash-based scheme from our previous work [12]. Given $< IPaddress, portnumber >$ pairs for two nodes x and y, we use a consistent hash function H with range normalized to real interval $[0, 1]$ (MD-5 or SHA-1 could be used). Then, nodes x, y are related by the following *hash-consistent condition*:

$$y \in PS(x) \Longleftrightarrow H(y, x) \leq \frac{K}{N}$$

Here, K is a small fixed number (typically a constant) and N is a fixed parameter that reflects the expected system size. Our analysis holds even if N is off by a constant factor from the current system size. It is easy to see that an expected $O(K)$ nodes will be present in $PS(x)$ for any node x. It is also evident that this relationship is consistent, random, and verifiable at any third node.

3.2. Monitor Discovery via Coarse View

We describe an efficient, scalable, and load-balanced protocol for discovering monitors according to *any arbitrary* monitor selection scheme, as long this scheme is consistent and verifiable. For concreteness however, we assume the hash-based monitor selection from Section 3.1. Monitors are discovered via each node x maintaining a *coarse view* $CV(x)$, a random subset of other nodes in the system. The size of the coarse view is a maximal cvs entries. The goal is to have this coarse view be random. This is achieved by the two sub-protocols below.

I. Joining Sub-Protocol: Node x initiates this protocol whenever it either joins the system freshly (i.e., after being born), or rejoins it. The goal of this protocol is to inform a set of other nodes (expected number of cvs) about node x, at any point of time. The protocol works by having x create a JOIN message, specifying its own id (x), and an integer *weight*. The weight is selected so that an expected cvs nodes point to node x at any time; we detail this after sub-protocol (II) below. This JOIN message is sent to a random node to start with. When a node y receives such a JOIN message with a non-zero weight c, it first checks if either x is already present in $CV(y)$, or if $|CV(y)| = cvs$. If neither is true, then y includes x in its CV, and decrements the weight value c of the JOIN. In any case, if $c = 0$ now, then y forwards two JOIN messages with weights set to $(\lfloor \frac{c}{2} \rfloor)$ and $(\lceil \frac{c}{2} \rceil)$ respectively, each to a random node from its $CV(y)$.

II. Coarse View Maintenance and Discovery: This maintenance sub-protocol is executed at each node once every *protocol period* (also known as "rounds")- protocol period durations are fixed at nodes, but are executed asynchronously across nodes.

Each protocol period, this sub-protocol at node x has three tasks: to eliminate from $CV(x)$ nodes that have left the system (and may or may not rejoin), to shuffle $CV(x)$ with new entries (to keep it random), while discovering monitoring relationships in the process. First, x picks a single node z uniformly at random from $CV(x)$, and pings it - an unresponsive node is removed[5]. This implies that a dead node z (i.e., one that has left for good) will *eventually* be deleted from all coarse views that contained z (Theorem 2 in Section 4.1). Further, since an expected cvs nodes know about any given node z, and the probability of any of these nodes picking z to ping is $\frac{1}{cvs}$ per protocol period, the expected number of nodes that delete a non-alive node z from their coarse view is $cvs \times \frac{1}{cvs} = 1$ per protocol period. Second, node x also picks a random and alive node $w \in CV(x)$, and fetches its coarse view $CV(w)$. It checks the hash-consistent condition (Section 3.1) among all pairs of nodes (u, v) and (v, u), where $u \in CV(x) \cup \{x\}, v \in CV(w) \cup \{x, w\}$, and $u \neq v$. Any node pair (u, v) discovered to satisfy the hash-consistent condition are informed via a NOTIFY message sent to both. Thirdly, to maintain randomness of coarse views, node x selects a new coarse view $CV(x)$ by selecting cvs elements at random from the set $CV(x) \cup CV(w)$ [6].

We discuss the initial weight assigned to a JOIN message ((I) above). A freshly joining node (being born) sets this to cvs. On the other hand, for a re-joining node x, this weight is set to the minimum of cvs, and the number of protocol periods elapsed since the last departure of node x from the system. This is because, once node x leaves the system, sub-protocol II ensures that the average rate at which nodes delete x from their own coarse view is 1 per protocol period.

