Massive Deployment of Management Agents in Virtual Data Centers

Sven Graupner, Vadim Kotov, Holger Trinks
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-321
December 7th , 2001*

E-mail: {sven_graupner, vadim_kotov, holger_trinks} @hp.com

virtual data center, management system, management agent, control system, deployment

The starting point for this work was: What would be the implications when systems and with them their management systems would become two orders of magnitude larger then they are today? We focus on one problem: How management agents can be automatically deployed in massive amounts (>>10^4) providing the infrastructure for management and control systems of large-scale virtual data center environments? Management agents perform tasks such as monitoring, processing and dissemination of management and control data, decision-making and actuation. Under deployment we understand the processes of software distribution, installation and configuration assuming a hierarchical organization of agents.

Three main areas are discussed: a deployment infrastructure with its components, the automated deployment process with protocols and a recursive generation method which allows the deployment of massive amounts of management agents. The principles presented in this paper can be generalized for common services.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2001

Massive Deployment of Management Agents in Virtual Data Centers

Sven Graupner, Vadim Kotov, Holger Trinks
Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto, CA 94304
{sven_graupner,vadim_kotov,holger_trinks}@hp.com

Keywords: virtual data center, management system, management agents, managents, deployment, control system, planetary-scale computing.

Abstract

The starting point for this work was: What would be the implications when systems and with them their management systems would become two orders of magnitude larger then they are today? We focus on one problem: How management agents can be automatically deployed in massive amounts (>>10^4) providing the infrastructure for management and control systems of large-scale virtual data center environments? Management agents perform tasks such as monitoring, processing and dissemination of management and control data, decision-making and actuation. Under deployment we understand the processes of software distribution, installation and configuration assuming a hierarchical organization of agents.

Three main areas are discussed: a deployment infrastructure with its components, the automated deployment process with protocols and a recursive generation method which allows the deployment of massive amounts of management agents. The principles presented in this paper can be generalized for common services.

1 Introduction

Data centers have been and will continue providing the backbone for services in the Internet and in enterprise environments. With continuing growth of Internet-based services, backbone data centers are expected to grow as well. We anticipate significant growth in storage and processing capacities by at least two orders of magnitude in the foreseeable future. Further more, data centers will evolve from rather separated islands hosting applications to platforms providing capacity for virtual service environments potentially distributed across several data center locations. We summarize this trend under *Virtual Data Centers* [1]. Virtual service environments are comprised by the total virtualization of resource capacities and the virtualization of service execution environments leading to an isolation of applications from the actual machines where applications are running. Isolation means that applications are not installed and operated on individual machines and thus are bound to those machines, but are installed and operated in virtual environments, which are mapped onto underlying data center resource infrastructures in a programmable manner enabling dynamic resource provisioning for fluctuating demands [2], [3]. Virtual environments may be dispersed across several

organizational or geographic regions. Another trend is the need (caused by scale) for raising the levels of consideration and abstraction from hardware-oriented resource and component views (processing, storage, network) to higher, service-oriented notions of services providing capacities to meet other services' demands throughout all system layers [4].

These trends will have impact on how data centers, resources, applications and services will be operated, controlled and managed. We summarize these trends under *planetary-scale computing*:

- *scale* – two orders of magnitude more components, resources, services;
- *virtualization* – isolation of applications, services and data from machines and equipment by introducing a dynamic mapping capability;
- *raising the level of consideration and abstraction* – component views need to be evolved into uniform service-oriented views throughout system layers, including hardware and software; control will be driven by keeping demands and capacities in balance throughout all areas and layers of systems [5].

The implications for data center management and control systems are:

- control systems need to deal with scale – two orders of magnitude more elements and components, which need to be observed, monitored and controlled,
- controlling much more components implies further automation of management and control operations,
- the degree of automated decision-making needs significantly be extended,
- control systems need to be able to operate in virtual environments crossing geographic and organizational boundaries;
- control systems need to coordinate and control virtualization,
- radical simplification and stratification of abstractions and control operations by raising the granularity and abstraction of controlled units,
- tolerating partial failure by moving from deterministic to statistical operation.

