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

Ramses V. Morales and Indranil Gupta†

Department 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 range of protocols 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 optimal variants that minimize memory, bandwidth, computation, and discovery time of monitors (or different combinations of these metrics). 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.*

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

†Corresponding Author.

1 Introduction

Large-scale distributed applications running atop PlanetLab [10] 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., [3, 4, 7, 9, 11, 14]. These strategies aim to make such distributed systems churn-resistant and and churn-adaptive.

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, 11]), 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, the benefits of an availability monitoring service that overcomes this challenge, are numerous and varied. For instance, such a service can be used for availability-based replica selection [3, 4, 7, 14], availability-based parent selection in overlay multicast trees [7], and for implementing availability-based reliability predicates for multicast [11]. 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, e.g., [3, 9]), the solution space for the sub-problem I is relatively less-explored. In other words, any existing technique for availability history maintenance, such as raw, aged, recent, etc. [9], can be used orthogonally with any availability monitoring overlay.

Thus, our focus in this paper is only on the more challenging sub-problem I above: of selection and discovery of the Availability Monitoring Overlay. Formally, this problem can be stated as follows (following the notation of [11]). 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. That is, each node would like to have its availability seen as high as possible by the system. In addition, a given node may have up to a constant number of colluders ("friends") that always misreport its availability. This makes the design of an availability monitoring service challenging, and none of the existing solutions appear to satisfy this model.

To address the above challenges, we specify six goals for our problem. The first three goals are for the selection of the pinging set of a node - *consistency*, *verifiability*, and *randomness*. In addition, we would also desire *discoverability*, i.e., a node should be able to locate its pinging set and target set quickly and easily, and in a manner that is *load-balanced* and *scalable*. These six requirements can be stated concretely as follows:

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, with 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. Consistency is also required in order maintain long-term availability history of node x at each of its monitors, and to avoid having to transfer such histories upon node churn. Consistency can avoid pollution of monitoring node sets with colluders. Finally, consistency is related to the next important property called Verifiability.

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:* For each node x, $PS(x)$ should consist of a random selection of nodes from across the system, and this selection should be completely uncorrelated with the $PS(y)$ of any other node y. Formally, this can be understood as two requirements: (a) [randomness] every node should be picked as a candidate for $PS(x)$ with the same likelihood as any other node, i.e., given three distinct nodes x, y, z, and that $y \in PS(x)$, then $Pr(z \in PS(x)|y \in PS(x)) = Pr(z \in PS(x))$; and (b) [non-correlation] given four distinct nodes w, x, y, z, and that two nodes $y, z \in PS(x)$, and also that $y \in PS(w)$, then $Pr(z \in PS(w)|y, z \in PS(x)\&y \in PS(w)) = Pr(z \in PS(w))$. Condition (a) is important in order to have each node be monitored by others that are less likely to be collusive with it and report a high availability for it. In addition, this also helps in load-balancing. Condition (b) is needed to avoid having groups of nodes (e.g., y and z) that are

correlated to be present together in several pinging sets. If such correlated nodes start maliciously over-(or under-)reporting availabilities, availability information for several other nodes could get jeopardized.

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, 11, 14], to gather information about individual nodes' availability[1].

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 availability monitoring overlay schemes in literature today predominantly adopt one of the following three approaches - self-reporting, centralized, or DHT-based. (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 [13, 15]) overlay to decide the monitoring set for a node, e.g., akin to [3, 4].

Each of the above existing schemes has disadvantages, and none of these schemes satisfies all of the conditions (1)-(6) we laid down above. Firstly, self-reporting does not follow randomness and thus allows nodes to potentially lie about their own availability by reporting arbitrarily high values. Secondly, central monitoring is neither load-balanced, nor scalable.

Finally, the DHT-based approach typically decides the $PS(x)$ for a node x based on the position of the hash of x in a DHT ring, by selecting the neighboring K nodes with id's around the hashed nodeID of x (i.e., a "replica set" around a hashed value). This approach does not satisfy either Consistency or Randomness, and has problems w.r.t. Verifiability[2]. Consistency may be violated when there is churn, e.g., consider a newly born node joining very close to the hashed value of x; this changes the set of monitoring nodes. This could thus cause 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 hash to nearby points on the ring, and thus are likely to appear together in other pinging sets as well (of other nodes that hash nearby). Finally, verifiability could be expensive under node churn - birth of new nodes which hash on the ring in between a previously verified monitor's hash and x's hash will require monitors to be updated. Such overhead is avoided by consistent monitor selection in our new system called AVMON.

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

[2]Other approaches based on the hashing of x also suffer from the same disadvantages.

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: (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 optimal variants of this discovery protocol, in order to optimize different combinations from among the metrics of memory and communication bandwidth (M), discovery time (D), and computation complexity (C). Specifically, for (ii), we discover three variants of AVMON that satisfy different optimality conditions - MD (optimal w.r.t. M and D), DC (optimal w.r.t. D and C), and MDC (optimal w.r.t. M, D, and C). The AVMON system uses the consistent hash-based pinging set selection leveraged from [11] - we describe this briefly and assume it in the rest of the paper. 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 peer-to-peer system (Overnet) - shows that AVMON works well in a variety of churned distributed systems.

The rest of this paper is organized as follows. Section 2 discusses other related work. Section 3 describes the basic AVMON algorithms, and Section 4 analyzes the discovery algorithms, while also deriving optimal variants. Section 5 presents experimental results, and Section 6 concludes.

2 Other Related Work

Section 1 discussed existing availability monitoring approaches, and systems that use such a monitoring service. This section briefly touches on other work that is related to our approach.

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. SWIM [5] and the gossip-based membership protocol by van Renesse et al. [16] use probabilistic mechanisms to have each node maintain a complete neighbor list of all nodes in the system (i.e., a complete membership graph).

Several systems have aimed to build a random membership graph among the nodes of a distributed system (without attention to consistency or verifiability). The SCAMP membership system [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. The CYCLON system [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] is yet another membership protocol that is able to support a generic class of membership graph predicates, thus allowing arbitrary membership graphs (such as random ones) to be formed by local node actions.

Somewhat like SCAMP, AVMON uses a random spanning tree approach to have joining nodes inform a subset of other nodes of their presence. AVMON also uses a mechanism similar to (but simpler than) CYCLON to constantly keep changing the neighbor lists of nodes. While the goal of CYCLON was to have the neighbor lists change to combat system churn, AVMON's goal is to also use the neighbor lists to discover monitoring relationships among node pairs. Although AVMON has some design decisions similar to some of the above systems (as noted), none of these systems addresses availability monitoring as a first class problem. While the constructed membership graphs are random, the conditions of consistency and verifiability are not addressed by these systems.

Finally, as noted before, the monitoring relationship in this paper is borrowed from our previous work [11]. However, that paper [11] did not address scalable discovery of monitors (as is the focus of the current paper). Instead, [11] had each node broadcast (to everyone in the system) whenever it joined the system - this led to quick discovery, but used a very high message bandwidth. We label this approach of [11] as *Broadcast*, and compare our new AVMON approaches analytically with it (see Table 1 in Section 4).

