The One Minute Manager: Lightweight On-Demand Overlays for Distributed Application Management

Jin Liang, Indranil Gupta and Klara Nahrstedt
Department of Computer Science, University of Illinois at Urbana-Champaign

Abstract
The emergence of large-scale distributed computing clusters such as PlanetLab and Utility Grids has fueled the development of applications ranging from content distribution to name service to large-scale prototype experiments. However, the management of such applications when they are deployed in a real world, wide area environment remains a challenging problem. In this paper, we present MON (Management Overlay Networks), a simple, scalable and lightweight system for distributed application management. At the most basic level, MON builds short-lived, on-demand overlays that can be used to execute management commands such as status query and control. To further address the coverage, reliability and performance issues of on-demand overlays, we exploit techniques such as incremental overlay construction, overlay adjustment and opportunistic DAG (directed acyclic graph) based aggregation, which greatly improve the practicality of on-demand overlays. Our extensive experiments on the PlanetLab show that for a large group of more than 300 nodes, on-demand overlays can be built to (1) cover more than 95% of the nodes (2) last for tens of minutes even without failure repairs; and (3) achieve an end-to-end response time of just a couple of seconds. Further, we demonstrate the utility of MON by showing how it can be used to query the aggregate state of a real application (Pastry) deployed in a real world environment.

1 Introduction
In the last several years, research efforts on large scale peer-to-peer systems [11, 26, 28, 30, 32] and the deployment of distributed computing infrastructures such as PlanetLab [23] and the Grid [10] have enabled the development of many large scale applications, such as content distribution [1], name service [22, 25], storage [8, 29], publish-subscribe [24], and web caching [17]. Such applications often consist of a large number of application nodes that collaborate in a peer-to-peer fashion in order to provide a service. As a result, they can improve the perceived service quality, e.g., by re-directing clients to nearby service nodes, by routing data around network faults, or by sharing load among the service nodes.

Despite the successful design of many large scale applications, there has been little work that address the management of these applications when they are deployed in a real world environment, either for initial test or for production runs. As a result, it is fairly common for application developers to spend much time just to get their applications running in a real world environment.

Management of end-user applications has routinely formed 24% to 33% of the total cost of ownership (TCO) [13] in today’s distributed infrastructures such as clusters. As applications continue to be deployed in a wider area environment, the cost for application management is likely to increase even further. Therefore, we believe it is imperative that we carefully study the capabilities needed for distributed application management, and to design management systems that provide the necessary capabilities.

Distributed Application Management Managing large scale distributed applications requires management capabilities that are different from traditional localized distributed systems. First, such applications often need to run in a wide area environment where there could be all kinds of network and computing node failures. For example, Table 1 shows some of failures that we have experienced on the PlanetLab. Since these failures are fairly common, and some of them (e.g., routing problems) may automatically recover, continuously monitoring the application nodes and alerting the human managers upon every failure may be unnecessary. Instead, the ability to dynamically query the application...
Table 1: Some Example Failure Modes on PlanetLab

<table>
<thead>
<tr>
<th>time</th>
<th>nodes</th>
<th>symptom</th>
</tr>
</thead>
<tbody>
<tr>
<td>06/2005</td>
<td>planetlab2.nbgisp.com</td>
<td>disk error caused the CoMon service to be inaccessible</td>
</tr>
<tr>
<td>08/2005</td>
<td>planetlab2.ucb-dsl.nodes.planet-lab.org</td>
<td>system out of memory caused &quot;sendto(): No buffer space available&quot;</td>
</tr>
<tr>
<td>09/2005</td>
<td>planetlab1.uc.edu and seu1.6planetlab.edu.cn</td>
<td>routing loop caused connectivity problem between the two nodes</td>
</tr>
<tr>
<td>03/2006</td>
<td>planetlab1.cs.uiuc.edu and dlut2.6planetlab.edu.cn</td>
<td>persistent high loss rate (76%) and large rtt (about 300ms)</td>
</tr>
<tr>
<td>03/2006</td>
<td>planetlab1.cs.uiuc.edu and planet2.njit.edu</td>
<td>very large persistent rtt (about 4 seconds) and large loss rate (14%)</td>
</tr>
<tr>
<td>03/2006</td>
<td>pli2-3-hpl.hp.com and planetslug3.cse.ucsc.edu</td>
<td>high loss rate (70%) but very small rtt (about 4ms)</td>
</tr>
</tbody>
</table>

status and take control actions may be more desirable.

Second, traditional distributed system management has focused on system level information such as CPU load and memory usage. For distributed application management, however, we are also (probably more) interested in application level state. For example, how many nodes have incorrect successor pointers in a DHT system; and what is the average (or top 10) parent-child delay in an application level multicast tree? Due to the large number of nodes in an application, and the huge amount of state information within a node, collecting complete information about the application state would incur too much overhead and still be uninformative. Therefore, we need a mechanism to aggregate the state information before it is presented to the application manager.

Finally, unlike system level management, where a small number of professional managers are responsible for managing large computing infrastructure, distributed applications are often managed by their respective developers/deployers. As a result, a simple, lightweight and easy-to-use system is needed.

Large distributed applications are often designed with certain degree of self-organization and failure resilience. However, it is important to distinguish between the management of an application from the application’s self-management property. The latter is necessary for an application to continue functioning when faced with failures. However, the former is concerned with if (and how well) the application is functioning. For example, detecting and replacing failure entries in a routing table belongs to self-management. However, counting the average number of failure entries in a routing table (and reporting it to the application manager) belongs to the management of an application.

