Towards Self-Management in Distributed Application Management System

Chongnan Gao, Hongliang Yu, Weimin Zheng
gaochongnan@gmail.com, {hlyu, zwm-dcs}@tsinghua.edu.cn
Computer Science and Technology Department, Tsinghua University, Beijing 10084, China

KEYWORDS

Self-management, distributed system, epidemic algorithm

ABSTRACT

Distributed application management system is important for managing applications on distributed computing platforms. One of the main caveat of using a distributed management system is that the management system itself need to be deployed and maintained continually. In this papar, we propose Self-Managed Overlay Network (SMOM) and explore the challenges associated with designing a management system with self-management capability. SMON manages itself using epidemic approach at runtime. SMON can automatically deploys itself on a set of machines and recovers failed peers *securely*. It can also upgrade itself to new versions online. Through mathematical analysis and evaluation on PlanetLab platform, we show that SMON achieves good performance and scalability.

I. INTRODUCTION

Distributed systems [1], [2] are at the heart of today's Internet services. While the cost of commodity computers continues to drop, it is common for distributed computing platforms to contain hundreds or thousands of computers, such as Planet-Lab [3], Amazon EC2 [4] and Teragrid [5]. These platforms provide huge amount of storage and computing capabilities that can boost performance of distributed applications significantly. However, it is still difficult to design, deploy and manage distributed applications at large scale and fully utilize the resources.

Managing a distributed application faces several challenges. The application must be first deployed to a set of machines. The deployment should have good scalability so that it can work at large scale. During the deployment, failures—such as machine or network failures—are inevitable. They must be handled at any time during file transfer and application installation. Only when the application is deployed and configured successfully can the execution begin. After the application is started, certain mechanisms are need to monitor and recover application from failures. Depending on the application's semantic, the recovery may be simply restarting the failed processes. For some cases, failures may lead to aborting and restarting the whole application on many machines. For all the applications, quick failure detection and recovery are required to ensure that applications can run successfully. Distributed applications are also commonly upgraded for bug fixes and performance improvement.

Distributed application management system is designed to ease the burdens of managing applications on many machines. It automates many aspects in deploying and maintaining applications. By hiding the underlying details and providing a user-friendly interface, distributed applications can be deployed, monitored, recovered and upgraded automatically given instructions from developers. The developers can focus on bug-fixing and performance tuning of their applications, rather than managing the applications on a set of distributed resources.

To manage distributed applications at large scale, the management systems have to take distributed designs, or to say, they are also distributed applications. A management system either adopts centralized (i.e. client/server-like) or peer-to-peer (P2P) approach. Generally, a distributed management system consists of many "peers"[1] on each machine where applications will be deployed. The peers form an overlay network, and they work corporately to deploy applications to a set of machines. After applications are started, each peer monitors and maintains applications' processes within the same machine box.

There is one important caveat regarding the use of an application management system: the management system itself have to be deployed first and maintained continually throughout its lifecycle. Intuitively, we can use an existing management system M' to deploy and maintain a new management system M. But this approach doesn't *solve* but just *migrate* the problem: M' still faces the same problem recursively. Currently, developers usually resort to writing scripts—which leverage OS services such as sshd on Unix/Linux—to maintain management systems centrally. This approach is very preliminary and has several disadvantages. First, it is not scalable, especially at transferring large amount of data or managing distribute systems on many machines. Second, it cannot handle failures well. When network partition happens, it loses control of machines in different partitions. Third, the management functionality is limited by underlying OS services (e.g. sshd) at some extent.

In this paper, we argue that a distributed management system should have self-management capability and not rely on external tools to manage itself. We implement the idea

[1]The word peer here will refer to client if the management system is in client/sever architecture.

as Self-Managed Overlay Network (SMON). SMON is a distributed management system with built-in self-management capability, namely, self-deployment, self-recovery and self-upgrade. User of SMON has little work to do on maintaining it. A SMON peer monitors and maintains other ones running on different machines. It will automatically deploy new peers on fresh machines, recover failed peers, and upgrade itself to newer versions. The collective behavior of all peers gives rise to a management system that can deploy, monitor, recover and upgrade itself automatically. SMON can also manage a set of applications.

In designing SMON there are several challenges.

The first challenge is that SMON should have good scalability. Second, to enable automatic deployment and recovery, it need security mechanism support so that peers can login into other machines without manual assistant. And access to SMON should be authenticated and encrypted to avoid misuse of the system for malicious activities. The Third challenge is that SMON should define reasonable application management semantic.