Management and control systems will be very large with $>>10^4$ components and relationships. Hierarchies with domain structures have proven to scale well as organizational method. We do not anticipate a global singular hierarchy, rather loosely coupled federations of large hierarchies, which will be distributed across various geographic and organizational boundaries.

In this report, we focus on one aspect: the automated deployment of massive amounts of management agents we refer to as *managents*. Managents are local representatives of a management system in a managed system performing tasks of monitoring, data collection, event detection, data dissemination [6], local decision-making and actuating decisions. Managents are responsible for a certain scope of components in the controlled system. Managents are organized in a communication topology among themselves mostly formed as a hierarchy. Leaf-node managents are performing direct monitoring and actuation tasks in the controlled system and higher-ordered, non-leaf managents provide a

reporting and delegation hierarchy connecting the managent hierarchy to other parts of the control system such as administrator consoles, repositories, and instances for decision-making [7], [8].

1.1 Impact of Scale on Systems of Management Agents

Managents and managent hierarchies exist today in management systems. They exist in form of sensors and management servers such as the Network Node Manager [9] in HP's OpenView product [10]. Managents are mostly implemented in software, which needs to be distributed, installed and configured on destination systems.

We understand as the *deployment of managents* the process of distributing software and the installation, configuration and activation of managent instances. Today, a high degree of manual work is involved in the deployment process of managents. For example, management servers need to be manually configured in order to establish a reporting hierarchy. This is not a problem in a deployment environment with 50 or 100 management servers. Anticipating two orders of magnitude more management servers (translating into 5,000 or 10,000 management servers in this example), this is becoming an issue. Software distribution, installation and configuration for managents as well as maintenance of existing managent instances are becoming a problem.

We propose an infrastructure for the automated deployment of massive amounts of managents comprising software distribution, installation, configuration and activation of large-scale managent hierarchies. We also address maintenance by automating re-configuration and re-deployment of managent instances. The proposed solution has been built as a prototype and operated in a 524-node Linux cluster [11].

The proposed solution addresses the following questions:

1. How to **deploy** massive numbers of managents?
    - How to distribute managent software among hosting systems?
    - How to install managent software on each hosting system?
    - How to configure large numbers of managents?
2. How to **organize** large numbers of managents?
    - How to provide managents with their roles and functions?
    - How to provide managents with their identity they use for communicating with others and to obtain proper configuration information?
    - How to establish and where to place managents in a topology?
3. How to **maintain and control** managents in large topology?

Maintenance and control refers to operations such as 'start' and 'stop' as well as 'reconfigure', 'redeploy' and 'uninstall' applied on active managent instances.

Viewing managent deployment and maintenance processes under the perspective of massive amounts of managents, the following questions arise:

- How to deal with numerous sets of configuration data for large numbers of managents and how to organize a topology among managents without having to configure them individually?
- How can be avoided that each individual managent needs to be configured to which supervisor managent it is supposed to report?
- How to establish a topology among managents in general? Can this topology be established automatically?
- How managents can obtain an identity and a role independently of their physical location provided by the local hostname or IP address in order to work in virtual environments? This is particularly important when management or control systems span multiple virtual LAN environments, as it is the case in virtual data center environments.
- How can be avoided that each individual managent installation and configuration process needs to be initiated separately by a human operator?

1.2 Structure of this Report

The report is structured as follows. We start with a brief comparison with the state of the art in system management and control systems regarding two common technologies: software distribution and configuration engines. We explain why these do not provide sufficient support for the scenario described above. We then state assumptions we make before we subsequently explain our approach in detail.

In section 2, we start with the description of the managent deployment infrastructure explaining the basic components such as boot stubs, repositories, directories, and initiators. In section 3, we explain the automated process of managent deployment with state machines defined for boot stubs and managents followed by a detailed description of related protocols in section 4. In the following section 5, we explain how this infrastructure is used to deploy massive amounts of managents in a controlled, recursive process. The main innovations are: the automatic recursive deployment process (recursively means that deployed managent instances may be assigned roles to further deploy sub-instances), controlling this process by matching very dense descriptions maintained in repositories, the automatic assignment of identity to all generated managent instances independently of the underlying communication infrastructure (no IP addresses or hostnames are used), and the proper configuration of all managent instances providing them with their roles.