3 The AVMON Approach - Overview and Algorithm

This section first discusses the system model, then gives a brief overview of the AVMON approach, and then describes individual components of the AVMON algorithms.

System Model and Problem: We assume a distributed system where each node may leave, fail in a crash-stop manner, or 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. As noted in Section 1, each node may be selfish and colluding. That is, each node would like to have its availability seen as high as possible by the system. In addition, a given node may have up to a constant number of colluders ("friends") that always misreport its availability. 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].

Recall that our problem is to select and discover, for each node x, a set of pinging set nodes $PS(x)$ that will monitor x's availability. In addition, node x needs to be informed of nodes in its target set $TS(x)$ (the inverse of $PS(.)$) that it needs to monitor. Our goal is to satisfy all the properties (1)-(6) described in Section 1 [3].

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 Monitor Selection Scheme - Using Consistent Hashing

This section describes a consistent, verifiable, and random, scheme that AVMON uses in selecting when a given node x is a monitor for another given node y. We leverage this scheme from our previous work [11]. 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), and two consistent parameters K and N to determine if $x \in PS(y)$ (or $y \in PS(x)$). Here, K is a small fixed number (typically a constant) and N is a fixed parameter that reflects the expected system size. Even though distributed systems undergo churn, the actual system size stays quite stable [2], thus it suffices to set this N to a value that is approximate - the properties of our algorithms continue to hold as long as the real system size stays within a constant factor of parameter N.

Given this, two nodes x, y are related as:

$$y \in PS(x) \Longleftrightarrow H(y, x) \leq \frac{K}{N} \qquad \textbf{[Consistency Condition]}$$

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, verifiable, and random, at any third node.

[3]Although garbage collection of dead nodes is not an explicit goal, we will address it via the optimization called "Forgetful Pinging" in Section 3.3.

3.2 Monitor Discovery - Composition and Maintenance of the Coarse View

While the previous section described a specific consistent monitor selection scheme, this section aims at discovering monitors according to *any arbitrary* monitor selection scheme, as long this scheme is consistent and verifiable. Such a discovery protocol needs to be efficient, scalable, and load-balanced.

The discovery is done by having each node maintain a *coarse view*, a random subset of other nodes in the system. This coarse view is used to discover monitors for other nodes. Although the discovery protocol described below is generic, for concreteness we will assume the hash-based monitor selection from Section 3.1.

Each node x maintains a set of neighbors in its coarse view, denoted as $CV(x)$. The size of each node's coarse view is limited to a maximal *cvs* entries. In order to maintain randomness of the coarse view at each node, we describe two sub-protocols below: (1) the (re-)joining sub-protocol executed for nodes (re-)entering the system, and (2) the coarse view maintenance and monitor discovery sub-protocol.

Joining Sub-Protocol: Figure 1 describes the join 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 have an expected *cvs* other nodes know 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* (values of which are detailed in the next paragraph). 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 x is already present in $CV(y)$. If not, y includes x in its CV, and decrements the weight value c of the JOIN. Finally, 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 $CV(y)$.

The goal of this protocol is to have an expected *cvs* other nodes include given node x in their coarse views, by creating a random spanning graph with the requisite number of internal and leaf nodes. Thus, the initial weight assigned to the JOIN message by a freshly joining node (being born) is thus *cvs*. On the other hand, for a node x that is rejoining the system, this weight is set to the minimum of *cvs*, and the time elapsed since the last departure of node x from the system (time in "protocol periods", term defined in the next paragraph). This is because, once node x leaves the system, the protocol described next (in Figure 2) ensures that the average rate at which nodes delete x from their own coarse view is 1 per protocol period. Section 4 analyzes the join sub-protocol in detail, showing that it is unlikely for multiple JOIN messages to be received at a node, and that the spread time of the JOIN information is $O(log(cvs))$ w.h.p.

Coarse View Maintenance and Discovery Sub-Protocol: Figure 2 shows the pseudocode for maintaining the coarse view and for discovering monitoring relationships. The maintenance protocol described in the figure 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. The sub-protocol at node x has

```
For node $x$ to (re-)join the system:
    Pick a random node $y$;
    if (this is the first join of this node)
        Send $y$ a JOIN$(x, cvs)$ message (with an integer weight of $cvs$);
    else
        let $t_{down} = \frac{\text{time since the latest leave from system of this node}}{\text{protocol period duration}}$
        Send $y$ a JOIN$(x, min(cvs, t_{down}))$;
    Inherit view from this random node;
    Participate in the Coarse Membership Protocol (Figure 2);

Each node $y$ receiving a JOIN$(x, c)$ message:
    if$(c \leq 0)$
        discard JOIN message; return;
    if ($x$ is not already present in $CV(y)$)
        add $x$ to $CV(x)$;
        $c := c - 1$;
    Create two messages JOIN$(x, \lfloor \frac{c}{2} \rfloor)$ and JOIN$(x, \lceil \frac{c}{2} \rceil)$;
    Send each of these messages to a randomly selected node from $CV(x)$;
```

Figure 1: **Joining (and Rejoining) Sub-Protocol for Nodes.**

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.

The detailed protocol operations are as follows. Once during each protocol period, node x picks a single node z uniformly at random from its $CV(x)$, and pings it - an unresponsive node is removed from the CV[4]. Observe that 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).

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. Recall that this motivated the weights assigned

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

```
At Node $x$: Periodically (once every protocol period)
    Pick a random node $z$ from $CV(x)$ and ping it;
    If ($z$ does not respond)
        Remove $z$ from $CV(x)$;
    Pick a random alive node $w$ from $CV(x)$;
    Fetch $CV(w)$ from $w$;
    Check all $(u, v)$ pairs $(u \neq v)$ in $(\{CV(x) \cup \{x, w\}\} \times \{CV(w) \cup \{x, w\}\}) \cup$
                        $(\{CV(w) \cup \{x, w\}\} \times \{CV(x) \cup \{x, w\}\})$
            for the consistency condition $H(u, v) \leq \frac{K}{N}$;
        for (each pair $(u, v)$ discovered to satisfy the above consistency condition)
            Inform both nodes $u$ and $v$ of the match by sending a NOTIFY$(u, v)$ ;
    $CV(x) := \emptyset$;
    $CV(x) :=$ Subset of $cvs$ random entries from $CV(x) \cup CV(w) \cup \{w\}$;
```

Figure 2: **Coarse Membership and Monitor Discovery Sub-Protocol.**

to JOIN messages in Figure 1.

During each protocol period, node x also picks a random and alive node $w \in CV(x)$, and fetches its coarse view $CV(w)$. It checks the consistency condition (Section 3.1) among all pairs of nodes (u, v) and (v, u), where $u \in CV(x) \cup \{x, w\}, v \in CV(w) \cup \{x, w\}$, and $u \neq v$. Any node pair (u, v) discovered to satisfy the consistency condition are informed via a NOTIFY message sent to both. Finally, to maintain randomness of coarse views, node x selects a new coarse view $CV(x)$ by selecting *cvs* elements at random from the set $CV(x) \cup CV(w) \cup \{w\}$ [5]. Section 4 will show that (1) if a node and its potential monitor both stay alive for long enough, they will *eventually* discover each other (Theorem 1); (2) the expected discovery time is small for reasonably small values of *cvs*.