3.3. Using the Monitoring Overlay

This section briefly discusses how the monitoring overlay is used to track availability, how nodes report their monitors, and an important optimization. From the previous section, whenever a NOTIFY(u, x) message is received at node x, if node u is not already present in $PS(x)$, the hash-consistent condition $H(u, x) \leq \frac{K}{N}$ is re-checked and if true, node u is included in $PS(x)$ (i.e., node x will be monitored by node u from now on). Similarly, a node x receiving a NOTIFY(x, u) message executes actions to check whether u should be included in $TS(x)$ or not.

The availability of a node's target set is monitored by periodically sending *monitoring ping*s to these nodes, once each *monitoring period*. This period is different from the protocol period for coarse view maintenance. This measured availability information can then be maintained either raw, aged, or recent, or via other orthogonal techniques.

Whenever a node y wants to discover a given node x's pinging set nodes, *it is the burden of node x to report to node y the requisite number of its monitoring nodes*. For instance, node y's policy may be to require x to report at least $l \leq K$ monitoring nodes. Node x can then select any l of its $PS(x)$ nodes to report to y, but cannot lie about these, since y can check the hash-consistent condition for each reported monitor. Node y can then ask each reported monitor individually for x's availability history.

Optimization - Forgetful Pinging: When nodes die, $TS(x)$ and $PS(x)$ may be filled with garbage nodes that may never join the system again. These garbage entries cannot be deleted since there is no way of knowing whether the pointed-to node will rejoin or not. Instead, we reduce the frequency of monitoring pings sent to such nodes. If a node u in $TS(x)$ has been unresponsive for time t, and $t > \tau$, where τ is a time-threshold, and $t_s(u)$ was the last up-time (session time) for node u measured at node x, then pick u to ping with probability $\frac{c \cdot t_s(u)}{t_s(u)+t}$, per monitoring protocol period. This will reduce the frequency of pings sent to nodes

[5]Notice that this pinging is *not* the same as the availability monitoring (which will be discussed in Section 3.3).

[6]Observe that this protocol means that at any time at node x, the hash-consistent conditions for all pairs within $CV(x)$ have already been checked.

that have left the system a while ago. Section 5 evaluates the forgetful pinging optimization.

4. AVMON Analysis and Optimality

We present the basic analysis of AVMON, its optimality analysis, and the effect of colluding nodes. An extended analysis of AVMON is available in our technical report [10].

4.1. Basic Analysis

In the following analysis, we assume $cvs = o(\sqrt{N})$.

Dissemination Time of JOIN information: Since the weight of the very first JOIN(x, w) message from a freshly born node x is set to cvs, no more than cvs nodes can add x to their coarse views right after x's birth. In addition, when the node rejoins, it sets the initial weight to make up for the expected lost number of entries pointing to it. thus the expected number of coarse views pointing to x stays at cvs.

We upper-bound the expected dissemination time of a newly born node x's first JOIN message; this bound also applies to rejoining nodes. Notice that the spread of JOIN$(x, .)$ messages, via the random coarse view graph, is akin to building a random spanning tree with cvs total nodes (internal + leaf nodes). This gives a spread time of $O(log(cvs))$ time for the JOIN information, *unless a large number of nodes receive duplicate JOIN(x,.) messages*. In fact, this "unless" clause is improbable - the probability of a given node receiving a JOIN$(x, .)$ message in a given round with m forwarders is $= 1-(1-\frac{2}{N})^m \leq 1-(1-\frac{2}{N})^{cvs} \simeq \frac{2 \cdot cvs}{N}$, since $cvs = o(\sqrt{N})$. Thus, the expected number of duplicate JOIN-receiving nodes, per protocol period, is $\leq cvs \times \frac{2 \cdot cvs}{N} = o(1)$. In addition, the probability that none of the cvs nodes receive duplicate JOIN messages is $=(1-\frac{2}{N})^{cvs} \simeq (1-\frac{2 \cdot cvs}{N})$, which $\rightarrow 1$ as $N \rightarrow \infty$.

Thus, with high probability, the JOIN$(x, .)$ spreads quickly to the requisite cvs nodes in time that is $O(log(cvs))$. In the worst case, this time is $O(log(N))$.