One approach for distributed application management might be to build management capabilities into the application, so that it can be directly managed (queried and controlled). However, this would make the application (unnecessarily) more complicated, and hence more likely to exhibit emergent, abnormal behaviors that are unexpected by the application developers [20]. In addition, it does not facilitate the re-use of management capabilities across applications. Therefore, we believe a better choice is to design a simple management system that can be re-used across different applications.

Figure 1: Control Plane Management Overlays. Note nodes D and F are not directly connected in the application overlay, but they are in the management overlay.

The On-demand Approach

This paper presents the Management Overlay Networks (MON) system that is designed to provide the necessary capabilities for distributed application management. At a high level, MON facilitates the management of distributed applications by providing a mechanism to aggregate the distributed application state, and to control the application if necessary, as is shown in the left part of Figure 1. At a closer look, MON builds on-demand, control plane overlays for management command execution. As is shown in the right part of Figure 1, the management overlay spans all the application nodes. However, it is for management purpose and it is independent of the application overlay.

The management overlay is on-demand in that it is not maintained for a long time. Instead an overlay is constructed whenever one or more management commands (called a management session) need to be executed. The overlay is discarded as soon as the commands are finished.

This on-demand approach is motivated by the observation that overlays such as trees and DAGs (directed acyclic graphs) are well suited to distributed application management (e.g., state aggregation and command propagation). However, maintaining trees or DAGs for a long time when there could be simultaneous node and links
failures is difficult. On the other hand, Distributed hash tables (DHTs [28, 30]) have been built to have good failure resilience. However, layering management functions on top of DHTs may introduce additional overhead, especially if the management overlay is only needed for a short time.

In view of this, MON adopts the on-demand approach and builds overlays only when they are needed. No attempt is made to maintain the overlay for a long time. This enables MON to be simple and lightweight, because it does not need complex failure repairs mechanisms, and it incurs little overhead when no management commands are being executed.

We have described the basic idea of on-demand overlays and simple status query in a previous workshop paper [16]. In this paper, we further address important issues of on-demand overlays such as coverage, reliability and performance, and improve the usability of MON by providing high level programming support. Specifically, this paper makes the following contributions:

1. We design different techniques such as incremental overlay construction, overlay adjustment and opportunistic DAG based aggregation, which can be used to improve the coverage, reliability and performance of on-demand overlays.

2. We design a SQL-like language that allows users to dynamically query the aggregate state of an application (e.g., Top-K and Histogram), and to take control actions if necessary. We also provide a client side API so that users can integrate the management capability of MON into higher level programming logics.

3. We conduct extensive experiments on the PlanetLab to thoroughly evaluate our MON system. Our results show that on-demand overlays can be built to have good coverage, reliability and performance. In addition, we demonstrate the utility of MON by using it to query a real application (the Pastry DHT) that is deployed in a real world environment.

In the rest of the paper, we first present the basic MON Architecture in Section 2. We then describe the coverage, reliability and performance issues of on-demand overlays in Section 3. Section 4 discusses the management capability of MON and how to use it for distributed application management. Section 5 are the evaluation results. Finally, Section 6 discusses related work and Section 7 concludes the paper with some further discussion.

2 Basic MON Architecture

MON adopts an on-demand approach for distributed application management. This means whenever some management commands need to be executed, the user will first create an overlay on-demand, then execute the commands on the overlay. After the commands are finished, the overlay is removed.

To support on-demand overlay construction and command execution, MON deploys a daemon process (called a MON server) that runs side-by-side with each application node. Each MON server has a three layer architecture as shown in Figure 2. The bottom layer is responsible for membership management. The middle layer creates overlay on-demand, and the top layer executes the management commands on top of the overlays.

2.1 Gossip-Based Membership Management

When an overlay is constructed on-demand, the set of nodes that should be included are already up and running. This is in contrast to many existing peer-to-peer systems, where overlay construction is handled by individual peers joining and leaving the system. As a result, some kind of membership information is needed, so that we know which nodes should be included.

Due to the dynamics of large scale applications, maintaining up-to-date global membership information might involve too much overhead. Therefore, we use a gossip-style protocol [9, 14] for lightweight distributed membership management. Specifically, each MON server maintains a partial list of $m$ nodes currently in the system (called a partial view). Periodically, a node picks a random target from its partial view, and exchanges some membership information with the target.

Randomized membership exchange allows newly joined nodes (e.g., those recovered from a crash) to be integrated into the system. However, one problem with standard gossip protocol is that even if a node has failed, its information may still be gossiped around for a long time. To quickly remove such failure nodes and maintain the freshness of membership entries, we associate an age with each entry, which estimates the time since a message is last received from the corresponding node. For example, when node $B$ receives a gossip message from node $A$, it will create an entry for $A$ and set its age to 0. Later, when $B$ gossips $A$’s information to $C$, it will include the age of $A$, which is the time since the entry was
created. When the partial view is full and some entries need to be dropped, the oldest entries are dropped first.

2.2 On-demand Overlay Construction

The partial membership views maintained at each MON server effectively create a densely connected directed graph \( G \) among all the nodes. The nodes in the graph correspond to the MON servers, and the links correspond to the membership entries (i.e., a link \((A, B)\) exists in the graph if and only if node \(A\) has the membership information of \(B\)). To create an overlay among the nodes is equivalent to creating a spanning subgraph of the graph \( G \). Note, however, the graph \( G \) is inaccurate in the sense even if \(A\) has a link to \(B\), \(A\) may not directly contact \(B\) because the network link between the two nodes may be broken (i.e., \(A\) may have obtained \(B\)'s information from some other node).

