Reliable On-Demand Management Operations for Large-scale Distributed Applications*

Jin Liang, Indranil Gupta and Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
{jinliang, indy, klara}@cs.uiuc.edu

ABSTRACT

This paper argues for attention to, and proposes a novel direction to solving, *instant monitoring and management* tasks for large-scale distributed applications running across hundreds of hosts. We present the MON (Management Overlay Networks) approach[1], which uses a novel concept called *on-demand overlays*, in order to support instant commands such as queries and software pushes. On-demand overlays are built on-the-fly and probabilistically, by leveraging weakly-consistent gossip-style membership information underneath. Thus, they are lightweight in terms of memory, computation, and bandwidth. We augment on-demand overlays with several notions of application-specified reliability, and show how MON detects and adheres to these. MON is available atop PlanetLab, and we present experimental results. We conclude with a series of promising open problems in this direction.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Communication.Distributed Systems

Keywords

Monitoring, Instant Commands, On-demand Overlays, Reliability

1. INTRODUCTION

Several wide-area and large-scale distributed computing systems have emerged in the recent few years, e.g., utility Grids [7, 30], experimental Grids [1], and lately PlanetLab [21]. More importantly, these large scale distributed infrastructures have become increasingly popular for running distributed applications such as content distribution [2, 5, 18, 34], application-level DNS [19, 22], cooperative caches [8, 13], publish-subscribe systems [15], and large-scale experiments, e.g., [32, 33].

While today many management tools are available for the management of computing infrastructures themselves, e.g., [7, 14, 26, 35, 37, 38] and they are very useful to the infrastructure's system administrators, there is a scarcity of significant tools that application developers and managers can use for *managing their applications* on such systems [10, 20]. Cluster-management tools allow querying of resource variables associated directly with the infrastructure, and have limited applicability to distributed application management. Intuitively, the latter is more of an *end-to-end management task* while cluster management orthogonally deals with the underlying cluster.

Distributed application management deals with several application "instances" (processes, not necessarily replicas) running on multiple hosts that communicate with each other. On each host, the application may create and modify certain *objects* - these include files (e.g., log files created by a distributed experiment), variables representing system resources used by and relevant to the application on the host (e.g., CPU utilization, RAM space free, disk space free, bandwidth usage in past 5 minutes). In addition, there may be certain system-wide properties that the application developer may want to enforce, e.g., a guarantee that at any time, at least 60 unique hosts are running instances of the application. Hence, the distributed application management must allow for querying these objects and system-wide properties, as well as for manipulating them.

Such monitoring tasks are an important component of distributed applications management, pointed out as one of the grand challenges for the next decade by CRA (Computing Research Association), as well as being the focus of much work in IBM, HP, Google, etc. [3]. Management of end-user applications routinely forms 24% to 33% of the TCO (Total Cost of Ownership) of today's distributed infrastructures such as clusters [27]. This cost for applications is increasing dramatically as the clusters are becoming geographically distributed and new distributed infrastructures are being created. Further, with the migration of services out from the traditional Internet and into such distributed infrastructures [17], application monitoring has become an important problem.

This paper considers a class of monitoring tasks that has been much-ignored so far. In any of the above distributed

*This research was supported in part by NSF CAREER grant CNS-0448246 and in part by NSF ITR grant CMS-0427089.

[1] This paper is an extended version of our workshop paper [12], and includes additional contributions on augmenting on-demand overlays with reliability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

applications, often the application developer wants to *instantaneously query* the instances ("nodes") of the distributed application. This allows an application deployer to query, at run-time, a variety of system-wide statistics and information, which are functions of the node-level objects discussed earlier in this section. For instance, a user experimenting with a peer-to-peer (p2p) multimedia streaming system [31] may wish to query the *current* system-wide average of node CPU utilization, or the current system-wide histogram of delay between neighboring nodes in the overlay. This information can be useful for assessing the effectiveness of different partner selection policies, and for ensuring continued good performance of streaming.

