SMON: Self-Managed Overlay Networks for Managing Distributed Applications

Chongnan Gao†, Hongliang Yu†, Guangyu Shiy‡, Jian Chen‡, Weimin Zheng†

†Computer Science and Technology Department, Tsinghua University, Beijing 10084, China
‡Huawei Technologies Co., Ltd, Shenzhen, 518129, China
Email: gaochongnan@gmail.com, {hlyu, zwm-dcs}@tsinghua.edu.cn

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, as a distributed application, need to be deployed and maintained continually. In this paper, 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 to a set of machines and recovers failed peers *securely*. It can also upgrade itself to new versions online. Through mathematical analysis and evaluation on Planet-Lab platform, we show that SMON achieves good performance and scalability.

I. Introduction

Distributed applications [1], [2] are at the heart of today's Internet services. These applications run on computing platforms with hundreds or thousands of machines, such as Planet-Lab [3], Amazon EC2 [4] or private clusters. The applications utilize huge amount of resources provided by the platforms and boost their performance significantly. They handle faiures gracefully and the failure of a single element in application or computing platform has little influence on application functionality and performance.

Managing these distributed applications faces many challenges. Distributed application management system is designed to ease the burdens of management at large scale by automates many aspects in application deployment and maintenance. To manage distributed applications efficiently on many machines, the management systems have to take distributed designs, or to say, they are also distributed applications. A managent system consists of many management peers on each machine where applications will be deployed. It adopts different schemes to organize management peers, such as centralized, hierarchical or peer-to-peer schemes. The management peers work corporately to deploy applications to a set of machines. After applications are configured and 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 they use 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 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, certain security mechanism is needed so that peers can log into other machines without manual assistant. And access to

Supported by National Natural Science Foundation of China under Grant No. 60603071, the Major State Basic Research Development Program of China (973 Program) under Grant No. 2007CB311100, National Key Technology R&D Program under grant No. 2006BAK15B10 and Huawei Research Fund under contract No. YBCB2009032.

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. The membership list contains machines where SMON peers should be deployed and running. It will deploy new SMON peer on a fresh machine or recover failed peer. 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, and it 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. Also, the epidemic approach is robust and resilient to failure.

For the second challenge, we employ a separate authentication agent to assist peers to log into remote machines automatically to deploy new peers or recover failed peers. To log into a machine, a peer will redirect 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 protect 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 peer can be recovered independently. Most application management systems, such as Plush [5], are 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. In this sense, SMON becomes a "kernel" which bootstraps other management systems.

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.

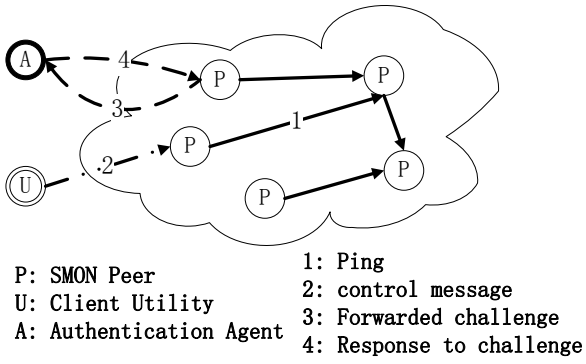

Fig. 1. SMON system design. SMON peers monitor each other by sending ping messages (arrow 1). User can send query or control messages using client utility (arrow 2). Authentication agent resolves authentication challenge from peers to help them log into other machines automatically (arrow 3 and 4).

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. The membership list contains machines names where SMON peers should be deployed and running. With the help of authentication agent, a SMON peer can log into remote machines automatically to recover failed peers or deploy new peers. SMON defines the management semantic for long running internet services. User can use the client utility to instruct SMON on application management.

A. Self-management

A SMON peer runs the pseudo-code in figure 2 to manage other peers. We start from line 5 and explain line 2 and 3 later. A SMON peer will choose a random machine and send it a ping message (line 5-7). If timeout happens, it will try to deploy a new peer or recover the failed peer on the remote machine (line 10-16). If the remote peer is running, it will reply with a pong message. The version number of two peers are piggybacked in the ping/pong messages and the peer with lower version will upgrade itself automatically.