Discovery time of Monitors (D): Firstly, we have the following theorem for eventual discovery of monitors:

Theorem 1: If (x, y) satisfy the hash-consistent condition, and if nodes x, y stay alive (i.e., online) long enough, then x will eventually discover y, i.e., $y \in TS(x)$ eventually.

This is because y will see x an infinite number of times in its coarse view, and thus node y is guaranteed to eventually pick x to exchange coarse views with, during which y will check for the hash-consistent condition with x.

Secondly, we show that discovery is in fact fast. We do so by upper-bounding the expected discovery time, where we are interested only in an asymptotic bound. Given a pair of nodes x, y, the discovery of the monitoring relationship between x and y occurs at the *first instance* when *some* node (not necessarily either x or y) checks for the hash-consistent condition with the pairs (x, y) and (y, x). Based on the coarse view maintenance protocol, this check happens only during the coarse view fetches. Given a node u that fetches the coarse view of another node w, the probability that x will be present in $CV(u)$ and that y will be present in $CV(w)$, is $= (\frac{cvs}{N} \times \frac{cvs}{N}) = \frac{cvs^2}{N^2}$. Thus, the probability that the (x, y) pair is *not* checked by this particular coarse view fetch is $\leq (1 - \frac{cvs^2}{N^2})$ (ignoring the residual probability of $y \in CV(u)$ and $x \in CV(w)$, since we are interested only in asymptotic bounds). Now, notice that per protocol period, there are a total of N such coarse view fetches. Putting all this together, we can derive the probability of the pair (x, y) being checked by at least one of the fetches in one protocol period as greater than or equal to:

$$1-(1-\frac{cvs^2}{N^2})^N = 1-((1-\frac{cvs^2}{N^2})^{\frac{N^2}{cvs^2}})^{\frac{cvs^2}{N}} \geq 1-e^{\frac{-cvs^2}{N}}$$

Thus, the expected time to discovery of the monitoring relationship for an arbitrary node pair (x, y) can be bounded as: $E[D] \leq \frac{1}{1-e^{\frac{-cvs^2}{N}}}$ protocol periods. Now, with $cvs = o(\sqrt{N})$ and $N \rightarrow \infty$, the exponential series expansion can be used to simplify this as: $E[D]_{\text{upper-bound}} \simeq \frac{N}{cvs^2}$.

Effect of Dead Nodes: From the coarse view maintenance protocol of Section 3.2, we have:

Theorem 2: A dead node z (i.e., one that has left for good) will *eventually* be deleted from all coarse views.

Further, if a dead node $z \in CV(x)$, then x will delete z w.h.p. $\frac{1}{N}$ in $T^* = (cvs \cdot log(N))$ protocol periods. This is because the probability of deletion of z from $CV(x)$ in T rounds is $= 1-(1-\frac{1}{cvs})^T$, which is $\simeq (1-\frac{1}{N})$ for $T = T^*$.

If $N_{longterm}$ is the number of nodes that has been born in the system recently, then the expected size of $TS(x)$ is $K \cdot N_{longterm}/N$. For minimal-death PlanetLab-like Grid systems, N can be chosen to be the maximal number of machines, thus $E[|TS(x)|] \leq K$. For p2p systems, if $N_{longterm}$ is within a constant factor of N, $E[|TS(x)|]$ is still bounded.

Memory, Message Overhead (M): The per-node x memory is $(|CV(x)| + |PS(x)| + |TS(x)|)$, or $O(cvs + 2K) = O(cvs)$, since typically $K < cvs$. The per-node message overhead, per protocol period, for the coarse view maintenance protocol is $O(cvs)$ bytes, and for the monitoring protocol is $O(K)$ bytes. The first of these terms dominates, but stays small. With $cvs = O(\sqrt[4]{N})$ (Optimal-MDC - see Section 4.2 below) at $N = 1$ Million (thus, $cvs = 32$), protocol period=1 second, and 8 Bytes per entry, the per-node bandwidth is $256Bps$, which is reasonably small.