For distributed application management, we consider the construction of overlay trees and directed acyclic graphs (DAGs). Trees are well suited to information aggregation, and DAGs can be used to improve performance and reliability of on-demand overlays (as we will show in Section 3).

Ideally one may want to create an overlay that includes all the current live nodes (i.e., has full coverage). However, since each node only has partial view of the system, deterministic coverage would involve too much message overhead, because one message must be sent over every link in the graph \( G \). As a result, we are content with probabilistic node coverage and focus on quick and efficient overlay construction algorithms. In practice, due to transient and permanent node failures, a user is often prepared if not all the desired nodes can be accessed.

Tree Construction The first algorithm we consider for on-demand tree construction is a simple randomized algorithm. The on-demand overlay construction is initiated by a client side software (called a MON client) that sends a Session message to a nearby MON server. Each node (MON server) that receives a Session message for the first time will respond with a SessionOK message and become a child of the Session sender. It will also randomly pick \(k\) nodes from its local partial view, and send the Session message to them. \(k\) is called the fanout of the tree overlay and is specified in the Session message. If a node receives a Session message for a second time, it will respond with a Prune message. It has been shown that assuming the partial views represent uniform sampling of the system, such tree construction will cover all the nodes with high probability, if \(k = \Omega(\log N)\), where \(N\) is the total number of nodes in the system [15]. Note a Session message (and hence the overlay) is identified by the initiator IP address and a unique number called a session ID. As a result, each MON server can participate in multiple sessions at the same time.

The random tree construction algorithm is simple and has good coverage (with sufficient fanout \(k\)). However, it is not locality aware so it may take a long time to execute a management command on such overlays. Therefore we have designed a second algorithm called two stage, which attempts to improve the locality of a tree, while still achieve high coverage. To do this, the membership layer of each node is augmented with a local list in addition to the partial view, which consists of nodes that are within certain delay threshold \(d\) to the local node. Each node is also assigned a random node id, and the local list is divided into left and right neighbors (those with smaller and larger node ids). The tree construction is divided into two stages. During the first several hops, each node selects its children randomly from the partial view, just like the random algorithm. The goal is to quickly spread the Session message to different areas of the network. In the second stage, each node first selects nodes from its local list, then from the partial view if not enough local neighbors are present. To avoid the case where a small cluster of nodes mutually select each other as children, equal number of children are selected from the left and right neighbors.

**DAG Construction** The above tree construction algorithms can be modified to create DAGs. Specifically, each node is assigned a level \(l\) that is propagated in the Session messages. The level of the initiator (MON client) is set to 0. The level of a each MON server is set to 1 plus the level of its first parent. Suppose a node has set its level to \(l\) and a second Session message is received, 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, thus a DAG.

2.3 Overlay Maintenance and Command Execution

Once a management overlay is constructed, it can be used to execute one or more management commands (called a management session). During a session, MON uses a simple heartbeat mechanisms for failure detection. This means each parent periodically sends a Refresh message to each child, and expects to receive an RefreshOK from the child. If no RefreshOK is received from a child for \(t\) consecutive periods, the parent will assume a failure has occurred, and will remove the child from its children list. Similarly, each child expects to receive a Refresh message from its parents periodically. If no Refresh is received from a parent for \(t\) (called a life_count) consecutive periods, it will assume a failure
has been detected and will remove the parent. If this is
the last parent, the child will assume the overlay is no
longer needed, and remove all state associated with the
session. The Refresh period is in the order of several
seconds. The goal is to detect non-transient failures that
last for tens of seconds or more. The number \( t \) is called
the life.\( \text{count} \) for the parent/child and it is specified by
the user when the overlay is constructed.

When a management session is finished, the MON
client can send a Stop message to explicitly remove the
overlay. However, since the session state is soft, even
if the client terminates before sending a Stop message,
the entire overlay will timeout and terminate soon (Note
even if the MON client sends an Stop message, it may
not reliably reach every node. As a result, the soft state
approach is necessary to guarantee eventual termination
of all garbage state).

To execute a management command (called a management task), the MON client sends the command to its MON server. Upon receipt of a command, each MON server will first send an ack message back. Next the command is propagated to the children nodes. Finally the command is executed locally. When the command results from all children are received, the MON server will aggregate them with local execution result, and send the aggregated result to the parent.

Both the command and the result messages may be
lost. As a result, each MON server will keep re-
transmitting the command message to each child until
data from the child is received. The retransmission fre-
cuency is determined by the delay between the parent
and the child (as measured by the periodic Refresh
messages), but is limited to some maximum frequency.
The goal is to ensure quick response time despite mes-
sage losses, but not to impose too much network traffic.

When a child receives a retransmitted command, it
will send either an ack or the command result, depend-
ing on if it has had the aggregate data ready. If a parent
fails to receive ack or result message for \( t \) consecutive periods, it will ignore the data from the child and return
whatever is available to its parent.

3 Coverage, Reliability and Performance

Although the basic MON architecture is extremely sim-
ple, it does not address several important issues about
on-demand overlays, which we will address in this sec-
tion:

- **Coverage**: Since we use randomized algorithms for
overlay construction, how can we ensure that as many nodes as possible are included in the overlay?
- **Reliability**: Since we use simple heartbeat mech-
anism for overlay maintenance and command execu-
tion, how can we improve the life time of an on-
demand overlay, and how can we improve the comple-
teness of the command result?

- **Performance**: Even for our twostage algorithm,
there will be some random overlay links with bad
performance (e.g., large delays). How can we im-
prove the performance of the overlay constructed
using simple algorithms?