Another instance of an instant query is an operator of a p2p DNS system [22] querying the current system-wide replication degree for a group of domain names - this could be useful for adjusting replication levels of DNS entries at run-time, or to diagnose problems with individual domains. Overall, such an ability to instantly monitor and diagnose application-related information can help users understand and tune distributed applications better at run-time, quicken the response to service outages such as in e-business [11] and Internet services, as well as enable data center managers (and their customers) to better understand the application performance and return on investment (ROI).

MON Approach: We present MON (*Management Overlay Networks*), a system that allows such instant queries to be executed for large-scale distributed applications. MON's basic philosophy is to build *control-plane management overlays*, as depicted in Figure 1. In doing so, MON addresses the following four concerns:

• **Instant Queries via On-demand Overlays:** MON executes instant queries at run-time by building *on-demand overlays*, with one overlay built per query (by default). The monitoring queries we consider can be executed by constructing tree and DAG-based overlays. This scales extremely well in systems where the injection rate of commands is low, allowing MON to incur a very low overhead. MON nodes maintain weakly-consistent membership information only, thus avoiding using persistent overlays such as distributed hash tables [23, 24] or unstructured resource discovery systems [36]. Persistent overlays may be too heavyweight for management tasks, since they involve maintenance of invariants and highly-optimized membership management.

• **Scale and Frequent Failures:** MON seeks to scale to applications with several 100's to 1000's of nodes, as in data centers and distributed clusters. On-demand queries complete within a few seconds. The likelihood of a node failure during that time interval is small; thus, if it occurs, the command can be re-executed. However, if the overlay is reused for other commands, we need to address the issue of reliability, as we do next.

• **Overlay Reuse and Reliability:** We explore the spectrum between on-demand overlays and persistent overlays by considering notions of reliability for the built on-demand overlay. This would enable the overlay to be used for continuous execution of a command, as well as for other instant commands on that application. In order to address reliability, we discuss how the MON system can support *application-specified constraints* on both: (1) *session reliability*, which determines the reliability of an on-demand overlay as it is maintained for more than one command, as well as (2) *task reliability*, which determines the reliability of a single command on the on-demand overlay. In addition, we discuss how MON overlays can be built to improve their *coverage and performance*.

• **Simplicity of Management:** MON is built on the fundamental assumption that *management tools should be simple*, and less complicated than the applications on which they are used. This stems from concerns about "software bloat" pointed to by Tanenbaum in his SOSP 05 keynote [25], and the "complexity barrier" pointed to by CRA [3].

MON is currently running on more than 300 nodes on the PlanetLab, and we pepper our discussion with our experimental results. We discuss the basic MON system in Section 2, and analyze its feasibility vs. the persistent approach in Section 3. We extend MON to address reliability in Section 4, and conclude with promising directions in Section 5.

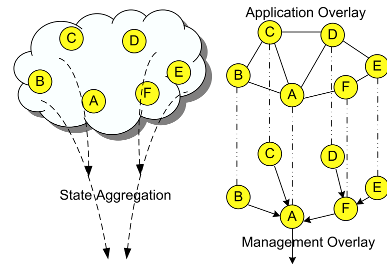

Figure 1: Control plane management overlays. Note D and F are not directly connected in the application overlay, but they are in the management overlay.

Distributed Command Execution
Overlay Construction
Membership Management

Figure 2: MON Architecture.

2. BASIC MON SYSTEM

Figure 2 shows the three basic MON components. Below, we first describe the management commands input by the application, then go bottom-up by describing membership, and finally the overlay construction.

2.1 MON Management Commands

MON views a distributed application as a database table, with each row representing the *current status* at one node. Such a table cannot be maintained at a central location, but has to be necessarily queried on the fly, somewhat akin to the use of TinyDB in sensor networks [16]. The attributes for a tuple in the database include the node's resource characteristics or log files, all maintained at the node itself. We view this data model as a starting step to extending MON, in the future, to historical data at each node, as well as a SQL-like query syntax (Section 5). MON currently supports the following operations:

1. select avg(<resource>) [where <condition>]
2. select top k <resource> [where <condition>]
3. select histo(<resource>) [where <condition>]
4. select <resource_list> [where <condition>]
5. select grep(<keyword>, <file>) [where <condition>]
6. select run(<shell_command>) [where <condition>]
7. count and depth: number of nodes in, and tree-depth of, on-demand overlay
8. push <file>

In the above commands, resource could be any node-level object: this could be a metric that is either host-based, process-level, or application-defined. Host-based and process-level metrics include CPU utilization, RAM, disk, etc. Examples of application-defined metrics include average delay to overlay neighbors (useful for a streaming application), or the number of DNS entries matching a particular domain (c.f., DNS application of Section 1). Further, condition can be any boolean expression over these node-level objects. For example, one could obtain the list of all nodes which store DNS entries for a given domain, or which have CPU utilizations above 50%. Finally, the select grep command allows the query of distributed log files for certain patterns, and the select run command allows the execution of arbitrary shell commands on many nodes simultaneously[2].

2.2 Gossip-Based Membership Management

Many p2p systems [23, 24] build overlays by maintaining certain invariant rules, with membership management tuned to converge to ensuring these invariants in spite of nodes joining and leaving the system. We eschew such approaches in favor of a weakly-consistent overlay approach where each node maintains soft state. This allows us to avoid maintaining up-to-date global membership information.

For this purpose, we use a gossip-style protocol, similar to [28, 29], for lightweight distributed membership management. Specifically, each MON node maintains a partial list of m other nodes currently in the system, called a *partial view*. A partial view has a fixed size, and may consist of some random entries as well as some close-by nodes (in the network). Well-known results [6] show that in a system of N nodes, $m = \Omega(\log(N))$ random entries at each node suffice to ensure connectivity in the overlay with high probability.

Partial views are updated periodically in a lazy manner - periodically (once every round, where round length is fixed at all nodes), a node picks a random target from its partial view, and exchanges some random membership entries with the target. To quickly remove failed nodes and maintain the freshness of membership entries, we associate an *age* with each entry, which estimates the time since a message was last received from the corresponding node. For example, whenever node B receives a gossip message from node A, it sets the age of A to 0. Later, when B gossips A's information to C, it includes the age of A, which is the time since the entry was created. Instead of timing out entries, new entries are continuously shuffled in (1 per period), with the oldest entry being dropped.

MON would work well even if the above membership protocol is replaced with an alternative such as SCAMP [6], Cyclon [29], T-MAN [9], or SWIM [4].

[2] For security reasons, we allow only certain IP addresses to initiate this command.

2.3 On-demand overlay construction

To execute instant management commands, we build two kinds of overlay structures: *trees* and *directed acyclic graphs (DAGs)*. A tree structure is suited for instant status query, and a DAG is suited for software push. Most MON operations use UDP, except for software push, where we use TCP. Since an overlay is created on-demand, we would like the construction algorithm to be quick and efficient, involving minimum startup delay and message overhead. These basic overlays have only probabilistic coverage of nodes in the system, i.e., they may exclude some nodes. Later, Section 4 describes how reliability can be specified and implemented within our basic MON overlays.

Tree Construction: We describe two tree construction algorithms - *random tree construction* and a locality-aware *two-stage construction*. For *random tree construction*, a MON node initiates this on-demand overlay construction by sending a Session message to k other randomly selected nodes from its view (membership list). k is called the fanout of the overlay and is specified inside the Session message. Each node receiving a Session message for the first time will respond with a SessionOK message, and becomes a child of the Session sender. It also randomly picks k nodes from its partial view, and sends the Session message to these nodes, just like the initiator. If a node receives a Session message for a second time, it responds with a Prune message.

The random tree construction algorithm is simple and will have good coverage (with fanout $k = \Theta(\log(N))$ [6]). However, it is not locality-aware to the underlying network. This motivates our second algorithm, called *two-stage construction*, which attempts to improve the locality of the tree, while still achieving high coverage. Session messages carry the number of hops transited, which is set to 0 at initiator, and incremented at every hop. A first session message received with hop number $\leq$ threshold h participates in the first stage of the protocol, and it will select children from among its random view elements. A first session message received with hop number $> h$ will participate in the second stage, selecting children from among its nearby-neighbors in the underlying network. For this stage, if not enough view elements exist for selection, random view entries are used.