Computational Overhead (C): This is $O(cvs^2)$ per protocol period per node, and arises from the hash-consistency checking in the coarse view fetches. For $N = 1$ Million, $cvs = \sqrt[4]{N}$, this turns out to be 1000 hash computations per protocol period. However, this is quite fast - [1] shows that, with the C++ implementation of MD-5 hash running on a 2.1 GHz P4, WinXP SP 1 machine, 1000 hash computations (each with 12 B) complete in about 0.375 ms. This

is reasonably small, e.g., a protocol period of 10 s implies that 0.003% CPU time used for the hashes.

4.2. AVMON Optimal-MDC Variant

We would like to minimize memory utilization and message bandwidth (M) at each node, the discovery time ($E[D]$), as well as the computational overhead (C). Ignoring the different units of each, this problem reduces to minimizing the function[7]: $g(cvs) = M + C + E[D] = cvs + cvs^2 + \frac{1}{1-e^{\frac{-cvs^2}{N}}} \simeq cvs + cvs^2 + \frac{N}{cvs^2}$.

Differentiating g w.r.t. cvs gives us $\frac{d(g(cvs))}{d(cvs)} = (1 + 2cvs - \frac{2 \cdot N}{cvs^3}) = 0$, or $cvs_{Optimal-MDC} \simeq \sqrt[4]{N}$. Notice that $\frac{d(g(cvs))^2}{d^2(cvs)} = 2 + \frac{6 \cdot N}{cvs^4} > 0$ at this optimum, thus implying it is a minimum of g. This gives a memory and per-round bandwidth of $O(\sqrt[4]{N})$ each, an expected discovery time of $\sqrt{N}$, and a computational overhead of $O(\sqrt{N})$ (with different units in each case). In comparison, the broadcast discovery algorithm of [12] had an $O(log(N))$ discovery time, but at the expense of $O(N)$ per-node bandwidth and memory.

4.3. Collusion-Resilience

We analyze the probability of pollution of $PS(.)$ due to colluders. Suppose node x in the system has $C(= o(\frac{N}{log(N)}))$ colluding nodes that are willing to over-report x's availability, should they become its monitors. Given $K = O(log(N))$, the probability that *no* such colluders appear in $PS(x)$ is $= (1 - \frac{K}{N})^C \simeq (1 - \frac{CK}{N}) \rightarrow 1$ as $N \rightarrow \infty$. Thus, it is probabilistically impossible for a node to have its $PS(.)$ set polluted by any colluders.

To extend this system-wide, assume D total colluding relationships in the system (across any colludee-colluder node pair), and $D = o(\frac{N}{log(N)})$, with $K = O(log(N))$. Then, the probability that the $PS(.)$ relationships capture none of these colludee-colluder pairs is $= (1 - \frac{K}{N})^D \simeq (1 - \frac{DK}{N}) \rightarrow 1$ as $N \rightarrow \infty$. Hence, none of the colluders will have any effect on the system, w.h.p.

5. Experimental Evaluation

We implemented AVMON in C, and present experimental results from a trace-driven discrete event simulation in this section. All tests were run on a 2.80GHz Intel P4 with 2GB RAM under Fedora Core 4 Linux.

We studied the effect of five different availability (or churn) models. These fall into three classes - (I) synthetic churn models (labeled in our plots as `STAT`, `SYNTH`, and `SYNTH-BD`), (II) churn traces from PlanetLab all-pairs-pings (labeled `PL`) [7], and (III) traces from the Overnet p2p system (labeled `OV`) [2]. `STAT` models a static network with no churn, while `SYNTH` models a system where nodes join and leave under exponential and memoryless rates (each $\frac{0.2N}{60}$ per min, giving per-hour a 20% join rate, and 20% leave rate), but there are no births or deaths. `SYNTH-BD` extends this with node birth and death, following exponential and memoryless rates (each $\frac{0.2N}{1440}$ per min, giving per-day a 20% birth rate, and 20% death rate). All models in category (I) ensure a stable system size, i.e., number of online nodes. `PL`, `OV` neither have a stable system size nor an exponential assumption, but instead reflect the true availability traces. `PL` and `OV` traces were originally once each sec and once each 20 min respectively, and are injected as such.