1.3 Comparison With the State of the Art

Remote software installation and configuration systems exist today. Configuration engines provide the centralization of configuration information and centralized interfaces for operators to perform remote configuration tasks in a distributed environment. An example of a configuration engine is cfengine [12] used in distributed Linux environments. Another example is to some extend HP's itsm change & configuration manager [13]. Configuration engines by themselves do not provide remote software installation. They provide support for centralization of configuration information and

remote execution of configuration operations. They assume complete configuration information in their databases.

Installation and configuration systems are also not designed with very large amounts of entities in mind. Configuration engines are "operator-driven", operators need to initiate management operations. They do not support full automation. They usually provide remote configuration based on remote execution of commands summarized in scripts. The engine knows all locations where managed software is installed and also configures each individual instance according to the information maintained in the engine's configuration base. Each installation and configuration must be completely pre-configured in the engine. Configuration engines are not designed to install and configure whole topologies as one controlled process described in a dense format. Configuration engines also do not facilitate automatic assignment of identity and roles to individual instances. All these reasons make clear that configuration engines need to be extended in order to address these issues stated at the beginning.

Another shortcoming is that each instance requires individual configuration information, which is difficult to maintain when the number of instances increases dramatically. It is no longer affordable to maintain individual configuration information for each individual instance. Our proposal allows summarizing configuration information for whole (assuming large) groups of managent instances in singular, condensed descriptions exploiting some managent properties and making some assumptions.

Another principle of configuration engines is that they would perform deployment as one flat process. We propose the deployment of "deployment engines", managents with dedicated roles of initiating deployment processes of sub-ordered managents, which may themselves be assigned with roles to initiate further deployment processes leading to a recursive, decentralized deployment process spawning a potentially large hierarchy.

1.4 Assumptions

We make several assumptions for our approach.

- We assume a *high degree of similarity* among groups of managent instances. This means that software packages for managents are widely shared, even when managents perform different functions. Managents obtain their specific roles during configuration implying that managent roles can be reduced to configuration data. We assume that only few software packages need to be maintained for large amounts of managents.
- We assume some protocol handler does exist on hosting systems for handling the initiating deployment protocol (BIP, the Boot Stub-Initiator-Protocol, see section 4.1). This protocol handler is referred to as *Boot Stub*. It is the least common functionality we assume on potential managent hosting systems. This assumption is comparable to the need for the BOOTP protocol handler required for bootstrapping in diskless workstations [14]. There might be multiple incarnations of Boot Stubs: they might be realized as processes launched by inetd, or they might be activated during the machine's bootstrap.

- We assume a *hierarchical organization* of the managent topology. This assumption is caused by the recursive deployment process for large amounts of managents described in section 5.
- Since we aim to avoid maintaining large amounts of individual configuration data for large amounts of managent instances, we assume a high degree of similarity in configuration data among managents as well. The more similarity exists, the more configuration data can be shared and do not need to be maintained separately for different sets of managents. Ideally, the only different property among all managent configurations is their identity tag that is generated automatically during the deployment process. The other extreme case is that each deployed managent instance requires an own set of configuration data, which needs to be maintained separately. In this case, large amounts of managents cause large amounts of configuration data. No benefits of sharing and compact notations can be exploited in this case.

2 Deployment Infrastructure

Automatic deployment[1] of managents includes the following steps:

- downloading the managent installation code,
- downloading configuration data for the installation process and for the managent,
- launching the installation code with its specific installation configuration data,
- activating the managent with its specific managent configuration data.

The infrastructure that supports this consists of the following components (see Figure 1):

- An underlying topology of protocol handlers called **boot stubs** on machines capable of hosting managents,
- **Repositories** containing information about installation code (boot code) and configuration data for managents,
- **Directories** containing topology models defining where boot stubs and managents are physically installed, and
- **Initiators** controlling the deployment and (re-)configuration of managents. Initiators may be management applications or may be managents themselves assigned with roles of deploying and controlling other managents.