Intuitively, the two stage algorithm "seeds" different parts of the network with the session message in the first stage, and covers these individual localities in the second stage. If h is limited to $o(\log(N))$, then the first stage contains a sub-linear (in N) number of messages, while most of the messages are sent over low-latency high-bandwidth paths in the underlying network, in the second stage. Finally, notice that though the view may contain stale entries (since it is weakly-consistent), even 50% stale entries imply that one would need to try an expected $(2 \cdot k)$ random entries for the first stage, and $O(\log(N))$ entries for the second stage.

Table 1 compares the performance of the random tree (with different fanout values) vs. an optimized two-stage approach (with $k = 5$) on a PlanetLab slice with 330 nodes. The numbers, averaged over 200 overlays we created, show that the two-stage approach achieves 97% coverage with a latency of a few seconds. The latency CDF of two-stage, used for the count query, is shown in Figure 3 - this is below 1500ms in 65% cases, and below 2000ms in 97% cases.

DAG construction: The above tree construction algorithms can be modified to create DAGs. Specifically, each node is assigned a *level* l, which is similar to the hop number

Table 1: Tree construction performance.

	rand5	rand6	rand8	twostage
coverage	314.89	318.64	320.52	321.59
creation time(ms)	3027.21	3035.46	2972.46	2792.03
`count` time(ms)	1539.19	1512.07	1369.92	1354.79

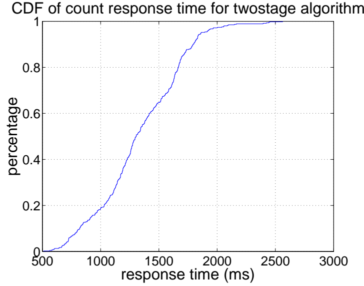

Figure 3: CDF of `count` time for twostage.

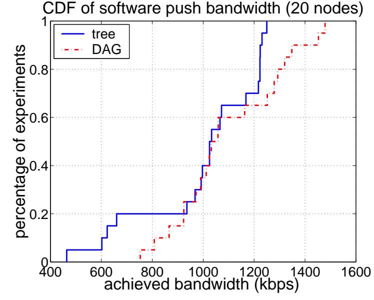

Figure 4: Software push bandwidth of MON.

in the tree messages described for the two-stage algorithm. The initiator (root) node is at level 1, and each other node has a level of 1 more than that of its first parent node. When a node with set level l receives a second `Session` message, it can accept the sender as an additional parent, as long as its level is smaller than l. This ensures the resulting overlay contains no loop, and is thus a DAG.

Figure 4 shows that the DAG approach (in a system of 20 nodes) achieves a total bandwidth of 22 MBps, which is *7 times the largest local bandwidth offered by any individual node*. The bandwidth stays between 900 kbps and 1.3 Mpbs, with an average of 1.1 Mbps. In comparison, the tree overlay has a bandwidth that is 10% smaller.

3. FEASIBILITY ANALYSIS: ON-DEMAND VS. PERSISTENT OVERLAYS

In this section, we explore, for monitoring queries, the feasibility of the on-demand overlay approach in comparison to an approach that uses persistent overlay maintenance techniques. Concretely, we use a back-of-the-envelope analysis that takes into account the background bandwidth and the per-query communication overhead, in order to calculate the query rates for which the on-demand approach consumes less bandwidth than the persistent approach. Our mathematical analysis only considers the random tree construction version of MON presented in the previous section, and thus it also bounds the performance for other intelligent approaches such as twostage. In addition, we will plug in real implementation numbers from MON into our analysis at the end of this section.

Consider a persistent overlay where each node spends a background bandwidth of B Bps to maintain *up-to-date* neighbor information. Up-to-date neighbor information implies none of the membership entries at the node are stale, and thus a query tree of a given fanout k can be constructed by directly selecting k children from the neighbor list (according to an arbitrary selection criterion). Let the cost of doing so be C Bytes - notice that a higher value of k implies a proportionally linear increase in the value of C. Further, note that we are counting only the tree construction cost here, since once the tree is constructed, the cost for executing the actual query remains the same regardless of the method used for tree construction.