In all these scenarios, AVMON used the following default settings: (1) protocol period (see Section 3.2) $T = 1$ min; (2) per-node coarse view size $cvs = 4 \cdot \sqrt[4]{N}$, where N was the stable system size for `STAT`, `SYNTH`, `SYNTH-BD`, and the long-term average system size for `PL` and `OV`;[8] (3) parameter $K = log_2(N)$ (see Section 3.1); (4) For $H(.)$, libSSL's MD5 implementation using only the first 64 returned bits; (5) forgetful pinging parameters $\tau = 2$ min, $c = 1$ (see Section 3.3); (6) monitoring protocol period $T_A = 1$ min (see Section 3.3).

Each experiment below was run for 48 hours, and points on plots show the average across relevant nodes.

I. Effect of System Size and Churn: We varied N from 100 to 2000 for `STAT`, `SYNTH`, `SYNTH-BD`. An initial warm-up period of 1 hour was used, where nodes were allowed to be born, die, join, and leave according to the respective model. For each of `STAT` and `SYNTH`, a control group, consisting of a new set of nodes numbering 10% of the stable system size, was then made to join simultaneously, in order to measure discovery time of their monitors. The new nodes followed the respective availability model. For `SYNTH-BD`, the control group was implicit, i.e., consisted of nodes born after the warm-up stage.

Discovery Time: Although the expected number of monitors K was set as $log_2(N)$, this experiment at first focuses only the time to find, for each node in the control group, *at least one* of its monitors. Henceforth, we call this as the "discovery time" of the first monitor of that node. For instance, this could be used to satisfy the "l out of K" policy (see Section 3.3) with $l = 1$. Figure 1 plots the average time to discovery of the first monitor for each node in the control group[9]. This plot illustrates two observations about the average discovery time: (1) it stayed consistently below 1 min (recall that the protocol period itself was 1 min), and (2) it was not affected by joins and leaves (compare `STAT` and `SYNTH` lines), although it appeared to be affected by births and deaths (see `SYNTH` and `SYNTH-BD` lines) [10].

Yet, discovery stayed fast in spite of births and deaths.

[7] Using a constant multiplicative factor with each of the terms (to account for their different units), would yield the same asymptotic result.

[8] We set $cvs = 4 \cdot cvs_{Optimal-MDC}$ for performance reasons.

[9] In calculating average discovery time for a setting, we ignored the highest outlier. The top ignored values were 110 min, 72 min, and 42 min.

[10] For the `SYNTH-BD` setting, discovery time was measured from among the new nodes born after the warm-up - these numbered 47 for $N = 100$, 198 for $N = 500$, 390 for $N = 1000$, and 785 for $N = 2000$.

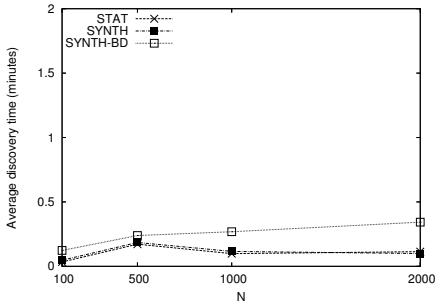

Figure 1. Average discovery times of first monitors for the control group nodes introduced in the three synthetic models.

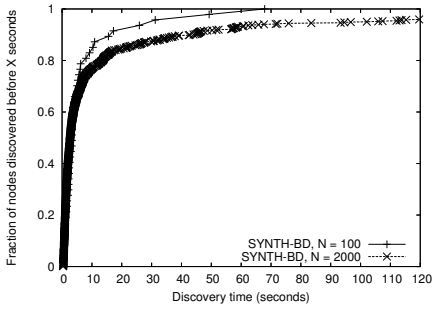

Figure 2. CDF of the SYNTH−BD points in Figure 1. For all values of N from 100 to 2000, at least 93.3% of the nodes were discovered within 60 seconds.

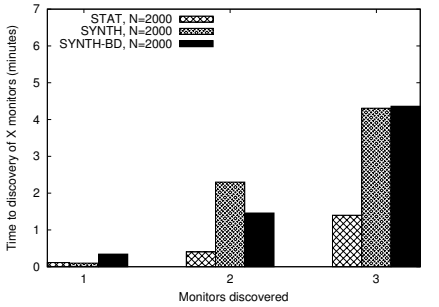

Figure 3. Average discovery times of first L monitors (L on x-axis) for each node in control group, for the three synthetic models.