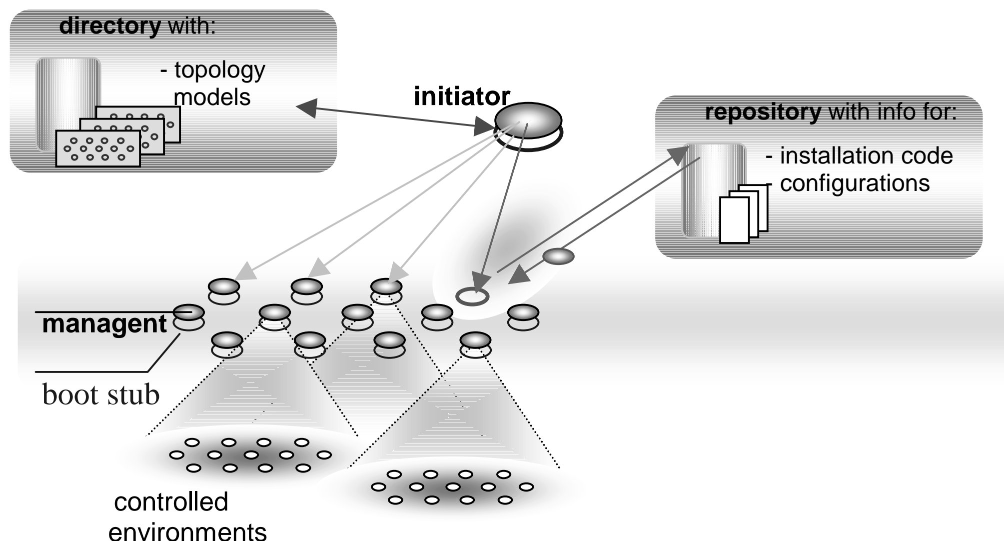

Figure 1: Infrastructure components for the deployment of managents.

[1] sometimes also referred to as "bootstrapping" which gave the initial name for the "boot stub".

2.1 Topology of Boot Stubs

The existence of a topology of boot stubs is assumed in an environment where managents will be deployed. Boot stubs are capable of receiving control messages from the initiator requesting the deployment of a managent on this location or requesting other control actions. Boot stubs may be implemented in different ways: as servlets behind web servers or as daemon processes launched on an initiator's request received on a certain port, through inetd on Unix, for example. It is assumed that the boot stub is running after its hosting machine has been booted. This is a valid assumption as the boot stub is a generic program that does not need any host-specific configuration. The boot stub could also be part of the boot image of the hosting machine. The boot stub's internal structure and its external protocols are described in sections 3 and 4.

2.2 Repositories

Repositories contain the information needed to control the deployment process of managents (software distribution, installation, configuration). For managents, three kinds of information are maintained:

1. *References to bootable managent installation code:* This is the code booted for a managent. Managent code will be shared for managents that have similar roles. Platform or role-specific differences should be controlled through configuration data.
2. *References to configuration data for the installation code:* This configuration data is passed to the managent installation code when it is launched. It can be used to control platform or role-specific differences during the installation process such as setting or testing specific environment variables, creating directories, etc.
3. *References to configuration data for managents:* This configuration data is passed to the managent when it is launched. It should contain information that is specific for a particular managent instance, e.g. references to managed objects, references to other managents in the control infrastructure (parent, children), interval times for status reports, etc.

Since we assume a high degree of sharing of code and configuration data among managents, we store references to these data in three separate lists in repositories each list containing entries for sets of managents sharing particular information. The selection of individual entries from the three lists for a particular managent deployment is based on matching the managent's unique identifier represented as a path expression (a node in a hierarchy can uniquely be identified within this hierarchy by the path starting from the top node; see section 5.2 how these path identifiers are automatically created and assigned to managents during the recursive deployment process). Path identifiers represent the position of a managent in a managent hierarchy.