Now, let the query rate input to the system be q per sec. Then, the bandwidth cost of such a persistent overlay approach for monitoring is:

$$B_{pers} = B + q \cdot C \quad (1)$$

On the other hand, consider an on-demand approach that maintains an overlay in a weakly-consistent fashion, so that only a fraction $\frac{1}{m}$th of the bandwidth B is spent on maintenance of membership information. With a background bandwidth cost of $\frac{B}{m}$ Bps, only a fraction $\frac{1}{m}$ of the neighbor entries at a given node will be up-to-date. When building the tree for a given query, the random tree construction selects each child by randomly probing entries in the neighbor list until an alive node is encountered. Combining this assumption and the reduced bandwidth, we conclude that a node needs to contact an expected m entries before it can find an alive child. Since this has to be carried out for each of the k children, the per-query cost becomes $(m \cdot C)$ Bytes[3]. Thus, given a query rate of q per sec, the bandwidth cost of the on-demand overlay approach for monitoring is:

$$B_{od} = \frac{B}{m} + q \cdot m \cdot C \quad (2)$$

From equations (1) and (2), the on-demand overlay is more feasible than the persistent overlay when:

$$\begin{aligned} B_{od} &< B_{pers} \\ \frac{B}{m} + q \cdot m \cdot C &< B + q \cdot C \\ q &< \frac{B \cdot (m-1)}{m} \cdot \frac{1}{C(m-1)} \end{aligned}$$

This gives us the following feasible range for on-demand overlays:

$$q < \frac{B}{mC} \quad (3)$$

Feasible Query Rates for our MON Implementation: In our MON implementation, the size of each gossip message 100 B, and a node sends one such message every 10 sec. Thus, the background bandwidth consumed by MON is $B = \frac{100}{10}$ Bps $=$ 10 Bps. The cost of each message sent to a prospective child (for the tree construction only) is 50 B,

[3] Although our analysis appears to ignore latency, our per-query bandwidth C accounts for it implicitly and numerically: due to the retrying in recruiting children, a higher per-query bandwidth implies a proportionally linear increase in latency.

and the value of fanout $k = 4$. Thus, the per-query cost at each node is the cost of sending messages and receiving aknowledgments from each of these children. This is $C = 2 \cdot k \times 50\text{B} = 400$ B.

Plugging in the above numbers into equation (3), we can derive the threshold query rate at which the on-demand approach consumes less bandwidth than the persistent approach, by setting $m = 1.01$ (note we should have $m > 1$ in the derivation). This is:

$$q < \frac{1}{40.4s}$$

Thus, if the application injects monitoring queries at a rate lower than around once every 40 seconds on average, then MON is more feasible than the persistent overlay approach. We believe this is a realistic operation range for MON since user-injected queries are associated with think-times of at least several minutes, since it takes at least that much time for the user to either inspect the results or to formulate a new query. On the other hand, for automated and continuous monitoring, if the above query rate limit is insufficient, then there are two options: (1) either increase the background bandwidth cost B in order to support a higher query rate, or (2) use the techniques in the next Section 4 in order to reuse the on-demand overlays and increase their reliability.

4. RELIABLE ON-DEMAND OVERLAYS

In order to explore the spectrum between on-demand and persistent overlays, as well as to improve the reliability of MON, this section describes design decisions that extend the concept of on-demand overlays to *medium-term overlays*. While an on-demand overlay was used only on a *short-term* basis, i.e., for a single command, a medium-term overlay survives for several hundreds of seconds and can be used for multiple commands and continuous monitoring. This is especially useful in extending MON to work at query rates higher than those calculated in the previous section.

However, we wish to do the above without spending any explicit maintenance bandwidth for the on-demand overlay that has been constructed. This motivates us to imbue MON with certain notions of *reliability* that are provided to the application. MON provides applications two co-existing flavors of reliability: *session* reliability and *task* reliability. Session reliability applies to the overlay itself (which can be used to execute multiple management commands), while task reliability latter applies to individual management commands. Each flavor of reliability is specified as a deterministic bound by the application, e.g., on the number of missing nodes in the overlay, or in the aggregated result. We describe below how MON detects violation of these bounds, retries commands if necessary, and how effective this is in PlanetLab.