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

1) 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. 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[6].

In detail, a peer P will periodically choose a random machine M from its membership list and send a ping message.

```
while(1):                                                    1
    if(livetag==False):                                      2
        continue                                             3
                                                             4
    p = random.choice(membership_list)                       5
    try:                                                     6
        remote_ver = ping(p, my_ver)                         7
        if(remote_ver > my_ver):                             8
            upgrade_myself()                                 9
    except Timeout:                                          10
        conn = Connect(p, auth_agent)                        11
        if(conn.is_peer_installed()==True):                  12
            conn.recover_peer()                              13
        else:                                                14
            conn.deploy_peer()                               15
        conn.close()                                         16
                                                             17
    sleep(WAIT_INTERVAL)                                     18
```

Fig. 2. SMON peer working flow

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.

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.

2) 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. In this way, although false positives are possible in fault detection, no system churn will be introduced by spurious recovery.

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.

3) 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. 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.

4) 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 (line 2-3 in figure 2). 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.

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.

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

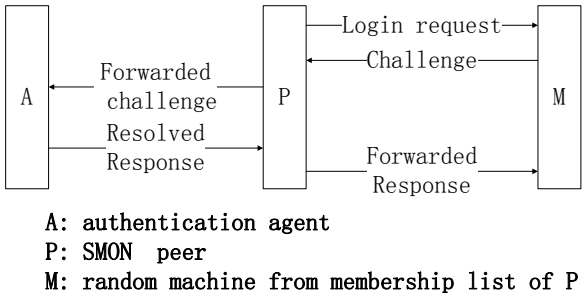

Fig. 3. Interactions among SMON peer, authentication agent and a remote machine when the peer logs into the machine.

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 3, 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. While other distributed system 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.

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.

	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.

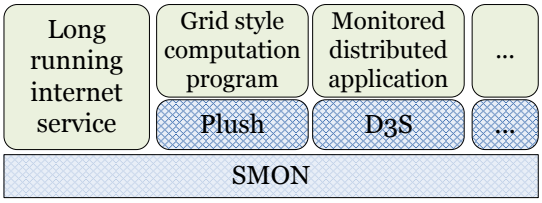

Fig. 4. Enrich management semantic by deploying other management systems upon SMON.

SMON support management semantic for long running internet services. Such services include file sharing, streaming, Content Distribution Network, 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 (shortened as "app-peer") can be started independently without particular sequences. The app-peers organize themselves into structured or random overlays and handles network churns and failures. When an app-peer fails, the overlay can re-organize itself. The failed app-peer should be restored as quickly as possible to maintain service availability and performance.

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 [5] upon SMON to manage grid style computing programs as shown in figure 4. Plush provides barrier support for dividing computations in phases, manage application in coordinated approach or on a sebset of nodes. Or we can layer D3S [7] 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 notify a random peer B with the application name. If B knows an un-deployed application, it will try to retrieve the installation package from A and install the application. If the retrieve fails, B will just wait for the next notification. The epidemic approach ensures that an application can be pushed out to all SMON peers eventually.

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. 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.

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.

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 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 the SMON 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 interface of SMON peers and agents are implemented as XML-RPCs, which are not listed for space limitation. SMON peer uses paramiko[1] to copy files or send commands to remote machines using ssh protocol. It will redirect the authentication challenge to the authentication agent using `resolve_challenge` XML-RPC interface. 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 as a compressed file.

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.

The epidemic algorithm has three variants, called push, pull and push-pull, and they are used by different maintenance tasks of SMON. 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 [6], 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 5.

From the equation above, 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.

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.

[1]a ssh library implemented in Python

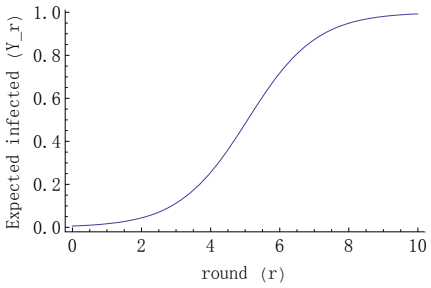

Fig. 5. Relation of expected infected population Y_r with round number r.