SMON's design addresses the challenges as follows. For the first challenge, SMON peers use epidemic algorithm to monitor and maintain each other. Periodically, a peer will choose a random one from its membership list and monitor its running status. It will deploy new SMON peer on a fresh machine or recover failed peer. The simultaneous deployment or recovery for the same peer from multiple peers is idempotent, which means only one deployment or recovery is actually performed. By exchanging version number with remote peers, a peer will upgrade itself to the latest version. This epidemic approach ensures good scalability ($O(\log N)$) when system goes large. While self-management capacity keeps all peers running as long as possible, it needs mechanism to be disabled so that SMON can be stopped. A global boolean variable `livetag` is used to indicate enabling or disabling status of self-management capacity. The `livetag` variable is also maintained consistently using epidemic algorithm by all peers.

For the second challenge, we employ a separate authentication agent to assist peers to log into remote machines automatically. When a peer needs to log into a machine, it redirects the authentication challenge from remote machine to the agent. The agent stores the credential used to authenticate with the remote machine. It will resolve the challenge on behalf of the peer. The peer replies the response from the agent back to the machine and login successfully. The credential will be never leaked out of the agent. The authentication load on the agent is light and a single agent can support a number of peers. We use a symmetric key to authenticate and encrypt access to SMON peers and the authentication agent.

For the third challenge, SMON defines its management semantic for long running internet services. The "long running internet services" shares these management requirements: a) peers of the application can be deployed and started independently without synchronization, b) the failure of a single peer doesn't affect the whole application from running and the failed peer can be recovered by simply restarting it. An example of long running internet services is SWORD [6] which is a resource discovery service on Planet-Lab. And most application management systems, such as Plush [7], application-manager [8], SmartFrog [9], are also considered as long running internet services. To enrich management semantic, user can deploy another management system upon SMON, and the "combined" management system still keeps self-management capability.

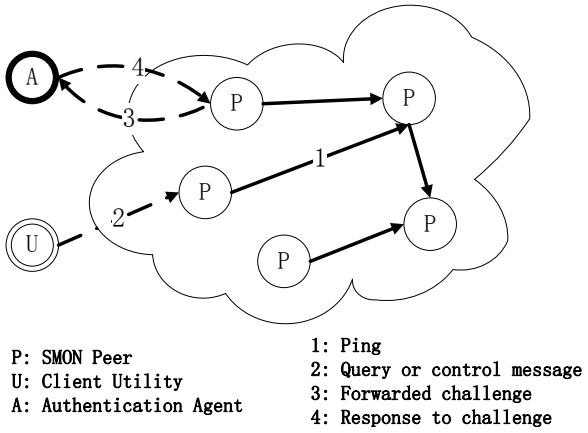

Fig. 1. SMON system design. SMON peers monitor each other by sending ping messages. User can send query or control messages using client utility. Authentication agent resolves authentication challenge from peers to help them log into other machines automatically.

In summary, the paper makes following contributions. We design a scalable and secure distributed management system with built-in self-management capability by leveraging epidemic algorithm. We implement it on Planet-Lab platform. The mathematical analysis and evaluation on Planet-Lab shows that SMON system achieves good performance and scalability.

The rest of paper is organized as follows. We describe SMON design in section II. Section III describes SMON implementation on Planet-Lab platform. We analyze performance and scalability of SMON in section IV. We present evaluation of SMON in section V. Section VI relates SMON with previous systems and section VII concludes.

II. Design

The design of SMON system is presented in figure 1. It consists of three parts: SMON peers, authentication agent and client utility.

The SMON peers run on a set of target machines where distributed applications will be deployed. The peers periodically monitor and maintain each other in their membership list. They will deploy new peers on fresh machines, recover failed peers and upgrade themselves to new version. In this way, a SMON system can manage itself automatically. With the help of authentication agent, a SMON peer can log into remote machines automatically to recover failed peers or deploy new peers. A SMON system can manage a set of distributed applications. It defines the management semantic for long running internet services. User can use the client utility to instruct SMON on application management. The utility can also be used to control a SMON system, such as upgrading it to new version, or query running states of SMON peers.

The design of self-management capacity uses epidemic algorithm for several reasons. First, it has inherent scalability.

Second, the epidemic algorithm is simple and easy to implement. By its simplicity, the SMON system is less prone to runtime errors so that it can run reliably for long time. Third, it is robust and resilient to failure.

A. Self-management

We describe how SMON deploys, recovers and upgrades itself automatically.

Self-deployment: Given a list of target machines, SMON can deploy itself to all the machines automatically. At the very beginning, there is no SMON peer deployed and running yet. A first peer have to be deployed and started manually. It then pings other machines in its membership list and deploys new peers. The new peers repeat the same ping-and-deploy process. While there are more SMON peers deployed and running, this process speeds up exponentially. It is proved that with probability 1 all the reachable machines can be deployed eventually[10].

In detail, a peer P will periodically choose a random machine M from its membership list and send a ping message. If a pong message is not received within predefined timeout intervals, it will start the deployment procedure on machine M. P first authenticates itself with M and log into M with help of the authentication agent (described in II-B), then it copies the installation package of SMON peer to M, starts the package remotely and logout. The installation package first checks the integrity of itself (currently using MD5 checksum) in case of transfer errors during remote copy, then it installs and starts SMON peer.