3.1 Coverage

The coverage of an on-demand overlay is defined as the
number of live nodes that are included in the constructed
overlay. To improve the coverage of a management over-
lay, we use a simple technique called incremental over-
lay construction. This means the user can first build an
overlay using the simple algorithms described in Sec-
tion 2.2. If not enough nodes are covered, the user can
invoke some special command to cover more nodes. The
special command provided by MON is called Recruit.
When the user issues the command, it will be propagated
to each node currently in the overlay. Suppose a node
receives the command and it currently has \( nc \) children.
If \( nc \) is smaller than the fanout \( k \) (as specified when the
overlay is constructed), the node will select \( k - nc \) ran-
don nodes from its membership list, and send a Ses-
session message to them, similar to the initial overlay con-
struction.

To allow the user to have more control on how
the overlay is incrementally constructed, the Recruit
command can take a scope parameter, which means the
MON servers should only try to recruit new nodes that
are within certain delay to themselves (i.e. a MON server
may try to recruit less than \( k - nc \) new nodes if the scope
requirement cannot be satisfied). The delay information
can be obtained during membership gossips.

3.2 Reliability

The reliability of on-demand overlays is an important is-
sue from the application’s viewpoint. In our MON sys-
tem, we consider two flavors of reliabilities: session and
task reliabilities. Session reliability applies to the over-
lay itself (which can be used to execute multiple man-
gagement commands), while task reliability applies to an
individual management command.

**Session Reliability** With our simple heartbeat based
overlay maintenance, the number of nodes included in
the overlay is likely to decrease over time. As a re-
sult, we define session reliability as the probability that
an on-demand overlay can be used for certain time pe-
riod before a specified number \( (\text{max_drop}) \) nodes are
disconnected from the overlay. In practice, we can use the expected life time of an on-demand overlay (before \(max\_drop\) nodes are disconnected) to represent its reliability. The longer an overlay can be used, the more reliable it is. Here \(max\_drop\) is a parameter specified by the user before the overlay is constructed.

Our MON system makes no attempt for failure repair, nonetheless, we can provide some minimum assurance so that users will have confidence in using the system. Specifically, (1) we provide mechanisms for the user to improve the reliability of on-demand overlays; and (2) we automatically detect session reliability violations (i.e., when more than \(max\_drop\) nodes are disconnected) and notify the user about it. The user can then decide whether to continue using the overlay, or to build a new one instead.

To improve the session reliability of on-demand overlays, we build DAG overlays where each node can have at most \(k'\) parents. \(k'\) is called the fanin number and is also specified by the user. To execute a command on a DAG, each parent node still sends the command message to each child. However, each child will only send its aggregate data to its “primary parent” (e.g., the first parent from which a \textit{Session} message is received). For other parents, the child will send an empty data message. The empty data message simply informs the parent that the child has finished command execution and the parent should stop command retransmission. Effectively, we are still using a tree structure embedded in the DAG for command execution. However, if the primary parent of a node fails, it can switch to a new primary parent. As a result, the overlay can still be used despite failures.

The \textit{Recruit} message, originally designed to improve the coverage of an overlay, can also improve its reliability. This is because a node can recruit not only new nodes, but also some existing nodes as children. If an existing node is recruited, the coverage is not improved, but the redundancy (and hence reliability) of the overlay is improved.

Detecting session reliability violations turns out to be a difficult problem. Below we first look at how reliability violations are detected in tree overlays, then we will look at DAG overlays.

Each node in a tree overlay maintains two variables: \textit{init\_count} and \textit{cur\_count}. For each tree node, which refers to the initial and current number of nodes in the subtree rooted at the node. The \textit{init\_count} is reported to the parent at overlay construction time, and it is not changed later. The \textit{cur\_count} is reported to the parent in each \textit{Refresh\_OK} message. In other words, the \textit{cur\_count} value is continuously aggregated. Whenever a node detects that the \textit{init\_count} – \textit{cur\_count} is greater than \(max\_drop\), it directly sends a notification message to the user (the session initiator).

For DAG overlays, each node still maintains the \textit{init\_count} and \textit{cur\_count}. To avoid duplicate counting, each node will only report these values to its primary parent, and as in the tree case, \textit{init\_count} is reported to the parent only at overlay construction time, and \textit{cur\_count} is continuously aggregated. For example, Figure 3 shows a DAG overlay. Initially node \(D\) reports its \textit{init\_count} and \textit{cur\_count} to its primary parent \(B\). Later, if link \((B, D)\) fails, \(D\) will report its \textit{cur\_count} (which is 1) to the new primary parent \(C\), but it still reports an \textit{init\_count} of 0 to \(C\). As a result, the root node can have the correct \textit{init\_count} and \textit{cur\_count} (Note in the DAG case, only the root can detect reliability violations, due to the fact that a child disconnected from one parent may still have other parents).

Above we assume the \textit{init\_count} of a node can be reliably communicated to its primary parent. In reality, this is not true. Suppose during the overlay construction phase, node \(D\) receives a \textit{Session} message from \(B\) and accepts it as a primary parent. \(D\) sends its \textit{init\_count} back to \(B\). However, suppose \(B\) crashes shortly after the \textit{Session} message was sent. \(D\) will not be able to know if \(B\) has received the \textit{init\_count} (and passed it on to its parents) or not. In the tree case, this is not a problem since \(B\) is the only parent. If it crashes, \(D\) is also disconnected. In the DAG case, however, \(D\) may have other parents so it may still be connected despite the crash of \(B\). If \(B\) crashes before it receives the \textit{init\_count} from \(D\), \(D\) should report its \textit{init\_count} to the new primary parent. Otherwise the DAG root will have an \textit{init\_count} that is smaller than \textit{cur\_count}. On the other hand, if \(B\) has received the \textit{init\_count}, \(D\) should not report it again, otherwise it will be counted twice.