4.1 Session Reliability

We define *session reliability* of an on-demand overlay, at any given time, as the probability that fewer than a number max_drop of nodes are disconnected at that time. The value of max_drop is specified by the application or user at overlay creation time, e.g., when the first of a series of commands is injected. This type of reliability is especially a concern for tree overlays, where failures and lost messages will cause the number of nodes in the overlay to decrease over time. Thus, given a value of max_drop, the tree overlays of Section 2 have short lifetimes.

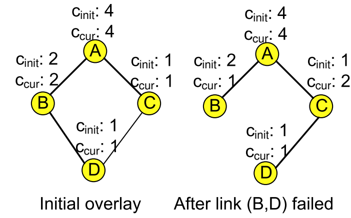

Figure 5: Reliability violation detection for DAG overlays. c_{init} and c_{cur} are variables maintained at each node.

In addressing this, and in keeping with our goal of spending no maintenance overhead, MON does *not* attempt to repair the tree by detecting failures of nodes. Instead, we focus on: (1) building robust on-demand overlays with good session reliability; and (2) *detection* of session reliability violations (i.e., when more than max_drop nodes are disconnected), so that the end-user or application can be *notified*. This allows the end-user (or application) to decide if a new overlay should be constructed.

To improve session reliability, even though a tree is used for aggregation and the command execution, we build a *shell DAG* initially, and the tree neighbors are chosen from this shell DAG. The shell DAG is built with a fanout $k' > k$, the tree's fanout. Each child only sends its aggregate data to the "primary parent," i.e., the first parent from which the overlay creation message was received. For other parents, the child sends an empty data message, which simply informs the parents that the child has finished command execution. The advantages of using the shell DAG are twofold: (a) once constructed, it does not need to be maintained, (b) it can be used by a node to switch to an alternative parent, should that node's primary parent fail.

We briefly describe detection of reliability violation for trees first, and then for DAGs. For the tree, each node maintains two variables: c_{init} and c_{cur}, which respectively refer to the initial and current number of nodes in the subtree rooted at the node. c_{init} is reported to the parent at overlay construction time, and it is not changed thereafter. c_{cur} is reported to the parent in the refresh ack message, thus it is continuously aggregated. Whenever a node detects that $c_{init} - c_{cur} > max_drop$, it declares a reliability violation and sends a notification to the initiator node for the overlay. For this, the root node's address is carried with the on-demand overlay creation message, and is remembered by nodes. For DAG overlays, the scheme works similarly, except that to avoid duplicate counting, each node only reports these values to its primary parent. Notice that c_{init} is initialized only once at overlay construction time.

Figure 5 shows a DAG overlay. Initially node D reports its c_{init} and c_{cur} to its primary parent B. Later, if link (B, D) fails, B will update its c_{cur} to 1, but its c_{init} is not changed, because it means the "initial" number of nodes included. After the link failure, node D reports its c_{cur} (which is 1) to the new primary parent C, but it still reports $c_{init} = 0$ to C. Thus, the root node obtains the correct c_{init} and c_{cur}.

One issue may arise with the above scheme for DAGs. If a parent failed soon after the c_{init} value was sent to it (and before it was implicitly acknowledged to the child via a message), then the child has no way of knowing whether

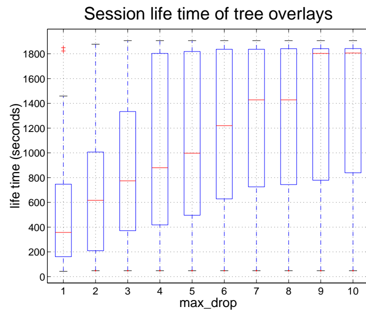

Figure 6: Session Reliability: Lifetime distribution plateaus with max_drop.