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 consistency condition $H(u, x) \leq \frac{K}{N}$

[5]Observe that the above protocol implies that at any time at node x, the consistency condition for all pairs within $CV(x)$ have already been checked.

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, on receipt of a NOTIFY(x, u) message at node x, the corresponding consistency condition is checked and if true, node u is included in $TS(x)$ (i.e., node x will monitor u's availability from now on).

Each node x periodically sends *monitoring ping*s to each of the nodes in $TS(x)$, and records this information in its persistent storage. Please note that monitoring pings are different from the pings in the protocol of Figure 1. Monitoring pings may be sent out at a frequency different from the protocol period of the coarse view membership protocol, i.e., monitoring pings are sent out once every *monitoring period*, which may be different from the protocol period of Figure 2. As noted in Section 1, the granularity in which measured availability information about $TS(x)$ nodes is stored at x can be arbitrary, i.e., stored availability information could either be raw, aged, or recent, etc.

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 consistency condition for each reported monitor. Node y can then ask each reported monitor individually for x's availability history. Section 4.3 analyzes how large K needs to be to support such an "l out of K" policy.

Optimization - Forgetful Pinging: When nodes die, $TS(x)$ and $PS(x)$ may be filled with garbage nodes that may never join the system again. Since deaths are silent, there is no way of knowing whether an unresponsive node will rejoin or not, thus these garbage elements cannot be deleted. Instead, we propose here an optimization to reduce the rate of monitoring pings sent to these unresponsive nodes. The goal here is to reduce the bandwidth consumption due to dead nodes.

The optimization works as follows at node x - 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 (alternatively, this could be exponentially averaged), then pick u to ping with probability $\frac{c \cdot t_s(u)}{t_s(u)+t}$, per monitoring protocol period. The goal here is to reduce the load of pinging, yet ensure that on average, between two successive joins, node u receives at least an expected c monitoring pings from each of its $PS(u)$ nodes. Section 5 evaluates the effectiveness of this optimization.

4 Analysis: AVMON Performance and Optimal Variants

We first analyze in Section 4.1 the performance of the joining and coarse view maintenance protocols (from Section 3.2), then Section 4.2 studies different optimal variants of AVMON discovery. Section 4.3 discusses values for K and the effect of colluding nodes.

4.1 Basic Analysis

As a precursor to our optimality discussion, this section first analyzes the join, coarse view maintenance, and discovery protocols of Figures 1,2. Then, we derive the memory, message, computation and discovery time complexity for the AVMON protocol. We will assume that $cvs = o(\sqrt{N})$ in all of the discussion below.

Coarse View Analysis: Spread and 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 lost number of entries pointing to it (Figure 1). thus keeping (as Section 3.2 explained) at cvs the expected number of coarse views pointing to x.

We analyze the expected dissemination time of a newly born node x's first JOIN message. This upper-bound also holds for JOINs sent by 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 upper-bounded by $cvs \times \frac{2 \cdot cvs}{N} = o(1)$. 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 the Consistency Condition for a node pair (D): Firstly, we have the following theorem for eventual discovery of monitors:

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

This arises because node y will see node 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 (Figure 2), during which y will check for the consistency 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 consistency condition with the pairs (x, y) and (y, x). Based on the protocol of Figure 2, 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 $x \in CV(x)$, 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:

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

Thus, the expected time to discovery of the monitoring relationship for an arbitrary node pair (x, y) can be bounded as (in number of protocol periods):

$$E[D] \leq \frac{1}{1 - e^{\frac{-cvs^2}{N}}}$$

Notice that with $cvs = o(\sqrt{N})$ and $N \to \infty$, the exponential series expansion can be used to simplify this as: $E[D]_{\text{upper-bound}} \simeq \frac{N}{cvs^2}$. We will assume $cvs = o(\sqrt{N})$ henceforth in the paper.

Effect of Dead Nodes: From the first three lines of Figure 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 that contained z.

In addition, a node x with dead node $z \in CV(x)$, 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 $T = T^*$.

Memory, Message Overhead (M): The size of the memory at each node x is $(|CV(x)| + |PS(x)| + |TS(x)|)$, which is $O(cvs + 2K) = O(cvs)$. The message overhead for the Coarse Membership Protocol (Figure 2) is $O(cvs)$ bytes per protocol period. The message overhead for the monitoring protocol is $O(K)$ per monitoring protocol period. Since K is usually smaller than cvs, the primary bandwidth consumption arises from the coarse view fetches. With $cvs = O(\sqrt[4]{N})$ (Optimal-MDC - see Section 4.2 below), this turns out to be a very small overhead. For $N = 1$ Million, $cvs \simeq 32$, a protocol period=1 second, and with 6 Bytes per entry, the per-node bandwidth consumption is $192Bps$, which is reasonably small.

Computational Overhead (C): This is $O(cvs^2)$ per protocol period per node, and arises from the consistency checking following the coarse view fetches (Figure 2). For $N = 1$ Million, and $cvs = \sqrt[4]{N}$ (Optimal-MDC - see Section 4.2 below), this turns out to be 1000 hash computations per protocol period.

However, MD5 is a very fast hash function - [1] shows a processing capacity of $32 MBps$ (on a Pentium 4 2.1 GHz processor under Windows XP SP 1, C++ program on Visual .NET 2003), which means that 1000 hash computations per protocol period (each with 12 B) will take about 0.375 ms. This is reasonable since the length of a protocol period in our Coarse Membership Protocol is very large (several seconds). For instance, a protocol period duration of 10 s implies that 0.003% of CPU time is used for such hash calculations.

4.2 Optimal Variants of AVMON

From the analysis in the previous section, the size of the coarse view (*cvs*) determines a tradeoff between memory, communication bandwidth, and computation on the one hand, and discovery time on the other hand. However, an application may be interested in optimizing only some of these metrics, and not others. For instance, if AVMON is run within the hosts of an in-house PC cluster connected over a high bandwidth LAN, we may be interested in minimizing only the computation (C) and discovery time (D), but not the message or memory. Alternatively, AVMON being run in a PlanetLab-like cluster with multicore machines may not care about computation, but would like to optimize discovery time (D), and memory and bandwidth (M). Of course, there are applications that would like to optimize all of M,D,C.

Below, we mathematically derive the values of *cvs* for each of these three optimal variants (MD, MDC and DC), and then discuss the practicality of these optimal variants[6]. For each of these, we will assume that $cvs = o(\sqrt{N})$.

Optimality Analysis 1 - Optimal-MD: We would like to minimize both memory utilization and message bandwidth (M) at each node (*cvs* units), as well as the discovery time (D) which is $E[D] = \frac{1}{1-e^{\frac{-cvs^2}{N}}}$. Thus, we want to minimize the function: $f(cvs) = cvs + \frac{1}{1-e^{\frac{-cvs^2}{N}}} \simeq cvs + \frac{N}{cvs^2}$.