Figures 2 shows the cumulative distribution function (CDF) of the discovery time for SYNTH−BD. In each setting, at least 93% of first monitors were discovered within 60 sec. Figure 3 shows that given a node x, $PS(x)$ nodes were discovered at uniform time intervals. We conclude that AVMON's discovery time is scalable, and low in spite of churn.

II. Effect of Coarse View Size: In order to isolate the effects of cvs from churn, only the STAT model was used. Four different values for cvs were used: $4 \cdot \sqrt[4]{N}$, $6 \cdot \sqrt[4]{N}$, $8 \cdot \sqrt[4]{N}$, and $10 \cdot \sqrt[4]{N}$. Figure 4 shows that the discovery time (both average and standard deviation) decreased as cvs was increased. For each value of N, the curve has a knee at the third data point - increasing cvs beyond this ($cvs = 8 \cdot \sqrt[4]{N}$, e.g., for $N = 2000$, this is $cvs \simeq 55$), did not improve either the average or variance of discovery

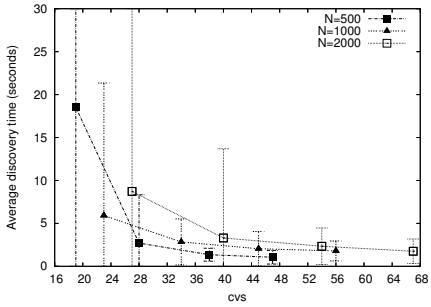

Figure 4. Average discovery time (with 1 standard deviation) vs. cvs on STAT churn model.

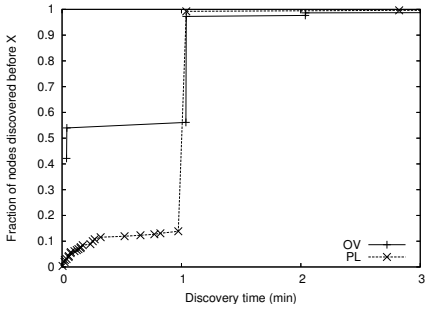

Figure 5. CDF of Discovery time of first monitors, for each of PL and OV traces.

time much. The per-node computational overhead was very small – even for $cvs = 68$, the expected 9375.79 hashes would take only about 3.52 ms to execute (this is once per min)[11]. Since $K = 11$ and there were 8 B per entry, the memory usage was 720 B, and the bandwidth was 632 Bps. Thus, cvs should be set based on the knee of Figure 4.

III. PlanetLab and Overnet Traces: For PL, we had the stable system size $N = 239$, and $K = 8, cvs = 16$. For OV, we had $N = 550$, and $K = 9, cvs = 19$. Figure 5 shows the CDF for the discovery time of the first monitor for each node. In OV, a total of $N_{longterm} = 1319$ nodes had been born after two days - 97.27% of these had discovered their first monitors within 63 sec after birth. For PL, $N_{longterm} = 239$ after 2 days, and over 98% of nodes' first monitors were discovered in around a minute after birth.

IV. Forgetful Pinging: To evaluate the benefits of forgetful pinging (see Section 3.3), we used SYNTH. For $N = 2000$, Figure 6 plots the ratio, for each node in the control group, of its raw estimated availability (averaged over its AVMON monitors) to its real availability (actual fraction uptime). While the lack of the forgetful ping optimization ("NON-Forgetful ping in plot") measured availability accurately, the plot for the "Forgetful ping" optimization had an average relative error of less than 5%, with a maximal error of 8%. Figure 7 shows that the optimization reduced bandwidth consumed by "useless pings" (sent by a node to nodes not currently in the system), by an order of magnitude.

[11] With C++ Visual .Net 2003 on 2.1 GHz P4 under WinXP SP 1.

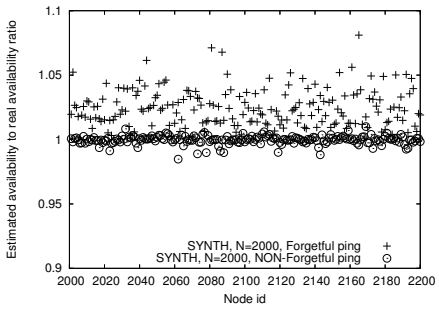

Figure 6. Ratio of estimated availability to actual availability, with and without the forgetful pinging optimization.