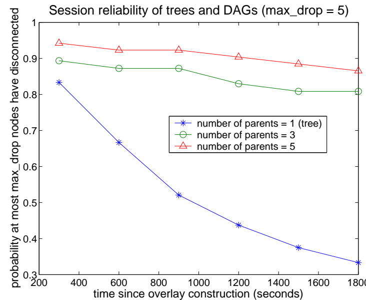

Figure 7: Session Reliability since overlay creation.

to report a value of c_{init} to its other new (secondary) parent or not. We take a conservative approach to solving this problem, having the child node always report a value of $c_{init} = 0$ to its new parent. At the root node, if the value of c_{cur} has been stable for several refresh periods and is greater than c_{init}, we set c_{init} to c_{cur}.

PlanetLab Experiments: Figure 6 shows the effect of different max_drop values on a PlanetLab slice of 325 nodes. The figure shows the min, max, 25th percentile, median and 75th percentile life time for different max_drop values. This plot shows that using $max_drop = 5$ suffices to obtain a median overlay lifetime of 1000 s (almost 17 minutes). Figure 7 then shows that at $max_drop = 5$, selecting 5 parents in the shell DAG suffices to provide 85% session reliability up to 1800 seconds after overlay creation.

4.2 Task Reliability

We define task reliability of a management monitoring command (i.e., a query) as the probability that at most $max_missing$ nodes are missing from the aggregate result. The value of $max_missing$ may be application- or user-specified at either command initiation time or overlay creation time (to apply to all commands run on that overlay). This reliability notion captures the incompleteness of data returned by a command execution, which may occur due to failures during the command.

MON detects a task reliability violation as follows. Each node keeps a number $num_missing$ for each query, which means the number of nodes whose results are missing from the current data. The $num_missing$ is aggregated toward the root along with the command data. If a node finds that $num_missing > max_missing$, then this is reported to the initiator node. Additionally, the child may be asked to *retry* the command. As a result, a node may be able to receive some data from nodes that are previously ignored. However, after several unsuccessful retries (implementation value=4), MON declares a failed query, sending any queried data to the initiator node anyway.

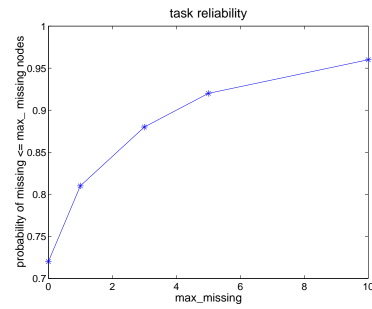

Figure 8: Task Reliability for different $max_missing$.

PlanetLab Experiments: Figure 8 shows the reliability of a monitoring query, measured over 2000 queries in a 325 node PlanetLab slice. First, notice that the query completes perfectly only 72% of the time (see $max_missing = 0$). However, if the application is willing to tolerate up to 10 missing values (see $max_missing = 10$), then the query can be completed 96% of the time.

5. SUMMARY AND LOOKING FORWARD

On-demand overlays built on top of weakly-consistent membership information can be used to execute monitoring and software push commands quickly (within seconds), scalably (in systems with 100's of nodes), and reliably (application-specified reliability parameters). This is achieved by our MON system, and we validate our claims via experiments from a PlanetLab deployment (see our web interface at: http://cairo.cs.uiuc.edu/projects/mon/).

Our work raises several interesting directions that would be fruitful to explore. Some of the issues we are looking at currently include (but are not restricted to) the following:

A. Persistent vs. On-demand: By imbuing MON membership information with a few deterministic invariants (e.g., virtual ring as in Chord), it is possible to improve the reliability of on-demand overlays. The challenge is to avoid high overhead - this appears feasible since invariants are not mandatory, but only improve coverage and latency.

B. Expressive Queries: Instant queries, continuous queries, and triggers, can all be specified via an expressive and complex SQL-like language. This raises the issues of creating multiple on-demand overlays, and tradeoffs such as overlay abandonment versus overhauling.