Differentiating f w.r.t. *cvs* gives us:

$$\frac{d(f(cvs))}{d(cvs)} = (1 - \frac{2 \cdot N}{cvs^3}) = 0 \Rightarrow cvs_{Optimal-MD} = \sqrt[3]{2N}$$

Notice that $\frac{d(f(cvs))^2}{d^2(cvs)} = \frac{6 \cdot N}{cvs^4} > 0$ at this optimum, thus implying it is a minimum of f. This is the optimal *cvs* value to minimize memory, bandwidth, and discovery time. The optimal values of memory, per-node message bandwidth (M), and expected discovery time (D) are each $O(\sqrt[3]{2N})$ (different units in each case).

Optimality Analysis 2 - Optimal-MDC: We would like to minimize both memory utilization and message bandwidth (M) at each node (*cvs* units), the discovery time (D): $E[D] = \frac{1}{1-e^{\frac{-cvs^2}{N}}}$, as well as the computational overhead (C). Thus, we want to minimize the function:
$g(cvs) = cvs + cvs^2 + \frac{1}{1-e^{\frac{-cvs^2}{N}}} \simeq cvs + cvs^2 + \frac{N}{cvs^2}$.

[6]Notice that optimizing MC does not make sense since both M and C increase with *cvs*.

Approach	**Memory, Bandwidth per Round (M)**	**Expected Discovery Time (D)**	**Computations per Round (C)**
Broadcast (from [11])	$O(N)$	$O(log(N))$	(One-Time only)
AVMON, Generic cvs	$O(cvs)$	$\frac{1}{1-e^{\frac{cvs^2}{N}}}$	$O(cvs^2)$
AVMON, $cvs = log(N)$	$O(log(N))$	$\frac{N}{(log(N))^2}$	$O((log(N))^2)$
(Optimal-MD) AVMON, $cvs = \sqrt[3]{2N}$	$O(\sqrt[3]{2N})$	$\sqrt[3]{2N}$	$O((2N)^{2/3})$
(Optimal-MDC,-DC) AVMON, $cvs = \sqrt[4]{N}$	$O(\sqrt[4]{N})$	$\sqrt{N}$	$O(\sqrt{N})$

Table 1: **AVMON Variants: Performance of different algorithms for availability monitoring (*cvs*=Coarse View Size).**

Differentiating g w.r.t. cvs gives us:

$$\frac{d(g(cvs))}{d(cvs)} = (1 + 2cvs - \frac{2 \cdot N}{cvs^3}) = 0 \Rightarrow cvs_{Optimal-MDC} \simeq \sqrt[4]{N}$$

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).

Optimality Analysis 3 - Optimal-DC: To minimize the discovery time (D) and computation complexity (C), the reader will notice that the optimizing function can be derived in a manner similar to the function g in the Optimal-MDC analysis above, and gives an optimal $cvs_{Optimal-DC} = \sqrt[4]{N}$.

Table 1 summarizes the results of our optimality analysis, and also compares the benefits of the AVMON approach to the naïve Broadcast algorithm of [11].

In practice - Optimal-MDC AVMON ($cvs = \sqrt[4]{N}$): As mentioned previously, for $N = 1$ Million, $cvs = \sqrt[4]{N} = 32$, and $K = log_2(N) = 20$, the size of $CV(x)$ is $192B$, and the per-protocol period bandwidth is $192Bps$. The expected discovery time is 1000 time units, however a given node x discovers one new pinging set node on average at least every 50 protocol periods. The computational overhead due to hash calculation is also low.

As pointed out previously, the death of nodes from the system may cause $TS(x)$ to contain garbage entries. This is unavoidable since deaths are silent. However, 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. Further, our forgetful pinging optimization (Section 3.3) keeps the bandwidth effect of garbage entries referring to dead nodes, very low. Experiments in Section 5 elaborate on the benefits of forgetful pinging.

4.3 Continuous Monitoring and Collusion-Resilience

In this section, we analyze how large K (the pinging set size) has to be in order to ensure continuous monitoring of each node at all times. We also bound the minimal and maximal sizes of the pinging sets, and finally we analyze the effect of colluders in the system misreporting the availability of a given node.

Choosing K: Suppose the average node availability (system-wide average) is a. Then the probability that for node x, at least one $PS(x)$ node is up is $= (1 - (1-a)^K)$. If $K = c.log(N)$, then this probability can be written as $= 1 - N^{-c/log(1/(1-a))}$. Choosing c such that $c/log(1/(1-a)) \geq 2$, implies that the probability for each node x to have at least one $PS(x)$ node alive is:

$= (1 - N^{-c/log(1/(1-a))})^N \geq (1 - N^{-c/log(1/(1-a))+1}) \geq (1 - \frac{1}{N})$, which $\rightarrow 1$ as $N \rightarrow \infty$.

In order to have the *minimal size of any* $PS(.)$ at least $l(= O(1))$ nodes w.h.p, we need to choose $K = (l+1).log(N)$. This will help support "l out of K" policies such as described in Section 3.3. This derivation comes about because the probability of a node having fewer than l nodes in its $PS(.)$ is:

$= \sum_{i=1}^{l-1}[C_i^N \cdot (1 - \frac{K}{N})^{(N-i)} \cdot (\frac{K}{N})^i]$, which:

$$\leq [(1 - \frac{K}{N})^N \cdot \sum_{i=1}^{l-1} C_i^N] = [(1 - \frac{K}{N})^N \cdot O(N^{(l-1)})] \leq [e^{-K} \cdot O(N^{(l-1)})] = O(\frac{1}{N^2}).$$

Thus the probability that none of the N nodes will have fewer than l nodes in their $PS(.)$ is

$= (1 - O(\frac{1}{N^2}))^N \simeq 1 - O(\frac{1}{N})$.

Finally, Balls and Bins analysis ([12], Theorem 1) can be used to predict that having $K = O(log(N))$ will ensure that the *maximal size of any node's* $TS(.)$ *(and thus* $PS(.)$*)* set is $O(log(N))$ w.h.p.

Resilience to Collusion: The AVMON system avoids nodes misreporting their own availability, but there is the possibility that a node may be able to recruit *colluders* to misreport its availability. We analyze the probability of pollution of $PS(.)$ by such colluders. Suppose each node x in the system has $C(= o(\frac{N}{log(N)}))$ colluding nodes, i.e., nodes that are willing to misrepresent x's availability as a value higher than the measured value, or in the worst case, as 100%. Given $K = O(log(N))$, the probability that none of the colluders of a given a node x appear in its $PS(x)$ 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.

The above calculation was for one node x. Finally, notice if there are D total of such colluding relationships in the system (across any colludee-colluder pair of nodes) and $D = o(\frac{N}{log(N)}), K = O(log(N))$, then the probability that none of the colludee-colluder pairs are reflected in the system-wide $PS(.)$ relationships is:

$= (1 - \frac{K}{N})^D \simeq (1 - \frac{DK}{N}) \to 1$ as $N \to \infty$.

