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 environments. The management system is designed in distributed approach and also need to be well monitored and maintained throughout its lifecycle. Currently, distributed management systems are monitored and maintained using centralized approach, which is not scalable and efficient.

We propose SMON (Self-Managed Overlay Network) in this paper. SMON is a distributed system with built-in self-management capability. It 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. SMON can manage a set of distributed applications. By deploying other management system on top of SMON, its management capability can be extended greatly.

1 Introduction

Distributed systems [8, 4] 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 [2], Amazon EC2 [7] and Teragrid [3]. These platforms can provide huge amount of storage and computing capabilities that boost performance of distributed applications significantly. However, it is still difficult to design and manage distriubted 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 process should scale well so that it still works on thousands of machines. During deployment,

Distributed application management system is designed to simplify tasks involved in deploying and maintaining the applications after they are designed and implemented. To efficiently manage distributed applications at large scale, the management system is designed in distributed approach. It consists of many "peers"[1] on each machine where application will be deployed. The peers form an overlay network, and they work corporately to deploy application to a set of machines. After the application is started, each peer monitors and maintains application's processes within the same machine box. User can query the applications status and control them on demand with the help of management system.

The distributed design makes management system a distributed application in essential and it introduces an important problem: the management system need to be deployed first and maintained continually through its lifecycle. Currently, this problem is addressed by using centralized approach. A central controller first deploys and starts the peers of management system on a set of machines. After that, the controller periodically monitors the peers and recovers the failed ones. If a crashed machine is replaced by a new one, the controller has to deploy a new instance of peer on the fresh machine. The controller can also upgrade peers to new versions with bug fixes and improved performance. The centralized approach has several disadvantages. First, it is not scalable at deploying peers. Second, it is not efficient at monitoring the peers. Frequent monitoring detects peers failures quickly, but it puts heavy burden on network resource of the central controller when establishing connections and sending messages to remote machines. Third, The static star topology from central controller to peers cannot handle changes or failures in network environments, such as network partition.

In this paper, we propose Self-Managed Overlay Net-

*Supported by National Science Foundation of China (Grant No. 60603071) and National Basic Research Program of China (973, Grant No. 2007CB311100).

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

work (SMON) that addresses the problem of deployment and maintenance of distributed management systems. SMON is a distributed management system with built-in self-management capability, namely, self-deployment, self-recovery and self-upgrade. It consists of peers on every target machines where applications will be deployed and maintained. The peers monitor each other and automatically deploy new peers on fresh machines, recover failed peers, or upgrade peers of old versions. The collective behavior of all peers gives rise to a distributed system that can deploy itself to a set of machines, recover failed peers and update itself to new versions. 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 at monitoring and maintaining itself. Second, there should be built-in security mechanism so that peers can automatically login into other machines to deploy new peers or recover failed peers. 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 have good extensibility.

SMON's design addresses the challenges as follows. For the first challenge, SMON peers monitor and maintain each other using epidemic approach. This ensures good scalability ($O(\log N)$) when system goes large. For the second challenge, a separate authentication agent is used to store and protect login credentials. When a peer needs to login into another machine, it redirects the authentication challenge from remote machine to the agent and replies the response solved by the agent to the machine. The credential is never leaked out of the agent. The authentication load on the agent is very light and a single agent can support a number of peers. We use a symmetric key to authenticate and encrypt access to SMON peers. For the third challenge, SMON can be easily extended by upgrade itself to a new version with new features or improved performance. It can also manage other distribute management systems so the management functionalities can be extended greatly.

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

The rest of paper is organized as follows. We describe SMON design in section 2. Section 3 describes our implementation on Planet-Lab platform. We present evaluation of SMON in section 4. Section 5 relates SMON with previous systems and section 6 concludes.

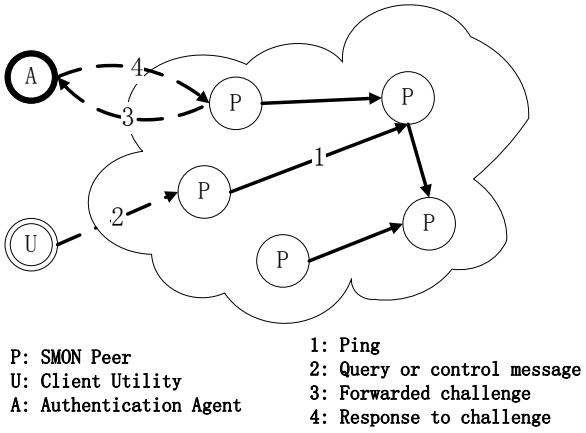

Figure 1: Architecture of SMON system. SMON peers monitor each other by sending ping messages. User can send query or control messages to any peer. Authentication agent resolves authentication chanllenge from peers to help them login into other machines automatically.

2 Design

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

The SMON peers run on a set of target machines where distributed applications will be deployed. Each peer maintains the full list of target machines as its membership list. The peers monitor and maintain each other epidemically. A peer periodically chooses a random peer and sends it a ping message. If the pong message is not replied within a timeout interval, it is considered as failed. The peer will try to recover the failed peer by restarting it remotely, or deploying and starting a new peer on fresh machines. Each peer has an associated version number. The peers exchanges their version numbers epidemically and peers of lower version upgrade themselves to the latest versions.