C. Leveraging the Application Overlay: A distributed application (e.g., a CDN or an experiment) spans its own overlay. Instead of an application using the MON solely in a black-box manner as above, by providing upcalls into the application, MON may be able to utilize (1) the peer and neighbor information from the application overlay, and (2) piggyback MON information onto application messages. This would lead to "zero-overhead" membership maintenance, especially for high-traffic applications.

D. Leveraging Overlapping PlanetLab slices: If overlapping PlanetLab slices create on-demand overlays at a high rate, then partial, cached, and stale membership in-

formation, as well as overlay information (e.g., parents and children) can be reused from one slice to another. This can reduce latency and overhead, the tradeoff of cache staleness versus bandwidth will need to be addressed.

6. REFERENCES

[1] F. Cappello and et al. Grid'5000: A large scale, reconfigurable, controlable and monitorable Grid platform. In *Proc. GRID*, 2005.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *SOSP'03*, 2003.

[3] CRA. Grand Research Challenges in Distributed Systems. http://www.cra.org/reports/gc.systems.pdf.

[4] 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.

[5] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing Content Publication with Coral. In *Proc. Usenix/ACM NSDI*, 2004.

[6] A. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, Feb. 2003.

[7] IBM. The Oceano Project. http://www.research.ibm.com/oceanoproject/.

[8] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

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

[10] F. Kaashoek and et al. Report of the NSF workshop on research challenges in distributed computer systems. http://www.nsf.gov/cise/cns/geni/workshop_report.pdf.

[11] T. Kuegler. The Billion-Dollar Question: The Impact of Web Site Performance on E-Commerce.

[12] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. Mon: On-demand overlays for distributed system management. In *Proc. Usenix WORLDS*, 2005.

[13] P. Linga, I. Gupta, and K. Birman. A churn-resistant peer-to-peer web caching system. In *Proc. 1st ACM Workshop Self-Survivable and Regenerative Systems*, Oct. 2003.

[14] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proc. 8th ICDCS*, pages 104–111, 1988.

[15] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client Behavior and Feed Characteristics of RSS, A Publish-Subscribe System for Web Micronews. In *Proc. Internet Measurement Conference (IMC)*, 2005.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. *ACM TODS*, 2005.

[17] NSF. The GENI initiative. http://www.nsf.gov/cise/geni/.

[18] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *WORLDS'04*, December 2004.

[19] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: improving DNS performance and reliability via cooperative lookups. In *Proc. 6th Usenix OSDI*, 2004.

[20] D. Patterson. A conversation with Jim Gray. *ACM Queue*, 1(4), June 2003.

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

[22] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *Proc. ACM SIGCOMM*, pages 331–342, 2004.

[23] 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.

[24] 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.

[25] A. Tanenbaum. Keynote address. ACM Symposium on Operating Systems Principles (SOSP), 2005.

[26] A. S. Tanenbaum and S. Mullender. An overview of the amoeba distributed operating system. *Operating Systems Review*, 15(3):51–64, July 1981.

[27] TechWise Research Inc. Are some RISC-based clusters easier to manage than others? http://h71000.www7.hp.com/openvms/whitepapers/sm_whitepaper.pdf, 2004.

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

[29] 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.

[30] T. Weiss. Grid computing gets push from Sun, IBM and Compaq. *Computer World*, Nov. 2001.

[31] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. DONet: A Data-Driven Overlay Network for Efficient Live Media Streaming. In *IEEE INFOCOM'05*, Miami, FL, 2005.

[32] Intel Outlines Strategy For Making The Internet Smarter, Safer, More Reliable And Useful. http://www.intel.com/pressroom/archive/releases/20040909corp.htm, Sep. 2004.

[33] HP Joins PlanetLab as Major Research and Technology Backer. http://www.hp.com/hpinfo/newsroom/press/2003/030624a.html, Jun. 2004.

[34] CoDeeN content distribution network. http://codeen.cs.princeton.edu/.

[35] Cooperative Association for Internet Data Analysis. http://www.caida.org.

[36] The Gnutella protocol specification. http://www9.limewire.com/.

[37] The Berkeley NOW Project. http://now.cs.berkeley.edu/.

[38] The Globus Alliance. http://www.globus.org/.