Thus, none of the colluders will have any effect on the system, w.h.p.

5 Experimental Evaluation

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

In order to test the efficacy of AVMON in practice, we studied individually the effect of injecting five different availability models, falling into three classes - (I) synthetic churn models (labeled in our plots as `STAT,SYNTH,` and `SYNTH-BD`), (II) availability traces from PlanetLab all-pairs-pings (labeled `PL`), and (III) churn traces collected from a deployed p2p system called Overnet (labeled `OV`). `STAT` models a static network with no churn, while `SYNTH` models a system with nodes joining and leaving according to exponential distributions (i.e., Poisson processes), but no births or deaths (i.e., no nodes that are new or leaving for good). `SYNTH-BD` extends the above models to include node birth and death according to exponential distributions (i.e., Poisson processes). All the above three models in (I) ensure a stable value for the system size, i.e., the current number of alive nodes in the system. The `PL,OV` models do not have the Poisson assumption and reflect the true availability variation in these systems. The PlanetLab host availability traces (for the `PL` experiments) [7] were done once each second, and were injected as such in the simulation. The Overnet availability traces were from the authors of [2], were measured once every 20 minutes (i.e., availabilities of all nodes were measured once every 20 minutes), and are injected as such in the simulation.

In what follows, we will distinguish results from the individual five scenarios mentioned above. However, in all these scenarios, the AVMON protocol used the following default settings: (1) the *protocol period* (see Figure 2) was $T = 1$ minute; (2) per-node coarse view size $cvs = 4 \cdot \sqrt[4]{N}$, where N was the stable system size for each of the `STAT,SYNTH,SYNTH-BD` settings, and was the long-term average system size for each of the `PL` and `OV` settings;[7] (3) parameter $K = log_2(N)$ (see Section 3.1); (4) `libSSL`'s MD5 implementation was used for the hash-based consistency condition, with only the first 64 bits returned considered (see Section 3.1); (5) the forgetful pinging parameters $\tau = 2$ minutes, $c = 1$ (see Section 3.3); (6) the monitoring protocol period $T_A = 1$ minute (see Section 3.3).

Before discussing the data, we need to mention a few more details about the five individual availability

[7]We set cvs a factor of 4 above $cvs_{Optimal-MDC}$ for performance reasons.

models used. The stable system size N for each of the STAT,SYNTH,SYNTH-BD models was varied from 100 to 2000. For SYNTH, we wanted to achieve a 20% per-hour rate of churn, akin to the Overnet traces [2], hence the leaving rate λ_l and rejoin rate λ_r were set as $\lambda_l = \lambda_r = \frac{0.2N}{60}$ per minute. In addition, for SYNTH-BD, we wanted to achieve a 20% per-day rate of births and deaths, thus the birth rate λ_b and death rate λ_d were set as $\lambda_b = \lambda_d = \frac{0.2N}{1440}$ per minute. Finally, the stable system size for PL was $N = 239$, and for OV was $N = 550$.

The metrics of interest measured in the following sections are discovery time, and memory, computation, and bandwidth overheads. Each experiment on each of the plots shown below was run for 48 hours. Each point on each plot depicts the average across relevant nodes considered.

5.1 Effect of Varying System Size in the STAT,SYNTH,SYNTH-BD Models

This section shows, for each of the synthetic models STAT,SYNTH,SYNTH-BD, the effect of different values of N, ranging from 100 to 2000, on the metrics of interest. For each of 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 3 plots the average time to discovery of the first monitor for each node in the control group[8]. This plot illustrates two observations about the average discovery time: (1) it stayed consistently below 1 minute (recall that the protocol period itself was 1 minute), 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 (compare SYNTH and SYNTH-BD lines) [9].

Yet, discovery stayed fast in spite of births and deaths. Figures 4 and 5, show the cumulative distribution function (CDF) of the discovery time for the two models STAT and SYNTH-BD respectively. Observe that in each of these settings, at least 93% of first monitors were discovered within 60 seconds. To wrap up, Figure 6

[8]For each point on the plot, the average was taken over all measured discovery times, but by ignoring the one highest measured discovery time datapoint for that setting. Such points are ignored because they are outliers (the top ignored values were 110 min, 72 min, and 42 min) and would skew our conclusions if included.

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

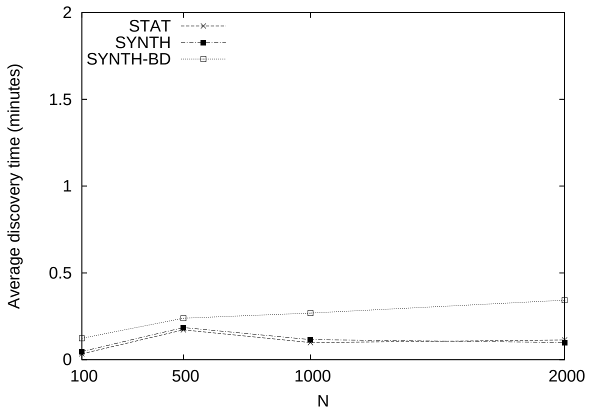

Figure 3: Average discovery times of first monitors for the control group nodes introduced in the three synthetic models.

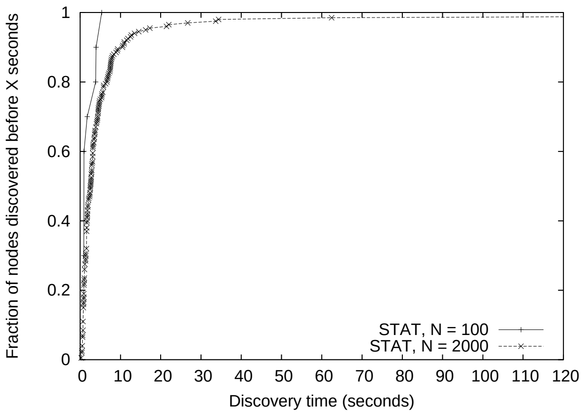

Figure 4: CDF of the `STAT` points in Figure 3. For all values of N from 100 to 2000, at least 96% of the nodes were discovered in under 30 seconds.

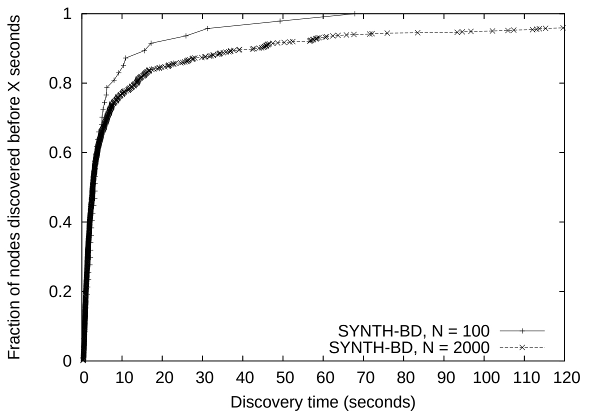

Figure 5: CDF of the `SYNTH-BD` points in Figure 3. For all values of N from 100 to 2000, at least 93.3% of the nodes were discovered within 60 seconds.