There may be failures (e.g. connection corrupted, machine crashed) at any time when peer P logs into M, copies and starts installation package remotely. When failure happens, P will just abort the deployment procedure. M will be chosen by another SMON peer at later time. It is expected that a new SMON peer will be deployed on M eventually.

Since peers communicate with each other epidemically, it is possible that two or more peers try to deploy a SMON peer on the same machine M simultaneously. This race-condition is solved without direct coordination among peers. The installation packages from different peers will be copied to different directories of M and they will not overwrite each other. The execution of simultaneously started installation packages are synchronized using OS provided synchronization facilities (e.g. lock file). So only one of the installation will finish and others will abort. In solving the problem, some overhead is introduced because multiple installation packages may be copied, which wastes network bandwidth and storage resources. Through mathematical analysis and evaluation on Planet-Lab platform, it is shown that the overhead is low (a constant value) on average. Since the size of the installation package is small (122KB), the overhead can be considered as insignificant.

It is not easy to define when self-deployment is finished globally. Ideally, it is finished when all the target machines are reached and deployed with SMON peers. However, considering some machines may be unreachable or shutdown from time to time, the 'finished-globally' situation is rare. Even after a machine is deployed with SMON peer, it may crash and be replaced with a new machine with the same host name. The peers then have to continue the ping-and-deploy process as long as they are live and leave the 'finish' definition to user. The user can query how many or which peers are deployed, and determine whether it is appropriate to consider the situation as a finish.

Self-recovery: Two kinds of failures may happen and must be handled by SMON, namely, machine failure and network partition.

When a machine fails, the SMON peer running on it is stopped also. Other peers know this event through periodic pings and they will try to recover the failed peer by restarting it remotely. It is safe to start multiple instances of SMON peer, but it is idempotent to start only one instance.

Network partitions can also be handled by SMON easily. When network partition occurs, SMON peers are splitted into several sub-systems, each of which is connected internally. Implied by epidemic algorithm, each partitioned part will keep manage itself. When two partitions rejoin, the peers in different partition will be able to contact and communicate with each other, leading to the mergence of the whole system.

Self-upgrade: As a distributed system, SMON can upgrade itself to a new version automatically. The upgrade goes online while SMON is running. User needs not to stop SMON before upgrade.

Each SMON peer has an associated version number and it is stored persistently with the peer. A peer will exchange and compare its version with others epidemically. If there is a difference, the peer with lower version will retrieve the installation package from the remote peer and upgrade itself automatically.

It is possible that a peer of new version is known by many peers of old version quickly. To avoid flash crowd, a peer will limit the number of simultaneous request for retrieving installation package.

To upgrade the whole SMON system to a new version, user only has to upgrade one peer to the new version and all the other SMON peers will converge to the latest version eventually. The user can upgrade a SMON peer by using the client utility described in subsection II-E.

Disable/enable self-management: We need mechanism to enable or disable self-management capability to stop SMON from running. When self-management is enabled, we cannot stop SMON by shutting down peers individually. If a peer is stopped by user, it will be restarted soon by another peer. The only solution under such condition is to stop all the SMON peers simultaneously, which is not feasible at large scale.

We use a boolean variable called `livetag` to control epidemic ping among peers. If the value is false, a peer will stop sending periodic ping message to others and the whole SMON system stops maintaining itself.

The `livetag` variable has an associated version number and is maintained on each peer distributedly. The peers exchange `<livetag, version>` tuples epidemically and update the tuple to the latest version. Note that even the epidemic update of `livetag` is controlled by the value of `livetag`. When `livetag` is false, peers still respond to

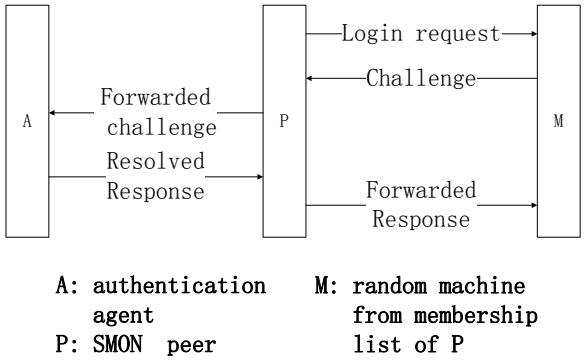

Fig. 2. Interactions among SMON peer, authentication agent and a remote machine when the peer logs into the machine.

incoming messages, so as to help disseminating the update to `livetag`.

Suggested by epidemic algorithm, the whole SMON system will be finally disabled when the `livetag` value of any peer is set to false and peers will not send ping messages to each other. A peer will stop itself from running if no messages is received for enough long time. In this way, a SMON system can be shutdown automatically by setting `livetag` to false.