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 $k>0$

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

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. Planet-Lab is a global research network consisting of 900+ nodes at 400+ sites around the world. We show that SMON design achieves good scalability and performance. The overhead introduced in self-deployment is low. And the load on authentication agent authentication challenge is low enough that a single agent can support nearly 1000 SMON peers on a moderate PC machine.

A. Performance

First we evaluate the performance of SMON. We concentrate on two representative activities: deploying a new SMON system and disabling it. We select 117 nodes on Planet-Lab, all of which are university nodes in North America. To deploy a SMON system, we use the client utility of SMON to deploy a peer on `planetlab2.ucsd.edu` (shortened as `ucsd2` node) and start it. Then SMON will deploy itself to other 116 nodes automatically. After the deployment is accomplished, we just disable it by issuing a RPC call to `ucsd2` to set the `livetag` to false. The ping interval of the SMON peer is configured as 5 seconds.

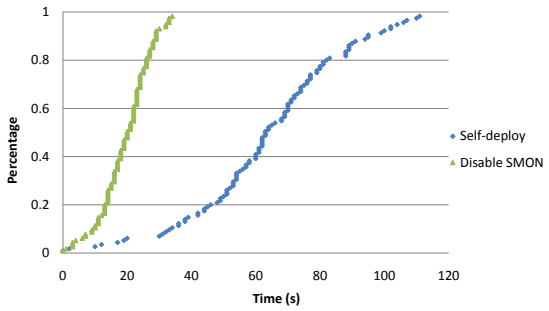

Fig. 6. Deploy SMON on 117 nodes and then disable it.

We collect logs generated by SMON peers and summarize performance data. Figure 6 summarizes the progress of deploying and disabling SMON on 117 nodes, and they take 111 seconds and 34 seconds in total, respectively. The progress curves show similar trends with analysis result (Figure 5). The difference in two curves are mainly caused by the time cost to "infect" a new peer. It takes longer time to deploy a new peer than to set `livetag` of another peer to false. Our experience of using SMON also shows that progress curve may exhibit long tail when some nodes are less responsive because the network connection to the nodes are slow or the nodes are overloaded.

When SMON deploy itself, multiple peers may try to deploy and start a new peer on the same nodes simultaneously. This introduces some overhead, although only one peer is started at last. In the analysis, we show that the average overhead is a constant value of 1.582, and we validate the analysis here. Table II summarized the statistics for the overhead number. On most of nodes (72 of 117 nodes, or 64.54%), there's exactly one deployment. The largest overhead number is 23. On average, the overhead number is 1.88, with an error of 18.84% comparing to analysis result. The error is caused mainly by the difference between reality and the assumption we made at analysis. In the analysis, we assume that SMON peer works in synchronized way and the deployments finish almost instantaneously. But in the real world, the deployments are performed asynchronously and take time. Before a new peer is finally deployed and started, many peers can try their deployments on the same machine. Things get worse when network connection to the machine is slow or the machine is overloaded, which make the deployment slower than normal. In this way, a large overhead number may be produced.

overhead	nodes	%
1	72	61.54
2	26	22.22
3	12	10.26
4	3	2.56
5	2	1.71
15	1	0.85
23	1	0.85

TABLE II
OVERHEAD NUMBER DISTRIBUTION.

B. Scalability

To evaluate the scalability of self-deployment process, we deploy a new instance of SMON system at a different scale (24 random nodes out of 117 nodes) and compare the performance of self-deployment process. Table III summarizes the statistics results. We can see from the table that the scalability is good. While scale difference between two systems is about 4.88 (117/24) times, the 90-percentile deployment time is only 1.57 (91/58) times of difference, while the median value is 1.31 (63/48) times of difference. The above two ratios is close to the ratio of logarithm of system scales ($\log 117 / \log 24 = 1.49$).