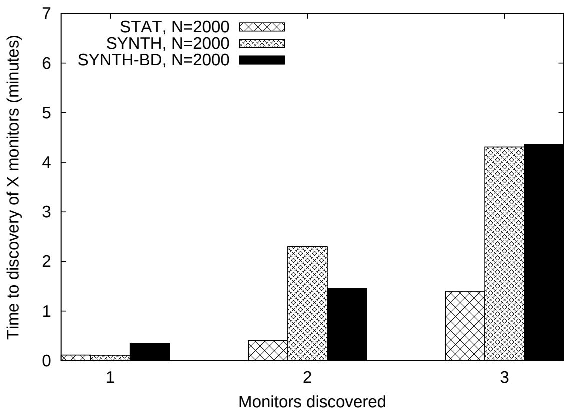

Figure 6: Average discovery times of first L monitors (L on x-axis) for each node in control group, for the three synthetic models.

shows the average times to discover, for each node, the first L monitors of the node. This plot illustrates that for each churn model, $PS(.)$ nodes were discovered at uniform time intervals. Thus, we conclude that AVMON's discovery time stays low, *regardless of birth, death, join and rejoin.*

Computational Overhead: Figure 7 shows that the average computations per second per node varied sublinearly with N, and the per-minute overhead was close to $2 \times cvs^2$. Figure 8 then illustrates the CDF of this distribution across nodes. This data showed us that even for $N = 2000$, the worst-case computation at any node was 0.611 ms per minute, which is about 1% of the CPU. Both plots, put together, show that the computational overhead of AVMON *is not influenced much by churn.*

Memory Usage: The total memory at each node x, due to $(CV(x)| + |PS(x)| + |TS(x)|)$, is plotted in

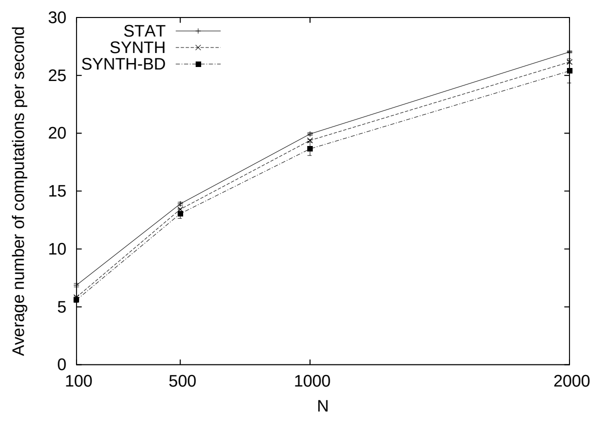

Figure 7: Average computations per time unit (with 1 standard deviation), averaged over all control group nodes, for `STAT,SYNTH,SYNTH-BD` models.

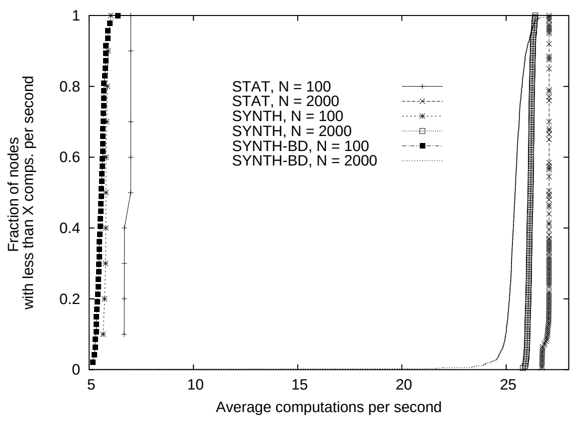

Figure 8: CDF of the computational overhead from Figure 7.

Figure 9, and the CDF of the distribution across all nodes is shown in Figure 10. The measured memory utilization was close to the expected value of $(2K + cvs)$. For instance, when $N = 2000$ (and $K = 11$, $cvs = 27$), Figure 9 shows that for `STAT`, the average per-node memory stayed below the expected value of 49 entries. For the churned models `SYNTH, SYNTH-BD`, the average value was only slightly above that expected, primarily because of garbage entries in the $PS(.), TS(.)$ sets. Finally, in Figure 10, from the CDF distributions for `STAT,SYNTH,SYNTH-BD` for $N = 2000$, we can conclude that the memory usage of AVMON is *minimally influenced by churn*.

Bandwidth: Although the upcoming Section 5.4 studies per-node outgoing bandwidth in detail, we can calculate this in the `STAT` model as $(K + cvs)$ per minute. For $N = 2000$ (with $K = 11, cvs = 27$), $8B$ per coarse view entry, $8B$ per monitoring ping message, this turned out to be $5.067Bps$ per node.

5.2 Effect of Varying Coarse View Size in `STAT` Model

As evident from our analysis of Section 4, varying the coarse view size cvs determines a tradeoff among discovery time, memory, bandwidth, and computation. This section studies the practical implications of varying cvs. In order to isolate the effects of cvs from churn, only the `STAT` model was used. We used four different values for cvs based on the stable system size: $4 \cdot \sqrt[4]{N}$, $6 \cdot \sqrt[4]{N}$, $8 \cdot \sqrt[4]{N}$, and $10 \cdot \sqrt[4]{N}$.

Figure 11 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 second data point - increasing cvs beyond this ($cvs = 8 \cdot \sqrt[4]{N}$, e.g., for $N = 2000$, this is $cvs \simeq 40$), did not improve either the average or variance of discovery time much. Figure 12 plots both the memory utilization (left y-axis) and the computational overhead per node (right y-axis). First, notice that for a fixed value of cvs, system size N had *no influence*

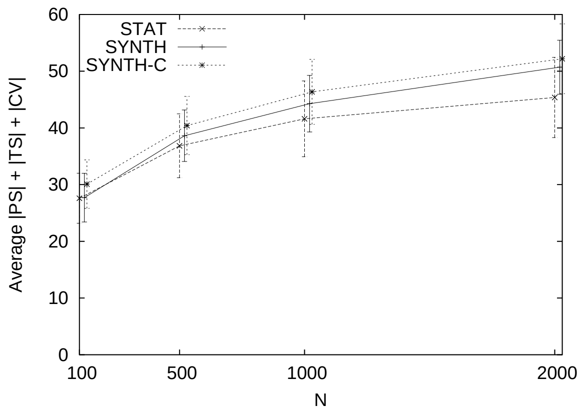

Figure 9: Average number of memory entries per node (with 1 standard deviation), over control group, for `STAT,SYNTH,SYNTH-BD`. Points perturbed slightly for clarity.

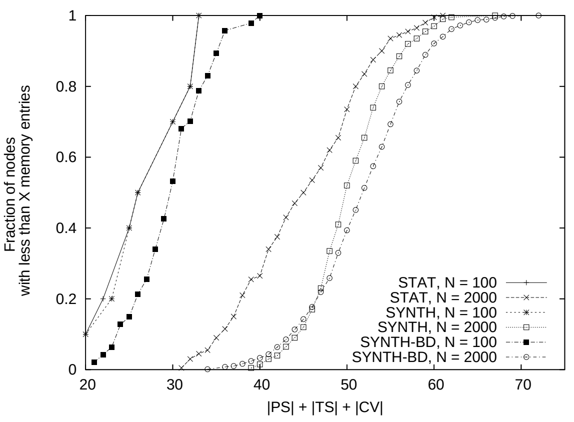

Figure 10: CDF of the memory overhead from Figure 9.