The authentication agent holds the credential (e.g. private key) used to authenticate with the target machines. When SMON peers try to deploy or restart another peer, it can automatically login into the remote machine with the help of the agent. The credential is never known to the peers or leaked out.

With the help of client utility, user can instruct SMON to manage a set of distributed applications, including other distributed management systems. In this way, the management functionalities of SMON can be extended greatly.

2.1 Self-management

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

Self-deployment

For 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[6].

In detail, a peer P will periodically choose a random machine M from its membership list and sends 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 login into M with help of the authentication agent (described in 2.2), then it copies an installation package of SMON peer to M, starts the package remotely and logout M. 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 logins 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.

Because 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 facilities (e.g. lock file). So only the first installation process 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 the evaluation, it is shown that the overhead is low on most machines. 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-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 in configuration file with the peer. A peer will exchange and compare its version with other epidemically. If there is a difference, the peer with lower version will retrieve the installation package from the other 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, the peer of new version 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 2.5.

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 epidemic pings and they will try to recover the failed peer by restarting it remotely, with help from authentication agent. 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 eventually converges to the consistent state regarding to the version of SMON peers. When two partitions rejoin, the peers in different part will be able to contact and communicate with each other, leading to the convergence of the whole system.

Combine them together

A SMON peer monitors and maintains other ones by combining above-mentioned three functionalities together.

Two points are worth mentioning. First, as an optimization, the version number of SMON is piggybacked in ping/pong messages among peers. Second, when peer A considers another one B as failed, there are two possible reasons: no SMON peer is installed or B is stopped. A will restart B directly if SMON is already installed. In this way, a peer can choose which action to take based on result of a ping message.

Disable/enable self-management

We need mechanism to disable self-management functionality of SMON to stop it from running. When self-management is enabled, we cannot stop SMON by shutdown peers one by one. If a peer is stopped by user, it will be started 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 monitoring 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. We can now stop SMON peers one by one.

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

2.2 Security mechanism

SMON should have security mechanism to achieve two goals:

- Help peers to authenticate with remote machines automatically so as to copy installation packages and execute commands remotely.
- Authenticate and encrypt access to SMON peers so it will not be misused (e.g. deploying malwares).

We describe how we design security mechanisms to achieve the two goals on Planet-Lab platform and explain how to apply the mechanisms 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 includes an separate authentication agent who holds user's private key to help peers login into other machines automatically, while user's private key is kept secret. 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 challenges from sshd to the agent and the agent will solve the challenges and send the result back. 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 among all the SMON peers and the authentication agent. It is used to authenticate 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. Note that K_E is never leaked out of network during deployment. K_E is safe to be stored within local machines because the storage resource among virtual machines are well isolated.

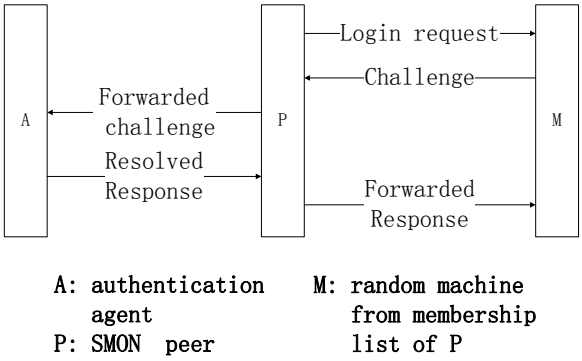

Figure 2: Interactions among SMON peer, authentication agent and a remote machine when the peer login into the machine.

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 security mechanism can be applies the distributed platforms fulfilling two requirements:

- 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.

2.3 Membership

The membership maintenance of SMON system is different from other distributed systems. While others maintains a list of peers that are currently running, SMON maintain a list 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 that SMON need to 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 version 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 add 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 stop all the epidemic activities (including monitoring and maintaining other peer, updating membership list and managing applications described below) and just replies to incoming messages. In this way, it helps spreading membership changes. After most peers update their membership list, the peers in removed machines will kill themselves after they haven't received any messages for enough long time.

2.4 Application management

One or more distributed applications can be managed by SMON. Each SMON peer maintains the peer (or client) of the application that run in the same machine. 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 undeployed application, it will try to retrieve the installation package from A and deploy the application.

After application is deployed and started, it is monitored and maintained by SMON peer on the same machine. The application has two states: online or offline. User can specify what state the application should keep on each machine. The choices 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 will report the application states to a central server timely. And whenever the application state changes, the SMON peer will report the changes immediately.

It is obvious that the application monitoring and maintenance semantic provided by SMON is very basic. It only considers whether an application is running, without further monitoring liveness or safety states. This is enough for maintaining long running services whose peers should be running as long as possible.