Without knowing for sure if there is a loss of \textit{init\_count}, we use a simple heuristic to address the problem. A node will report its \textit{init\_count} to the primary parent only once. However, if the DAG root finds that its \textit{cur\_count} has not changed for several refresh periods and it is greater than the \textit{init\_count}, it will assume there has been a loss of \textit{init\_count} and will set the \textit{init\_count} to its \textit{cur\_count}.

If a \textit{Recruit} command is executed, the \textit{init\_count}
of nodes can actually increase. However, each node will report the new init_count only once, and only to their current primary parent.

**Task Reliability** When we execute a management command (task), the aggregate execution result may contain incomplete data due to temporary failures (timeouts). We define task reliability as the probability that at most max_missing nodes are missing from the aggregate result. max_missing is also specified by the user. Since incomplete command result is caused by temporary failures, we attempt to not only detect, but also recover from the failures.

Specifically, each node keeps a number num_missing for command execution, which means the number of nodes whose result are missing from the current data. The num_missing is aggregated toward the root as the command result are aggregated. If a node finds that num_missing is greater than max_missing, it will require its children with num_missing > 0 to retry the command. The children nodes will ask their children nodes (those with num_missing > 0) to retry. As a result, a node may be able to receive some data from nodes that are previously ignored (see the command execution discussion in Section 2.3). If, however, after retry several times (MON current retries at most 4 times), there are still too many missing nodes, the data will be sent to the user. But a warning flag is also set, which informs the user of the task reliability violation.

### 3.3 Performance

Since our management tasks are mainly status query and control, we consider the end-to-end response time of the overlay as its main performance metric. This means when a command is issued by the user, how long does it take for the results to be sent back. Since our overlay construction algorithm cannot guarantee delay bound on the overlay links, there may be high delay links that affect the end-to-end response time. To improve the performance of the overlay, we exploit the idea of overlay adjustment. The general idea is that an overlay is first constructed using the simple algorithms. Thereafter, the performance of the overlay is gradually improved by introducing overlay links with small delays and removing those with large delays.

Physically removing an overlay links may reduce the session reliability. Therefore, in our MON system, we do not remove existing overlay links. Instead, we exploit the redundancy of DAG overlays and dynamically select the best overlay links for “opportunistic DAG based aggregation”.

Specifically, suppose a DAG overlay has been constructed. During the overlay maintenance process, each MON server can measure the delay to each child based on the Refresh and RefreshOK messages. The parent will communicate this delay to the child (by piggyback it to the next Refresh message), so that the child knows its delay to each parent. Each time when a MON server receives a management command, it dynamically selects one parent as the primary parent. This means it will only send the aggregate data to this parent. For other parents, it will immediately send an empty data message. It is clear that the more parents a node has, the more likely it is to find a good primary parent. The Recruit command as described above can be used to improve the redundancy of DAG overlays, thus improve the performance of the overlay.

Dynamically selecting the best overlay links is only one form of overlay adjustment. Other forms (although we have not implemented) include Disconnect and Attach. Disconnect means the root propagates a message down the overlay. Each node that has more than one parent should disconnect from the “worst” parent (the one with the largest delay), if the delay to this parent is larger than some threshold. This will actually remove bad overlay links and free children slot of the parent. Attach means the root propagates a message down to every node. Each node whose delay to the primary parent is greater than some threshold can attempt to connect to some nearby node as child. This allows nodes to actively locate good parent, instead of waiting to be recruited. Note that none of the three operations change the level of a node. As a result, the overlay is guaranteed to remain a DAG despite the adjustment. Allowing nodes to change their level would give us more freedom in overlay adjustment. However, it also means additional mechanisms such as loop detection is needed.

### 4 Application Management with MON

In this section, we describe the management capabilities of MON. We describe (1) the SQL-like language we have implemented for status query and control; (2) a client-side API that can be integrated into higher level programming logic; and (3) how to use these capabilities for application management.

#### 4.1 MON Query Language

We have implemented a SQL-like language that allows users to query the aggregate and non-aggregate status of the distributed applications, and to control the application status. We note that there exist many systems such as PIER [12], SWORD [21] and Astrolabe [31] that allow
the query of distributed systems using different (SQL-like) languages. These languages are often richer in semantics and can be potentially implemented via on-demand overlays. However, the discussion of this is beyond the scope of this paper.

The general language syntax for aggregate queries looks like the following:

```c
select agg(<resource>)
[where <condition>]
[maxmissing <num>]
```

Here `agg` is the aggregation function. We currently support three kinds of aggregation functions: AVG, TOP-K and HISTOGRAM. `resource` is the metrics that we want to query. It can be simple metrics such as CPU load and free memory. It can also be complex metrics that have parameters. For example `filesize("mon3.log")` queries the size of the file with the name "mon3.log", and `procmem("mon3")` queries the memory usage of the process "mon3". `condition` is a boolean expression over different resources (we have implemented only the conjunctive normal form (CNF) boolean expressions. However, it is known any boolean expression can be transformed to the CNF form). A command is locally executed on a node only if the `condition` evaluates to true.

For example

```c
select avg(freemem) where Load > 10
```

will return the average amount of free memory for those nodes with a CPU load greater than 10. Finally, `num` is the task reliability requirement on a command.