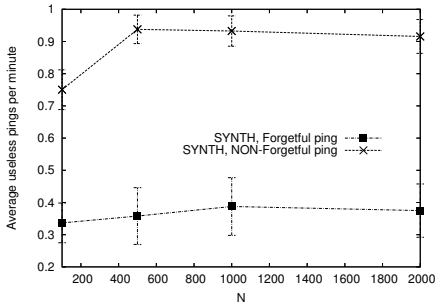

Figure 7. Forgetful pinging reduces useless pings sent to absent nodes. Bars show 1 standard deviation.

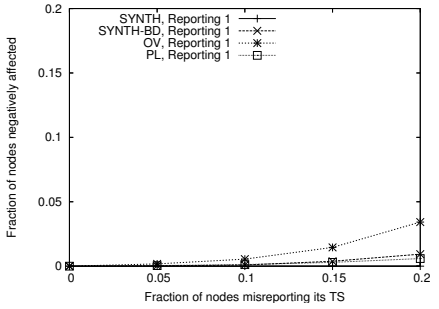

Figure 8. Fraction of nodes with above 0.2 error in measured availability, with some nodes overreporting all $TS(.)$ nodes' availabilities as 100%.

VI. Overreporting Attack: Finally, Figure 8 had a fraction of nodes (x-axis) report 100% availabilities for *all* their $TS(.)$ nodes. The plot shows that the fraction of nodes whose measured availability (averaged over their $PS(.)$ nodes) differed from their actual availability by over 0.2, was very small for all the four models `SYNTH`, `SYTNH-BD`, `PL`, `OV`.

6. Conclusions

We presented and evaluated AVMON, a system that selects and discovers an availability monitoring overlay for long-term host-level availability information in distributed systems. The core algorithmic contribution of this paper is a protocol for fast, load-balanced, and scalable discovery of each node's monitors, given any arbitrary monitor selection scheme that is consistent and verifiable; for our implementation, we used a hash-based monitor selection. An optimal variant of the discovery protocol was then derived to minimize memory, bandwidth, discovery time, and computation time. AVMON performed well under three synthetic churn models (static, join-leave, and join-leave-birth-death), and availability traces from PlanetLab and Overnet.

References

[1] Speed benchmarks for MD5 and other crypotgraphic functions. http://www.eskimo.com/∼weidai/benchmarks.html.

[2] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. IPTPS*, pages 135–140, Feb. 2003.

[3] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proc. Usenix NSDI*, pages 337–350, 2004.

[4] B.-G. Chun and et. al. Efficient replica maintenance for distributed storage systems. In *Proc. Usenix NSDI*, pages 45–58, 2006.

[5] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-consistent Infection-style process group Membership protocol. In *Proc. IEEE DSN*, pages 303–312, 2002.

[6] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE TOC*, 52:139–149, Feburary 2003.

[7] P. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proc. ACM SIGCOMM*, 2006.

[8] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay toplogy management. *Self-Organising Systems: ESOA*, LNCS 3910:1–15, July 2005.

[9] J. W. Mickens and B. D. Noble. Exploiting availability prediction in distributed systems. In *Proc. Usenix NSDI*, pages 73–86, 2006.

[10] R. Morales and I. Gupta. AVMON: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. Technical Report UIUCDCS-R-2006-2797, UIUC, 2007.

[11] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. ACM HotNets-I*, pages 59–64, 2002.

[12] T. Pongthawornkamol and I. Gupta. AVCast : New approaches for implementing availability-dependent reliability for multicast receivers. In *Proc. IEEE SRDS*, pages 345–354, 2006.

[13] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, pages 329–350, 2001.

[14] T. Schwarz, Q. Xin, and E. L. Miller. Availability in global peer-to-peer storage systems. In *Proc. WDAS*, 2004.

[15] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. Middleware '98*, 1998.

[16] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for p2p resource sharing. In *Proc. Wshop. Economics P2P Syst.*, 2003.

[17] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *JNSM*, 13(2):197–217, June 2005.

[18] Open Grid Forum. http://www.ogf.org/.