Combine them together: Each SMON peer runs monitoring and maintenance tasks mentioned above and self-management capacity of the whole SMON is achieved. Basically, a peer can run four maintenance tasks: one for peer deployment, one for peer recovery, one for self-upgrade and one for `livetag` maintenance. As an optimization, we combine first three tasks into one. The version number of SMON is piggybacked in ping/pong messages among peers. And when peer A considers another one B as failed, it will choose from deploying a new peer or recovering the installed peer by detecting installation status of B on remote machine.

B. Security mechanism

SMON's security mechanism should achieve two goals:

- Assist peers to authenticate with remote machines so as to deploy new peers or recover failed peers.
- Authenticate and encrypt access to SMON peers so that it will not be misused (e.g. deploying malwares).

We describe how we design security mechanism to achieve the two goals on Planet-Lab platform and discuss design issues on other distributed platforms. Planet-Lab uses public-key system to authenticate access to the platform. A user access his slice (a set of distributed virtual machines) by using ssh. The private key is kept in user's computer and the public key is distributed to all virtual machines of his slice.

To achieve above-mentioned two goals, SMON employs an authentication agent who holds user's private key securely. As described in figure 2, whenever a SMON peer needs to login into a machine, it will first connects to the sshd server of remote machine and sends a login request. The sshd server will pose an authentication challenge encrypted by user's public key. The peer will indirect the authentication challenge from sshd to the agent and reply to the sshd server with the solved response from the agent. In this way, a peer can login into any machine in his slice and the private key is kept secret for all the times. A symmetric key K_E is shared by all the SMON peers and the authentication agent. It is used to authenticate with peers and agent within the same SMON system, and establish confidential communication channels among them. The shared key K_E is included within the installation package of the SMON peer. When a SMON peer is deployed, K_E is deployed along. Because the deployment is encrypted by ssh, K_E is not leaked out of the network. K_E is safe to be stored within user's slice because the storage resource among virtual machines are well isolated.

The agent should run on machine where user makes sure his private key is safe. The agent can be also replicated if required. When a SMON system is deployed at large enough scales, there can be multiple instances of authenticate agent for performance considerations. SMON peers can be directed to their nearest agents by DNS server.

The above agent-based scheme works for challenge-based authentication method, e.g. password or public-key authentication, which is commonly used in many systems. It relies on two key factors:

- Using challenge-response authentication mechanisms, such as public-key or password authentication.
- K_E is safe to be stored with SMON peer.

For example, the mechanism can be applied on Amazon EC2 platform and private clusters. For Amazon EC2, user starts instances of virtual machines based on an AMI (Amazon Machine Image). The user access his AMI instances using private key. For private clusters, virtual machine may be not adopted but users who can access the system are not adversarial and K_E can be considered safe to stored on local file systems.

C. Membership

Each SMON peer maintains a list of machines as its membership list. The membership maintenance of SMON system is different from other distributed systems such as file sharing [11], streaming [12], Content Distribution Network [13]. While others maintains a list of peers that are currently online, SMON maintains a list of machines on which there should be peers running. The failure of SMON peers will not affect the membership list at all. The list is specified by user and should not changes very often. And an important requirement of SMON is, even when there is only one SMON peer running, it should be able to know all the target machines where SMON should be deployed. The implication is that every running SMON peer should be able to know the full list of target machines.

In current design, a simple scheme is used: each SMON peer will store a replica of the full list of target machines persistently in compressed format. The list also has an associated version number. SMON peers exchanges their membership versions epidemically and update the list. User can update the membership list to change the set of machines on which SMON runs. This scheme is enough for handling machine list containing even tens of thousands of machines because only the machine names or IPs are stored in the list. An estimation

shows that, storing 15000+ machines names in zip format takes about 100K bytes, which is an acceptable result.

When membership changes, newly added machines will be deployed with new SMON peers. The peer running on removed machines will receive the new machine list and find that they are not in the list. It will then only reply to incoming messages to help spreading membership changes. And it will stop maintaining any SMON peer. After most peers update their membership list, the peers in removed machines will kill themselves after they don't receive any messages for enough long time.

D. Application management

SMON can manage one or more distributed applications. The management semantic is defined in the following four dimensions, which is a general framework for describing application management requirements.

- resource requirement: different applications put various demands on computing resources, such as CPU load, free disk space and memory, network bandwidth and latency, etc. Besides from characteristics on current resource usage, user may be also interested in resource availability and reliability which can be inferred by summarizing historic data. The management system should find best set of resources as described in specification.
- deployment: The management system follows instructions given by user to copy, unpack and install applications on acquired resources. The management system should also support a variety of file transfer mechanisms.
- synchronization: distributed applications may work in multi-phases, such as scientific parallel programs. The management system should have barrier mechanism to execute application workflow in steps.
- maintenance and recovery: While application is running, the resource usage may vary and failures of different levels may happen. The management system should monitor dynamic resource conditions and recover applications from failure as soon as possible.