Non-aggregate queries generally look like the following:

```c
select <resource_list>
where <condition>
[maxmissing <num>]
```

Here `resource_list` is a list of one or more resources. The command should return the specified resource values on the nodes that satisfy the `condition`. Note the `where` clause is mandatory for non-aggregate queries. This is meant to remind the user that non-aggregate queries may return too much data. Therefore the user should provide a `where` clause to limit the amount of data returned.

The third category of commands is for status control. Right now we have provided the capability to execute any shell command on all the nodes. The general syntax is like the following:

```c
select run(cmd)
[where <condition>]
[maxmissing <num>]
```

It means the shell command `cmd` should be executed on any nodes that satisfy the `condition`. To facilitate the execution of common shell commands, we also implemented some higher level commands such as

```c
select grep(keyword, file)
[where <condition>]
[maxmissing <num>]
```

It will try to search the specified keyword in the specified file, and return the first line of match 1.

### 4.2 MON API and Scripts

Our MON client provides a command line interface for users to interactively query and control their applications. To fully explore the power of MON, we have provided a client side C++ API so that MON can be integrated into higher level programming languages for automated application management. The API consists of two simple function calls:

1. `mon_init()`: this initiates the appropriate data structures.
2. `mon_exec(char*cmd, MonResult* result)`: this executes a command (in the syntax described before) and wait for the results. We have used this API for all our experiments in this paper. The API can be easily integrated with some extensible scripting language such as Python, so that users can write high level scripts. For instance, the following script periodically queries the average CPU load on a set of nodes, and take some additional actions if the average load is greater than some threshold.

```c
while(1) {
    create_session();
    avg_load = mon_exec("select avg(load)")
    if(avg_load > 10){
        hosts = mon_exec("select top 10 load");
        //do something else
    }
    stop_session();
    //sleep some time
}
```

### 4.3 Building MON into the Distributed Application

While the basic MON commands and MON query and scripting languages can be used to externally query an application’s behavior, one also desires to be able to query the internal state of the distributed application at run-time. For instance, given a DHT application running on PlanetLab, one may wish to keep track of properties of the DHT routing tables maintained throughout the system, or find when error conditions were generated by different DHT nodes.

We briefly describe two different approaches for building MON into the distributed application in order to address the above problem. The first approach requires MON to be in-built into the application, while the second approach requires application nodes to create (local) log files and then uses MON to query the log file entries in a distributed manner.
Internal Status Query:  Distributed applications often have large amount of internal state that may be interesting to the application developer. Without MON, application managers can only infer the internal state of an application from its external behaviors. With MON, however, application managers can directly query the internal state of an application. To do this, the application needs to be instrumented in order to make its internal state available to MON.

The easiest way to query the internal state of an application is to link the application with MON. Thus, MON is deployed as part of the application itself. When the application is started, it provides a callback function to MON. Whenever MON receives a command to query the the application state, it calls the callback function to obtain the information. After MON obtains the internal state, it aggregates the state on the management overlay.

Alternatively, the application can provide a query interface, e.g., by creating a separate thread that listens on a UDP socket and waits for queries from MON. Whenever MON needs to query the application state, it queries the application by sending the message to the specified UDP port. We have used this approach for querying the FreePastry [3] system, which is written in Java.

Distributed Log Query:  The above instrumentation approach may be difficult to build into legacy applications. However, if the individual nodes running the distributed application each generate a local log file, where coarse events such as errors or finer events such as message receipts are logged, the MON infrastructure described so far can be used to directly query these log files, without any changes to the application itself. For instance, a simple grep command can be used to find out if a particular error has occurred in the log file of some application nodes. More sophisticated commands can return the number of occurrences in each log file, the most recent recent occurrence in each file, or some other relevant information.

If the application generates fine-grained information in the log files, e.g., a DHT application where each node periodically dumps its routing table entries into the log, then this approach can be used to study the evolution of such fine-grained application characteristics as time progresses. The only limitation of this approach is the amount of information that an application outputs to its log files. However, to output more information into the log files is fairly easy, and can be achieved without even changing the application (e.g., increase the log level at command line). Thus we believe log query will be an important capability for distributed application management.

5 Evaluation

The MON system has been implemented in C/C++ and deployed on PlanetLab. This section presents experimental results from our PlanetLab deployment. For most experiments, we use set both fanout and fanin to be 5. The overlay maintenance interval (refresh period) is 5 seconds. Using a simple MON query on its own bandwidth usage shows that on average, the maintenance overhead for one session is about 0.1KBps.

5.1 Comparison with Persistent Overlays

To compare the on-demand overlay approach with the persistent overlay approach, we choose to compare the performance of aggregation along two types of trees - the MON trees (which are on-demand) and the trees constructed by the Scribe system [6], using the implementation available with FreePastry [3]. Scribe is built on top of the Pastry DHT. Scribe is a publish-subscribe system, but for the purpose of our experiments, it is a tree-building service.

We deployed Scribe and MON on the same set of about 120 PlanetLab nodes. We then built 20 trees using Scribe and MON. For each tree, we executed 20 simple count queries and computed the average response time. The implementation of the count query in Scribe is exactly the same as in MON. Figure 4 and Figure 5 respec-
tively show the response times for the persistent Scribe tree and the on-demand MON tree. The Scribe response time is between 2 and 4 seconds, while the MON response is between 700ms and 1400ms.

There could be several reasons for the performance difference. First, the Scribe implementation that we use does not limit the fanout of a node, thus the root node is often overloaded by many children (50 or more). Second, Scribe uses TCP connections between parent and children, which may retransmit lost packets less aggressively. Third, Scribe is implemented in Java, which is arguably inefficient compared to MON’s C/C++ implementation. While any of the above reasons could be addressed via a different implementation, our experiments nonetheless show that using existing DHTs for tree construction without additional optimization may lead to performance that is several times worse than simple on-demand overlays.