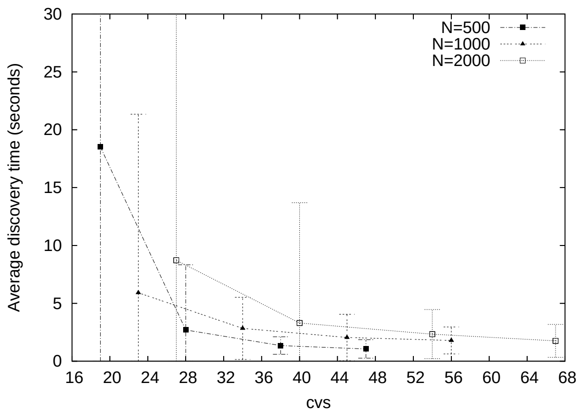

Figure 11: Average discovery time (with 1 standard deviation) vs. *cvs* on `STAT` churn model.

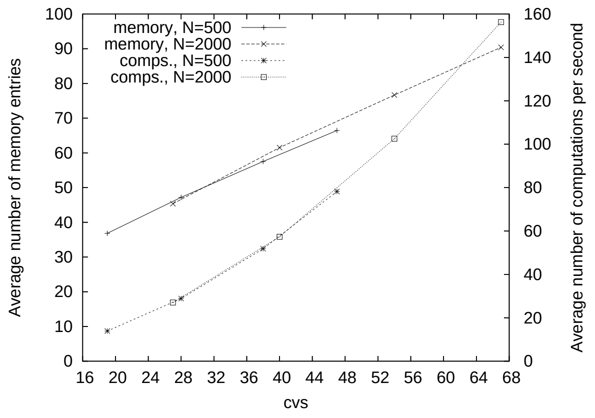

Figure 12: Memory entries vs. *cvs*, and computations per minute vs. *cvs* on `STAT` churn model.

on either memory or the computational overhead. Second, notice that increasing *cvs* too much beyond the knee of the curve derived from the previous Figure 11 raised the computational overhead. Yet, for practical purposes, the per-node computational overhead stayed so small as to be negligible – even for $cvs = 68$, the expected 9375.79 hashes would take only about 3.52 ms to execute (this is once per minute)[10]. Third, the memory varied linearly with *cvs* - even for $cvs = 68$, the memory usage was $728B$ (assuming $8B$ per coarse view entry). Thus, in practice, the value of *cvs* should be set based on the knee of the curve from Figure 11.

[10]Assuming a Pentium 4 2.1 GHz processor under Windows XP SP 1, C++ program on Visual .NET 2003.

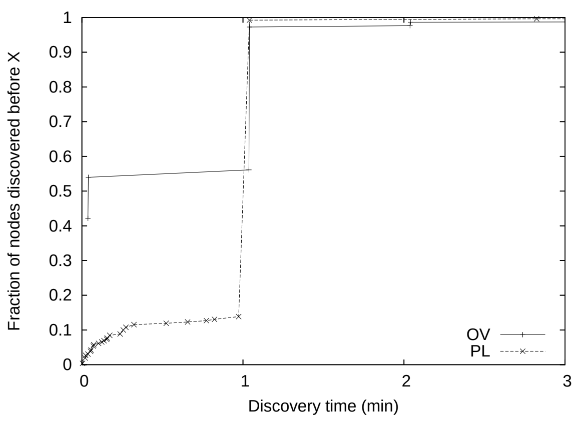

Figure 13: CDF of Discovery time of first monitors, for each of PL and OV traces.

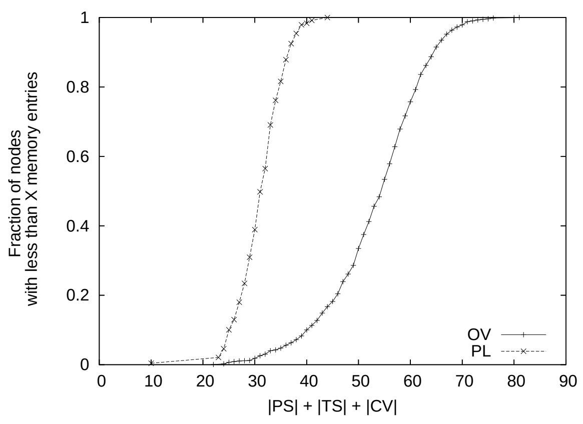

Figure 14: CDF of number of memory entries per node, for each of PL and OV traces.

5.3 Effect of Realistic and High Churn Models - PlanetLab and Overnet Traces

This section plots discovery time and memory utilization for AVMON under the per-second availability traces from PlanetLab (PL) and the per-20-minute availability traces from Overnet (OV). 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 13 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 seconds 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. Figure 14 shows that per-node memory utilization was uniformly distributed across nodes. The OV line in this plot shows that node birth and death cause memory utilization to be higher than the expected value (of $19 + 2 \times 9 = 37$), but that no node had over 81 total memory entries. For PL, this maximum number was measured to be 44.

Finally, to study the effect of very high churn, we used a new churn model called SYNTH-BD2, which resembled SYNTH-BD, except that its birth and death rate was twice as high (i.e., $\lambda_b = \lambda_d = \frac{0.4N}{1440}$ per min). Figure 15 shows that there was no noticeable difference between the discovery times in the two models, illustrating that AVMON discovery is churn-resistant. Figure 16 shows the additional memory entries due to the increased churn in SYNTH-BD2 was less than 10% over that in SYNTH-BD.

5.4 Forgetful Pinging, Optimizations, and Overreporting

Forgetful Pinging: We evaluated the practical benefits of the forgetful pinging optimization (see Section 3.3) for the SYNTH churn model. For this experiment, AVMON estimated each node's availability as the fraction of monitoring pings sent to that node which receive a response back. For $N = 2000$, Figure 17

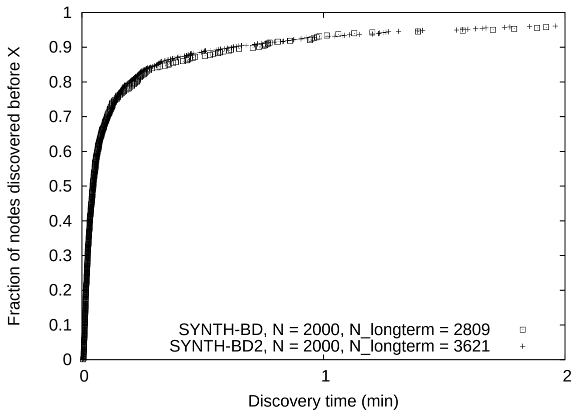

Figure 15: CDFs for the discovery time of first monitors, compared between two high-churn models.