Two common kinds of distributed applications are long running internet services and grid style computation programs. The management requirements are summarized in table I.

SMON support management semantic for long running internet services. Such services include file sharing [11], streaming [12], Content Distribution Network [13], etc. These services may run for months or years and must robust to a variety of failures. After the services are deployed, the application peers can be started independently without particular sequences. The peers organize themselves into structured or random overlays and handles network churns and failures. When a peer fails, the overlay can re-organize itself. The failed peer should be restored as quickly as possible to maintain service availability and performance.

To manage a long running service, SMON will deploy the application to a set of machines. It is up to the user to decide the machines list using resource discovery services. The application peers are started by SMON once they are deployed without synchronization. A SMON peer will monitor and maintain application peer in the same machine. When an application peer is failed, it will be restarted for recovery.

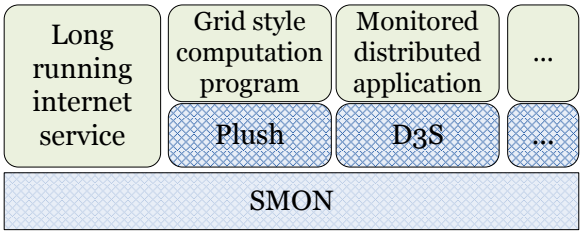

Fig. 3. Enrich management semantic by deploying other management systems upon SMON.

Because distributed application management systems fit in the catalogue of long running services, we can easily enrich the management semantic by deploying another management system upon SMON. In this way, the management functionality can be "extended" greatly, while the combined management system keeps self-management capability. For example, we can layer Plush [7] upon SMON to manage grid style computing programs with barrier support for dividing computations in phases, as shown in figure 3. Or we can layer D3S [14] on SMON to monitor and debug correctness of distributed applications as a set of predicates, which are defined on runtime states of application processes on many machines.

SMON deploys applications using epidemic algorithm. An application is uniquely identified by its name. Periodically, a SMON peer A will choose one locally deployed application and exchange the application name with a random peer B. If B knows an un-deployed application, it will try to retrieve the installation package from A and install the application.

After application is deployed and started, it is monitored and maintained by SMON peer in the same machine. SMON reads application specification for management options. User can specify application's default running state and change the setting using client utility provided by SMON.

The application's running states can be:

- Online: the application will be restarted once stopped.
- Offline: the application will be stopped.
- Ignore: the application will be started for the first time and its state will not be monitored after that.

In the specification, user should specify three scripts to start, stop and restart the application. The restart script is called to recover the failed processes, and may include clean up steps before application is restarted.

A SMON peer may report the application states to a central server timely if specified. And whenever the application state changes, the SMON peer will report the changes immediately.

E. Client utility

A client utility is provided for user to control SMON. The communication between the utility and any peers is authenticated and encrypted by the share key K_E. To deploy SMON, the utility will copy the installation package of SMON peer to a machine and start the package, then SMON will deploy itself automatically. To upgrade SMON, the utility mimics

	long running internet service	grid style computation program
resource requirement	prefer resources with good reliabilty to provide sustained performance for long time (may be years)	prefer resources with huge computation and storage capabilities, reliability is required only when computations are running (usually short, e.g. days)
deployment	platform specific	platform specific
synchronization	peers can be started independently, and there is no constraints on peers starting order	parallel computation programs must be started nearly simultaneously, and the computation may have multiple phases.
maintainance and recovery	failure of a single peer can be handled gracefully, and the peer can be restored to maintain sustained performance	failure of a single computation program may lead to failure of whole computation.

TABLE I
COMPARISON OF MANAGEMENT REQUIREMENTS FOR LONG RUNNING INTERNET SERVICES V.S. GRID STYLE COMPUTATION PROGRAMS.

itself as a SMON peer and notifies a random SMON peer that a new version available. The peer will then retrieve the installation package from utility's machine and upgrade itself. Then SMON will be upgraded. Similarly, the utility can notify a random peer to install an application, update the member list and enable or disable self-management capability of SMON. The latest version numbers of SMON peer, membership list and `livetag` are stored locally on user's machine. User can query any SMON peer with the help of the utility.

III. IMPLEMENTATION

We implement the prototype of SMON system in Python, and the final installation package size of a BON peer is about 122KB, with 1000+ lines of code.

The maintenance tasks of a SMON peer is implemented in several threads, including one for monitoring and maintaining other peers, one for updating membership list, one for update `livetag`. For each application, a separate thread will be started for maintaining the application.

The communication of SMON peers and agents are implemented as RPCs, and they are summarized in table II.

SMON peer spawns a ssh/scp process to deploy on remote machines. A modified `ssh-agent` is started which forwards the authentication challenge from ssh/scp to the remote authenticate agent. The agent only implements one RPC interface `resolve_challenge` described in table II.