5.2 Coverage
To measure the coverage of MON’s incremental construction of on-demand overlays, we ran an experiment on 325 PlanetLab nodes. Each time, we created a session, and queried the count for 40 times. After every 10 queries, we executed one Recruit command. The scope of the recruit was set to 20ms. We set max_missing = 100 in order to examine the response time without being affected by command retries. We repeated the above for 200 sessions, and computed the “average trajectory” of the experiment.

Figure 6 shows that when an overlay was initially created, only about 303 nodes were covered, and each node had about 2.1 parents. After one recruit (tree number=10), about 310 nodes were covered (over 95% of the deployment), and each node had about 2.3 parents. The marginal utility (in terms of coverage and parent redundancy) of additional recruits clearly decreased. Although more nodes were covered, the average number of missing nodes and response time for each query did not vary much.

While we were able to achieve a 100% coverage for networks of 120 nodes, Figure 6 shows that achieving 100% coverage is difficult for 300+ sized groups. This is primarily because some nodes are too “far away” from other nodes. Even if a Session message was sent to these nodes, their SessionOK would come back too late. This can be addressed by waiting longer for SessionOK messages. However, this comes at the expense of longer session construction and recruit time.

5.3 Reliability
This subsection investigates the effect of the two reliability mechanisms described in Section 3, i.e., session reliability and task reliability.

First, we study the session reliability. Figure 7 shows the session reliability of trees on 325 nodes. For this experiment, 50 trees were built and for each tree, we recorded the time until the i-th node is disconnected from the overlay. This allowed us to compute the session lifetime for different max_drop values. Figure 7 is a box-plot that shows the min, max, 25-th percentile, median and 75-th percentile life time for different max_drop. When max_drop is small, many trees would have a small lifetime. This is because the on-demand overlays may contain random overlay links that are long or lossy. Such links are likely to timeout and disconnect rapidly. However, when max_drop increases, more sessions will have longer lifetime, because the disconnection of a small number of nodes does not constitute a reliability violation. For example, if we can tolerate 5 nodes being disconnected from the initial overlay, then more than half of the time we will be able to use the overlay for about 1000 seconds, which is enough for quite some queries. Note for this experiment, if the lifetime of a session exceeds half an hour (1800 seconds), we will remove the session in order to reduce the experiment time.

Figure 8 shows the average session life time for trees
The average session lifetime for DAGs is much larger than trees, because nodes in a DAG may have multiple parents. Thus, the disconnection of one overlay link may or may not cause a node to be disconnected. However, the difference between $fanin = 3$ and $fanin = 5$ is less significant.

Next, we study the task reliability of MON. Figure 9 shows the response time of a simple query (select count) on PlanetLab. For this experiment we executed the simple query using different $max\_missing$ values (on 325 nodes). The experiment lasted for several hours and we executed 2000 queries for each $max\_missing$ value. Both the mean and standard deviation are shown. Observe that when $max\_missing > 1$, the average response decreases due to higher tolerance to incomplete results, and hence fewer command retries. However, the response time for $max\_missing = 0$ is actually smaller than for $max\_missing = 1$. This can be explained as follows. If $max\_missing = 0$, whenever a node fails to receive command result from a child node, it knows the task reliability is violated. Thus it immediately initiates command retry. However, if $max\_missing = 1$ and a node fails to receive data from only a single node (i.e., only one node is missing from the result), it will propagate the result to the parent. However, because another part of the overlay may also one single missing node, the parent (or some higher level ancestor) may find that the total number of missing node is larger than 1. As a result, the command retry is initiated from a higher level node, thus taking a longer time.

Figure 10 shows the cumulative distribution function (CDF) of the response time for different $max\_missing$. Observe that the CDF for $max\_missing = 0$ is different from other $max\_missing$ values. Specifically, the response time for $max\_missing = 0$ goes up earlier, because there are more command retries. But the growth is more slowly, because most retry takes less time (since they are initiated closer to the failure point).

Figure 11 shows the probability of having incomplete data for different $max\_missing$ values. We can see if $max\_missing = 10$, we will have incomplete data about 51% of the time. However, using $max\_missing = 0$ can recover from 23% of the temporary failures and reduce the probability to 28%. Figure 12 shows the average number of missing nodes. We can see when $max\_missing$ is large, there are fewer retries in MON. As a result, more missing nodes are expected in each command result.

Overall, having $max\_missing = 0$ seems effective at reducing the number of missing nodes in a command.
execution. However, it may incur larger response time more often. Therefore, if a small number of missing nodes is tolerable, then it is better to set \( \text{max}_\text{missing} \) to relatively large value (e.g., 10) in order to achieve fast response.

### 5.4 MON Query of Pastry’s Internal State

Although previous work [27] has studied the performance of DHTs under churn, those studies were simulation-based. MON allows us to study the performance in a real world environment. We instrumented the Scribe/Pastry system as described in Section 4.3, deployed it on 100 PlanetLab nodes, and used MON to query the state of Pastry routing table under churn. Two metrics of Pastry were queried. The first is the average “proximity” of the first and second Pastry rows. Pastry uses proximity neighbor selection algorithms to fill its routing table with nodes that are nearby. The first row is the least constrained, thus the average proximity is expected to be small. The second row is more constrained, therefore its proximity is expected to be large. The second metric queried is the number of live entries in the routing table. It is likely that during churn, the routing tables may contain fewer entries than under stable conditions.