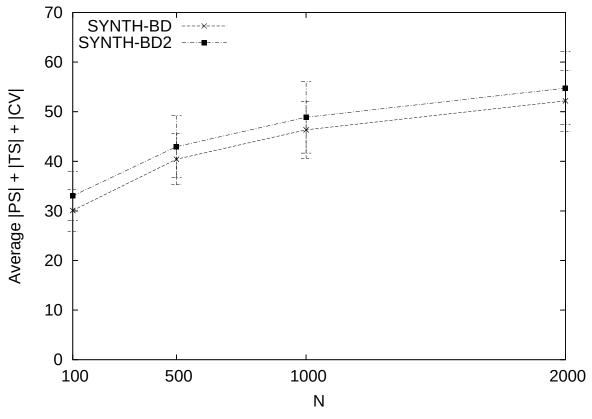

Figure 16: Average number of memory entries (with 1 standard deviation) for the experiment in Figure 15.

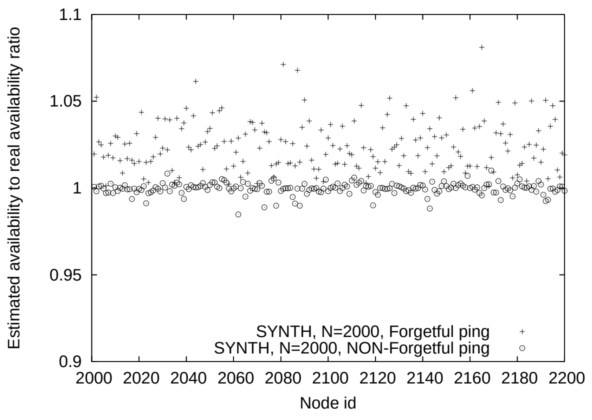

Figure 17: Ratio of estimated availability to actual availability, with and without the forgetful pinging optimization.

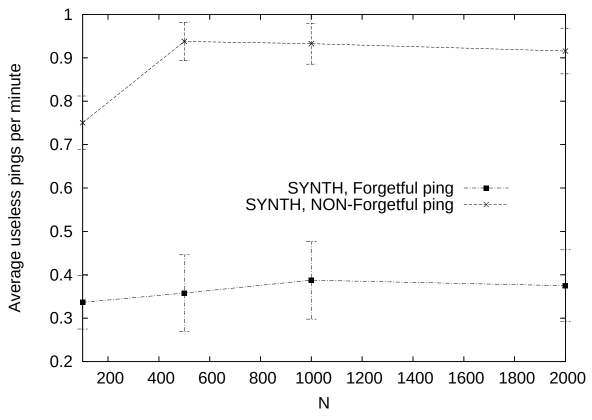

Figure 18: Forgetful pinging reduces useless pings sent to absent nodes. Bars show 1 standard deviation.

plots the ratio, for each node in the control group, of its estimated availability to its real availability (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%. In turn, Figure 18 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.

Bandwidth: Figure 19 shows the CDF for the *Outgoing Bytes per Second* (BW) that nodes incurred in the `STAT` and `OV` models. First, in `STAT`, with $N = 2000$ total nodes, 88% of the nodes had an outgoing bandwidth below $10Bps$. Notice that about 6.5% of nodes had a BW above $50Bps$. We surmise that this was due to indegree degradation owing to the static nature of the `STAT` model. To address this, we introduced an optimization (called "PR2") whereby a node that had not received a monitoring ping for two successive

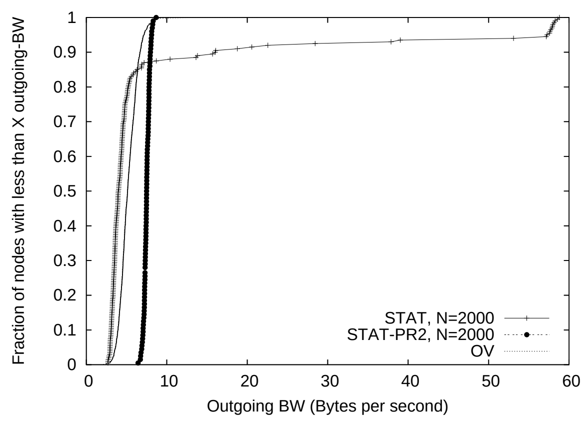

Figure 19: CDF of per-node outgoing bandwidth for the churned `STAT` model, and the churned `OV` model.

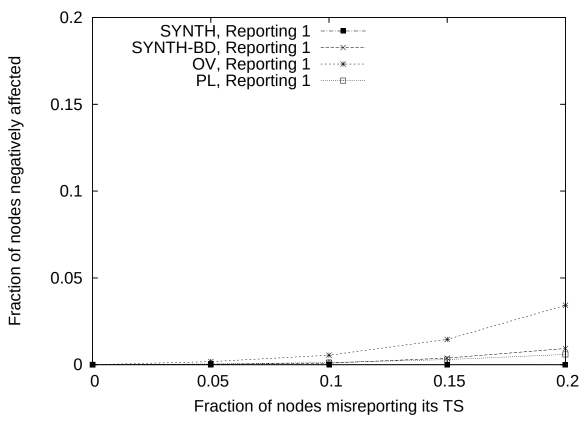

Figure 20: Fraction of nodes with above 0.2 error in measured availability, with some nodes overreporting all $TS(.)$ nodes' availabilities as 100%.

protocol periods would force all its coarse view nodes to add it to their own coarse views. Figure 19 shows that this variant (labeled "STAT-PR2") reduced bandwidth further for the `STAT` model - in fact, this caused all nodes to have an outgoing bandwidth below $9Bps$. In contrast to this, notice that in the Overnet churn traces (`OV`), there was constant change in the coarse views due to node birth, death, join, and leave, and consequently, the outgoing bandwidth was more uniform, with 99.85% of the nodes spending below $11Bps$.

Effect of Overreporting Attack: Finally, Figure 20 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`. In the worst-case, only 3.5% nodes were affected.

6 Conclusions

This paper has presented and evaluated AVMON, a system that selects and discovers an availability monitoring overlay for long-term host-level availability information in distributed systems. AVMON uses a hash-based consistency condition for selection of per-node availability monitors in a manner that is consistent, verifiable, and random. 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. Three variants of the discovery protocol were then derived to minimize different combinations of memory, bandwidth, discovery time, and computation time. The AVMON variant minimizing all metrics (Optimal-MDC) was found to perform well in a variety of churn models and traces - these included three synthetic models (static, join-leave, and join-leave-birth-death), and availability traces from PlanetLab and the Overnet p2p system.

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*, 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 Transactions on Computers*, 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] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *HotNets-I*, 2002.

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

[12] M. Raab and A. Steger. Balls into bins - a simple and tight analysis. *LNCS, Proc. 2nd Intnl. WRATCS*, 1518:159–170, 1998.

[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*, 2001.

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

[15] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, pages 149–160, 2001.

[16] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Proc. Middleware '98*, pages 55–70. Springer, 1998.

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

[18] Global Grid Forum. http://www.gridforum.org/.