The configuration parameters used at SMON runtime is stored in a configuration file. It stores the time intervals among consecutive maintenance tasks, the version numbers for SMON peer and membership list, the <livetag, version> tuple, the address of authentication agent. The configuration file is implemented as a SQLite database file to avoid data loss at machine crash. The membership list is stored separately in compressed format.

IV. ANALYSIS

We analyze performance of SMON in this section.

SMON uses epidemic algorithm to maintain itself. Peers will initiate periodic communication to other random peers. We will show that these maintenance tasks have good performance and scalability. In the analysis, we assume that SMON works in synchronized way, that is each peer performs its maintenance tasks in synchronized rounds and the tasks finish almost instantaneously. Although the real SMON system works asynchronously, the analysis can still give insights on performance of SMON design. We will also evaluate SMON in real conditions (see section V) and validate our analysis.

Self-deployment	push
Self-upgrade	push-pull
Self-recovery	push
Enable SMON	push
Disable SMON	pull
Application deployment	push-pull

TABLE III
DIFFERENT VARIANTS OF EPIDEMIC ALGORITHM USED BY MAINTENANCE TASK OF SMON.

The epidemic algorithm has three variants, called push, pull and push-pull, and they are used by different maintenance tasks of SMON as summarized in table III. Although they differ in details, the basic result of epidemic theory shows that each one of the three methods can eventually infect the entire population. The studies also show that starting from a single peer, this is achieved in expected time (the rounds number) proportional to the log of the population size ($\log N$).

Take self-deployment process as an example. Starting from a single SMON peer, all the target machines will be deployed with a SMON peer. According to existing studies [10], the self-deployment process can be modeled as a branching process with population n. After r rounds, the expected fraction of deployed machines is:

$$Y_r \approx \frac{1}{1 + ne^{-r}}$$

The relation of Y_r and r is demonstrated in figure 4.

The ratio of deployed machines to un-deployed ones increases exponentially, on average, by a factor of e in each round. This indicates that the self-deployment has good performance.

To deploy a fix fraction of machines, the expected round number r is:

$$r \approx \ln(n) - \ln(\frac{1}{Y_r} - 1)$$

r is in the logarithm of total population n, which implies good scalability of self-deployment process.

RPC	Description
`ping(ver)`	Call with local SMON peer's version and return remote peer's version.
`retrieve_peer(ver)`	Retrieve installation package of SMON peer with specified version.
`exchange_livetag(tag, ver)`	Call with local <livetag, version>, and return remote peer's <livetag, version>.
`exchange_member(ver)`	Call with local membership list version and return remote peer's membership list.
`retreive_member(ver)`	Retrive membership list of specified version.
`exchange_app(app_name)`	Call with an application name, return true if remote peer has installed the application.
`retrieve_app(app_name)`	Retrieve installation package of an application with specified name.
`resolve_challenge(challenge)`	Return the response to an authentication challenge.
`set_app_status(app_name, status)`	Set application status (online, offline, ignore).
`get_app_status(app_name)`	Get application status.

TABLE II
RPC INTERFACES IMPLEMENTED BY SMON PEER AND AUTHENTICATION AGENT

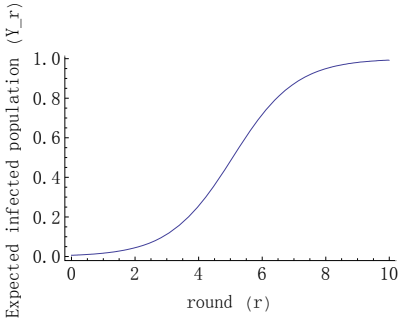

Fig. 4. Relation of expected infected population Y_r with round number r.

Another performance metric in concern is the overhead in the self-deployment process. Since we use epidemic algorithm, there may be duplicated deployed SMON peers in a machine. Although there will be only one instance running, the duplicated deployment will waste network resource and disk storage at some extent.

We next show that the overhead is small on average (constant actually). Assuming that there are n machines in total, and m machines have been deployed already. For any one of $n-m$ un-deployed machines, it may be selected by $k, (k <= m)$ peers as the deployment target in the next round. The distribution of k is can be expressed as binomial distribition with parameter $(m, \frac{1}{n})$.

$$P(X=k) = C_m^k (\frac{1}{n})^k (1-\frac{1}{n})^{m-k}$$

We define the simultaneous deployment number $k, (k>0)$ as the deployment overhead. It is the conditional probability when $k>0$.

$$\begin{aligned} P(X'=k) &= P(X=k|X>0) \\ &= P(X=k)/(1-P(X=0)) \\ &= P(X=k)/(1-(1-\frac{1}{n})^m) \qquad (k>0) \end{aligned}$$

The expectation of k, the overhead number, is:

$$\begin{aligned} E(X') &= \sum_{k=1}^{m} k \cdot P(X=k|X>0) \\ &= \frac{1}{(1-(1-\frac{1}{n})^m)} E(X) \end{aligned}$$

Further, note that when n is large, and m is approaching n, we have

$$\lim_{n\to\infty, m\to n} (1-\frac{1}{n})^m = 1/e$$

Thus,

$$\begin{aligned} E(X') &\approx 1.582 \cdot E(X) \\ &= 1.582 \times \frac{m}{n} \end{aligned}$$

And finally, we have:

$$\lim_{n\to\infty, m\to n} E(X') = 1.582$$

This shows that the average overhead number will approach a constant value when SMON is deployed at large scale.