	24 nodes	117 nodes
median	48 sec	63 sec
90% percentile	58 sec	91 sec
Final	61 sec	111 sec

TABLE III
COMPARISON OF SELF-DEPLOYMENT PROCESS AT DIFFERENT SCALES.

C. Load on authentication agent

SMON employs an authentication agent to help peers automatically log into another node to deploy a new peer or recover a failed peer. It solves the authentication challenge on behalf of the requested peer. Solving the authentication challenge involves expensive computations and it is a reasonable to ask whether the agent a possible performance bottleneck.

We monitor the CPU usage on the authentication agent while SMON is deploying itself to 117 nodes, and summarize the result as figure 7. The authentication agent runs on a

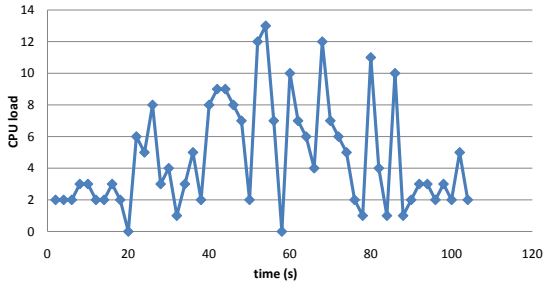

Fig. 7. CPU load on authentication agent.

dedicated PC machine of our lab. The PC is configured with AMD Dual Core CPU 2.1GHz, 2GB memory and run Windows XP. A monitoring script is employed to gather CPU usage on the agent machine using `mpstat` command.

It can be seen that the CPU load is about 13% at maximum, and usually below 10%. The load on the agent is proportional to the frequency of authentication requests, and the frequency reaches its maximum value when half of the nodes are deployed with SMON peer. The maximum frequency is also proportional to the scale of the nodes in theory. Thus we can conclude that a single authentication agent is able to support nearly a thousand of nodes to accomplish self-deployment. After SMON is deployed, the frequency of authentication requests depends on frequency of nodes failures and is usually low. And the agent could support SMON at large scale without becoming a bottleneck.

VI. Related Work

Many systems have been proposed to help managing distributed applications. PlanetLab Application Manager (appmanager [8]) is designed mainly for managing long running services and utilizes a simple client-server model. The clients actively report recent status to and receive new instructions from server. Plush [5] is an extensible management system in client-server model for large-scale distributed systems, including Planet-Lab and Grid. With user provided management specification, it handles resource discovery (using SWORD), deployment, maintenance and execution flow automatically. It has built-in barrier support and can manage grid-style applications which run in sychronized phases. Plush-M [9] further improves Plush's scalability by replacing star topology with RanSub [10]. SmartFrog [11] is a framework for building and deploying distributed applications. SmartFrog daemons running on each participating node work together to manage distributed applications. Unlike Plush, there is no central point of control in SmartFrog: work-flows are fully distributed and decentralized. Although many management systems have been proposed, their self-management problem is not fully addressed. User has to set up and maintain a management on their own, usually using centralized approach. Plush can bootstrap (self-deploy) its clients automatically, but the centralized approach limits its scalability and makes no difference. SMON provides a scalable and secure self-managment mechanism using epidemic algorithm. It turns any managment system "self-managable" by deploying it upon SMON.

Researchers have been studying self-* properties of distributed systems for long time. Self-stabilization [12] is a concept of fault-tolerance in distributed computing. A self-stabilization system will converge to a legitimate state from any state. DHTs [13], [14], [15] are distributed systems that have self-organizing, self-healing and self-tuning properties. They will automatically organize their overlay topologies conforming to predefined constraints, and handle network churns automatically. [16] proposes a 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] 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*, 2007.

[6] 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.

[7] 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*, 2008.

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

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

[10] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat, "Using random subsets to build scalable network services," in *USITS*, 2003.

[11] "SmartFrog," http://www.smartfrog.org/.

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

[13] 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*, 2001.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM*, 2001.

[15] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Middleware*, 2001.

[16] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe, "Dependable self-hosting distributed systems using constraints," in *HotDep*, 2008.