Figure 13 shows the average proximity (i.e., round-trip time) of Pastry routing table rows (across all nodes) with time. We began the query when most nodes just joined the system. We can see when nodes first join the system the average proximity is large, because nodes have not found nearby nodes for routing entries. However, after several minutes, the proximity of the first row quickly decreases to about 60. After about 10 minutes, it further decreases to below 40. The proximity of the second row also decreases from more than 400 to about 100. At time 1500 seconds, we began to introduce churn by restarting 2 randomly chosen Pastry nodes every 30 seconds (the bootstrap node is never restarted, so that the overall Pastry network is in one ring). We can see shortly after the churn begins, the average proximity of both rows began to increase. Although Pastry attempts to reduce the proximity, the overall level is still higher than stable case. At time 2400, we increased the churn rate by restarting 4 randomly chosen nodes every 30 seconds. We can see average proximity of the first row increased to about 100, and the average proximity of the second row increased to about 300.

Figure 14 shows how the average number of live routing table entries varies during this experiment. We can see the average number of routing table entries is close to 21 before the churn, but decreases to about 20 under churn, and about 19 under a higher churn rate. This shows that Pastry can quickly detect and replace failed routing table entries.

### 5.5 MON Log Querying

In this experiment, we examine MON’s log-querying capabilities. For this experiment, we created a log file on each of about 320 machines. The file sizes ranged from about 400KB to about 5MB, with an average of just over 2MB. Only six of these log files contain the word “Fail”. These six machines had the following delay (rtt) to our
Figure 15: CDF of the number of “interesting” nodes found by log query

local node: 0.3ms, 19ms, 27ms, 35ms, 59ms, 62ms (we call these as the “interesting nodes”). We used MON to execute the grep command to find out on which nodes the keyword “Fail” appeared in the log file. We executed such 1500 queries.

Figure 15 shows the CDF of the number of interesting nodes discovered. We can see about 10% of the time we discovered all 6 interesting nodes, about 40% of the time we discovered at least 5, and about 89% of the time we discovered at least 4. The average number discovered is 4.43. The average execution time of these log queries was just over 2 seconds, however, the average number of missing nodes is about 22. This shows that on the one hand, executing complex queries such as log query may require different timeout mechanisms than simple queries. On the other hand, for log queries, we do not need to have complete execution result in order to discover interesting nodes.

6 Related Work

Most existing systems [2, 4, 5, 7] for distributed system management are based on a centralized architecture. In such architecture a management agent is deployed on each device to be managed. These agents can report the status of the device to a central manager node, and control the device based on commands from the manager node. Such systems generally work fairly well for small scale, localized distributed system, but may be insufficient for widely distributed environment due to scalability reasons. For example, the current CoMon [2] PlanetLab monitoring system is configured to report the status of the PlanetLab nodes every five minutes. This may not be enough for distributed application management, where instant status query and control may be needed.

There are also distributed status monitoring systems such as Ganglia [19], Astrolabe [31]. Ganglia uses a hierarchical tree structure to monitor the status of federated clusters. To handle node failures, each interior node is manually configured with several children nodes. The assumption is that each child node is equally capable of reporting the status of a leaf cluster. Astrolabe also uses a tree structure for status aggregation in a large system. To be robust to node failures, Astrolabe uses gossip protocols to update information at different nodes. This, however, also means that it may take a long time to achieve eventual consistency.

Much previous work has realized the importance of querying the state of a large distributed system, be it a sensor network [18] or the Internet [12]. However, TAG [18] utilizes the routing algorithms in ad hoc networks to build the aggregation tree, while PIER [12] is built on top a DHT. The former is unavailable for distributed application management, while the latter introduces overhead that our MON system tries reduce.

Overlay maintenance in face of node failures has been a difficult problem. Distributed hashtables (DHTs [28, 30]) have been designed as a common infrastructure substrate for large scale distributed applications. Although DHTs generally have very good failure resilience, they nonetheless introduces additional overhead. Our MON system has explored an alternative approach to address failures. If the overlays do not need to exist for a long time, they can be constructed on-demand. This may not be suited for many applications. However, we have shown for distributed application management, on-demand approach can result in simple and lightweight management systems.

7 Conclusion and Discussion

In this paper, we have presented MON (Management Overlay Networks), a system we have designed for distributed application management. MON facilitates the management of distributed applications by building control plane overlays for instant status query and control. Such overlays are independent of the application overlay, and require little or no modification to the application being managed. The design of MON exploits different ideas such as gossip based membership management, on-demand and incremental overlay construction, overlay adjustment, opportunistic DAG based aggregation, etc. As a result, MON is not only simple and lightweight, but shows good performance in terms of coverage, tunable reliability, and end-to-end response time. Our extensive experiments on PlanetLab have demonstrated both the performance and utility of MON.

The focus of our work in this paper is on the performance of MON. This paper does not deal with security mechanisms in MON. However, simple mechanisms based on public-private key systems can be used to achieve some basic security in MON. Specifically, each MON server can be equipped with the public key of the application manager. Each management command is-
sued by the application manager is associated with a certificate that certifies the initiator, session ID and a timestamp of the command. Each MON server will execute the command only after it has verified the legitimacy and recentness of the certificate. As a result, only authorized parties can execute query and control commands on an application. To further improve security, all communication among pairs of MON servers can be encrypted, however, this may entail additional overhead.

Acknowledgements

We are grateful to Animesh Nandi and Jeff Hoye for their help on FreePastry and for suggesting the kinds of Scribe/Pastry metrics that we can query.

References

[29] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In ACM SOSP 2001 (October 2001).

Notes

1 Other operators, such as returning the first K lines of match, the last K lines of match, or random K lines of match can also be implemented.