At last, we show that the average number of ping messages received by any peer is also constant. When SMON is running on n peers, the number of ping messages p received by a SMON peer is in the binomial distribution of parameter $(n, 1/n)$. Thus we have

$$E(p) = n \cdot 1/n = 1$$

V. EVALUATION

We conducted experiments on Planet-Lab platform to evaluate SMON system, and compare the results with analysis in section IV. Planet-Lab is a global research network consisting of 900+ nodes at 400+ sites around the world.

We evaluate the performance of self-management design. As the maintenance tasks performed by SMON peer to deploy, recover and upgrade the system use the same epidemic algorithm, we only present the results on self-deployment and enabling a disabled SMON system.

First, we evaluate the performance of self-deployment of SMON. We choose 159 nodes in Planet-Lab platform and start a SMON peer. SMON then start deploying itself on all 159 nodes.

In our configuration, A SMON peer detects a random nodes and sleep for 5 seconds. We finally collected data from 154 nodes out of 159 nodes. There are 5 nodes unreachable at data-collecting stage and their data are omitted in the result. This kind of failure is common for large scale distributed systems. Figure 5 shows the progress of self-deployment. We

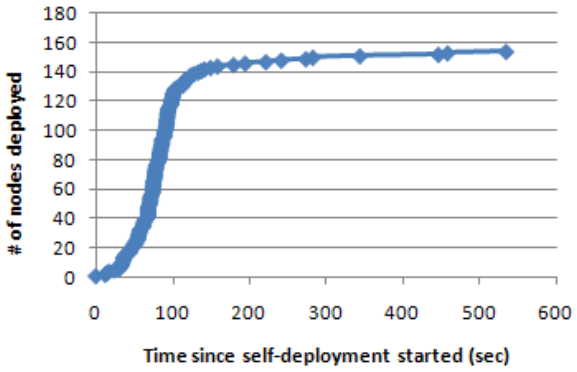

Fig. 5. Progress of self-deployment process on 159 nodes.

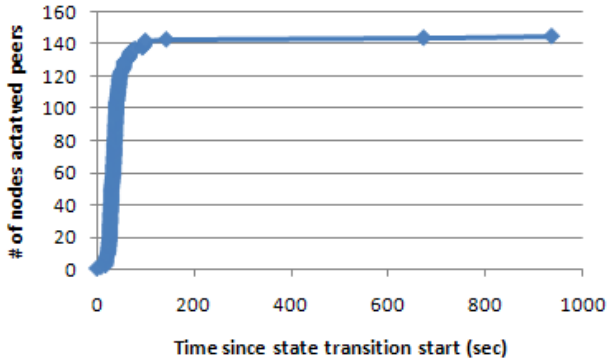

Fig. 6. Progress of state transition of self-management from disabled to enabled.

can see that 90% (143) of nodes are deployed successfully with a SMON peer within 149 seconds. The median deployment time is 93 seconds while the largest one is 533 seconds. As a comparison, figure 5 has similar trends with analysis result (figure 4). But figure 5 exhibits an obvious long tail. The long tail in figure 5 are caused by two reasons: either the network to the machine is slow or the the machine itself is overloaded and doesn't respond quickly.

We then evaluate the performance of enable/disable self-management functionality of SMON system. We first disable self-management of SMON system deployed on 159 PlanetLab and then enable it again. The progress of state transition from disabled to enabled is shown in Figure 6. For 90% percentile of peers, the states changes after 143 seconds and the median state transition time is 37 seconds.

To evaluate the scalability of self-deployment process, we deploy a new instance of SMON system at a different scale (24 nodes) and compare the performance of self-deployment process. Table IV summarizes the statistics results. We can see from the table that the scalability is good. While scale difference between two systems is about 6.6 (159/24) times, the 90-percentile deployment time is only 1.75 (149/85) times of difference, while the median value is 1.41 (82/58) times of difference. The above two ratios is close to the ratio of logarithm of system scales ($\log 159 / \log 24 = 1 : 59$).

During the self-deployment process, there are cases that

	24 nodes	159 nodes
median	58 sec	82 sec
90% percentile	85 sec	149 sec
Final	103 sec	533 sec

TABLE IV
COMPARISON OF SELF-DEPLOYMENT PROCESS AT DIFFERENT SCALES IN SECONDS.