If more management semantic or features are required, user can first deploy another application management or monitoring system on top of SMON. In this way, SMON's functionalities can be extend greatly. For example, D3S [11] is a monitoring tool which uses instrumentation [9] to monitor application's internal states. It calculates user specified predicates on global snapshots of application states and reports when predicates are false. User can delay D3S on top of SMON and define safety or liveness predicates to check application states in detail.

2.5 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 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.

3 Implementation

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

The epidemic activities 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 with peers and agents are implemented as RPCs, and they are summarized in table 1.

SMON peer copies installation packages or executes commands by spawning a ssh/scp process. 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 1.

The parameters used at SMON runtime is stored in a configuration file. It stores the time intervals among consecutive epidemic activities, the version numbers for SMON peer and membership list, the <livetag, version> tuple, the address of authentication agent. Each application can specify an address to which the application's states changes should be reported. The configuration file is implemented as a SQLite database file to avoid data loss at machine crash. It is updated by installation package of SMON and applications. The membership list is stored separately in compressed format.

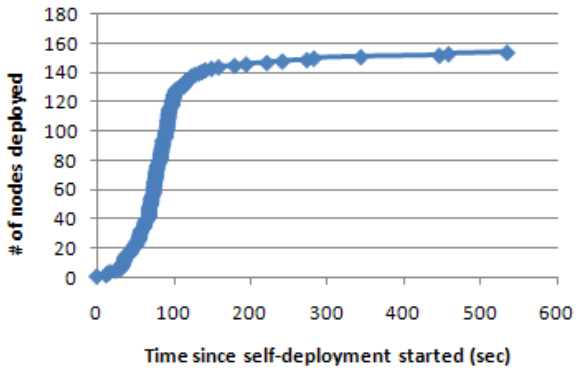

Figure 3: Progress of self-deployment process on 159 nodes.

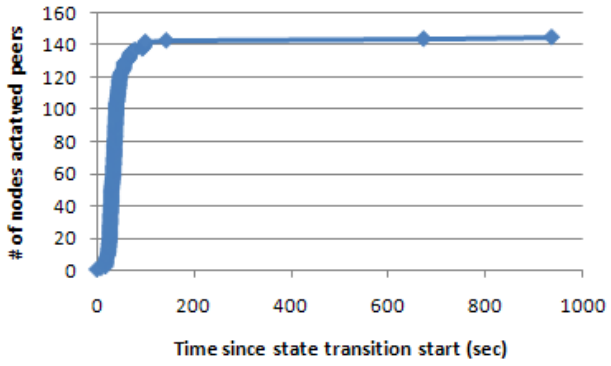

Figure 4: Progress of state transition of self-management from disabled to enabled.

4 Evaluation

We conducted experiments on Planet-Lab platform to evaluate SMON system. Planet-Lab is a global research network consisting of 900+ nodes at 400+ sites around the world.

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 3 shows the progress of self-deployment. We 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. The long tail in figure 3 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 Planet-Lab and then enable it again. The progress of state transition from disabled to enabled is shown in Figure 4. 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 2 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.

During the self-deployment process, there are cases that

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)`	Set application status (online, offline, ignore).
`get_app_status()`	Get application status.

Table 1: RPC interfaces of SMON peer and authentication agent

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

Table 2: Comparison of self-deployment process at different scales in seconds.

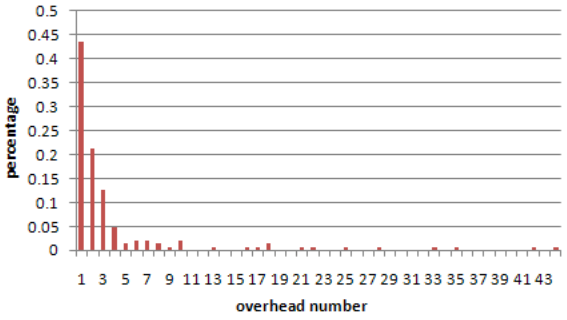

Figure 5: Overhead introduced by race condition during self-deployment process

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 5. 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.

5 Related Work

Many systems have been proposed to help managing distributed applications. Plush [1] is a management system 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 [10], but the management of Plush clients is still in centralized approach. SMON addresses the self-management property of distributed systems. It can be used to manage other management systems efficiently.

Researchers have been studying self-* properties of distributed systems for long time. Self-stabilization [5] is a concept of fault-tolerance in distributed computing. A self-stabilization system will converge to a legitimate state from any state. DHTs [14, 12, 13] are distributed systems that have self-organizing properties. They will automatically organize their overlay topologies conforming to predefined constraints under changing network conditions. [16] 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 is more stable and efficient with little human management efforts.

6 Conclusion

In this paper, we descibed 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 evaluation, it is shown that SMON achieves good performance and scalability.

References

[1] 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.

[2] 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.

[3] 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*, page 8, Washington, DC, USA, 2002. IEEE Computer Society.

[4] 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.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[6] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, 2004.

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

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[9] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[10] 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.

[11] 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*, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[12] 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*, pages 161–172, New York, NY, USA, 2001. ACM.

[13] 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.

[14] 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*, pages 149–160, New York, NY, USA, 2001. ACM.

[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] 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.