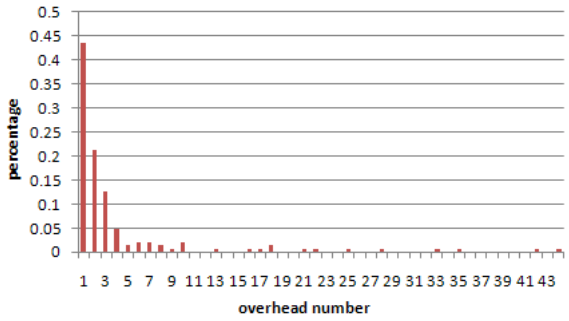

Fig. 7. Overhead (copy number of simultaneous deployment) introduced by race condition during self-deployment process

multiple SMON peers try to deploy an instance on the same node and this can cause extra overhead. The overhead wastes network bandwidth and the storage at the deployed node. We evaluate the overhead by count the simultaneous deployment happened at each node. The result is shown in Figure 7. We can see that for 43.71% of nodes there is exactly one deployment on each of them. Obviously, most of nodes have a deployment overhead within 10. These nodes account for 92.04% of all nodes and the average overhead of these nodes is 2.31. The average overhead for all nodes is 4.30 which is an acceptable number. The number 4.30 is larger than the analysis result (1.582) in section IV. The reason is that, in the analysis we assume that SMON works synchronously but the real system works asynchronously and the deployment takes time. During the time a peer chooses a machine to deploy a new peer, other peers will choose the same machine and don't notice the simultaneous deployment. This will cause greater overhead number than analysis.

VI. Related Work

Many systems have been proposed to help managing distributed applications. Plush [7] and Application Manager [8] are management systems which support deploying, monitoring and controlling applications centrally. Plush can bootstrap (self-deployment) its clients automatically, but the centralized approach limits its scalability. Plush-M [15] further improved Plush's scalability on managing applications by replacing star topology in Plush with RanSub [16], but the management of Plush clients is still in centralized approach. Condor-G [17] is a workload management system for compute-intensive jobs on grid infrastructure. None of the above management systems fully address self-management issue, while SMON addresses the problem using epidemic algorithm.

Researchers have been studying self-* properties of distributed systems for long time. Self-stabilization [18] is a

concept of fault-tolerance in distributed computing. A self-stabilization system will converge to a legitimate state from any state. DHTs [19], [20], [21] are distributed systems that have self-organizing properties. They will automatically organize their overlay topologies conforming to predefined constraints under changing network conditions. [22] proposed self-hosting system that acquires or releases resources dynamically and automatically deploys the system to acquired resources (as real or virtual machines) using underling system management tools. SMON is complementary to these systems and these techniques can be combined together to build distributed systems which are more stable and efficient with little human management efforts.

VII. Conclusion

In this paper, we describe SMON, a management system with self-management capability. By using epidemic algorithm and authentication agent, SMON can monitors and maintains itself automatically and securely on a set of distributed machines. Through mathematical analysis and evaluation, it is shown that SMON achieves good performance and scalability.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP*, 2003.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating system support for planetary-scale network services," in *NSDI*, 2004.

[4] S. Garfinkel, "Commodity grid computing with Amazon's S3 and EC2," *;login: (The USENIX Magazine)*, Febuary 2007.

[5] C. Catlett, "The philosophy of teragrid: Building an open, extensible, distributed terascale facility," in *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2002, p. 8.

[6] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson, "Design and implementation trade-offs for wide-area resource discovery," *ACM Transactions on Internet Technology (TOIT)*, vol. 8, no. 4, pp. 1–44, 2008.

[7] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Remote control: Distributed application configuration, management, and visualization with plush," in *LISA '07: Proceedings of the 21st Large Installation System Administration Conference*, 2007.

[8] "Planet-lab application manager," http://appmanager.berkeley.intel-research.net/.

[9] "Smartfrog," http://www.smartfrog.org/.

[10] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems." *IEEE Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[11] B. Cohen, "Bittorrent," http://www.bittorrent.com.

[12] D. Kostic, A. Rodriguez, J. R. Albrecht, and A. Vahdat, "Bullet: high bandwidth data dissemination using an overlay mesh." in *SOSP*, 2003, pp. 282–297.

[13] M. J. Freedman, E. Freudenthal, and D. Mazières, "Democratizing content publication with coral," in *NSDI*, 2004, pp. 239–252.

[14] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: debugging deployed distributed systems," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 423–437.

[15] N. Topilski, J. Albrecht, and A. Vahdat, "Improving scalability and fault tolerance in an application management infrastructure," in *USENIX Workshop on Large-Scale Computing (LASCO)*, 2008.

[16] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat, "Using random subsets to build scalable network services," in *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[17] J. Frey, T. Tannenbaum, M. Livny, I. T. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, 2002.

[18] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 149–160.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 161–172.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[22] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe, "Dependable self-hosting distributed systems using constraints," in *Proceedings of Fourth Workshop on Hot Topics in System Dependability*, 2008.