The repository matches managent's path identifiers presented for a particular deployment against patterns defined for entries in each of the three lists. All references are returned from all three lists for which matches have occurred. This provides a fine-grained mechanism for sharing individual configurations from each list independently. Managents may share code, but may not necessarily share installation or managent configurations or vice versa.

The more information can be shared, the less entries in the three lists are needed and the more compact descriptions can be provided. Similarity and sharing code as well as configuration data is an important condition we assume. In the extreme case that nothing can be shared, each individual managent would require its own entry in each of the three lists. The number of list entries would be equivalent to the number of managents, a situation we want to avoid in order to effectively deal with large amounts of managents.

The repository contains *references* to bootable code and configuration data. The actual code and configuration data may be stored somewhere, e.g. code at a web site of a company providing managent software and configuration data may be obtained from another company's web site providing management services for customer environments. Managent code and configuration data may also be stored in the same repository, then just referenced by the address of that repository.

After the boot stub has sent a request with the managent's path identifier, this identifier is sequentially matched against patterns of all entries in each list. References to code or configuration data of all entries with matching patterns will be returned. The boot stub then follows these references in order to obtain actual code and configuration data.

```
<repository-information>

  <managent-code> <!— two patterns match two shared code references -->
    <entry>
        <pattern path="*/dc_paloalto/*"/>
        <pattern path="*/dc_berlin/*"/>
        <ref url="http://sos.hpl.hp.com/repository/dc_all.sh"/>
    </entry>
    ...
  </managent-code>

  <install-config >  <!— two patterns match two different config sets -->
    <entry>
        <pattern path="*/cluster*/*">
        <ref url="http://sos.hpl.hp.com/repository/cluster_all.xml">
    </entry>
    <entry>
        <pattern path="*/SAP/*">
        <ref url="http://config.sap.com/repository/proccfg.xml">
    </entry>
    ...
  </install-config>

  <managent-config>
    <entry>
        <pattern path="*/customer_x/*"/>
        <ref url="http://sos.hpl.hp.com/repository/customer_x_all.xml">
    </entry>
    ...
  </managent-config >

</repository-information>
```

Figure 2: Deployment information maintained in a repository.

Figure 2 shows an example. A boot stub deploying a managent with a path identifier:

`/hp_mgmt_domain/dc_paloalto/cluster_21/customer_x`

would be provided with references for:

- the installation code (matches '`*/dc_paloalto/*`'),
- the installation configuration information (matches '`*/cluster*/*`'), and
- the management configuration information (matches '`*/customer_x/*`').

The boot stub can be implemented in a way that it can launch managent installation code of different formats, e.g. java classes, scripts or executable binaries according to the code obtained from code references. Configuration data are stored in form of XML documents. An example is shown in Figure 3. It contains configuration data obtained from a reference for the managent installation. When the installation code is launched, the boot stub provides it with the appropriate environment (`<env-params>`) and command-line (`<arg-params>`) parameters. In the same way, managent configuration data are provided when the actual managent program code is started.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <env-params>
        <env name="TMP_DIR" value="/tmp"/>
        ...
    </env-params>

    <arg-params>
        <arg name="MGMT_PORT" value="7078"/>
        ...
    </arg-params>
</configuration>
```

Figure 3: Example configuration data for managent installation code.

2.3 Directories

Directories contain topology models that describe where boot stubs and managents are currently installed in the system. Directories form the logical container for topology information. Physically, they may reside within repositories (see section 2.2).

A *boot stub topology model* contains information about where boot stubs are located in an environment such that an initiator can contact them and initiate the deployment of managents. The boot stub topology model can initially be obtained in several ways:

- A management application could initialize the directory with all the machines in an environment that host a boot stub.
- When a machine boots, and the boot stub is initialized, it could send a registration message to a pre-configured directory service.
- When the boot stub is initialized, it could send a broadcast message to find the directory service and could then register with it.
- The directory service could periodically send broadcast messages within its domain to find existing boot stubs. Boot stubs reply to this broadcast with a registration message to the directory.

The *managent topology model* reflects all managents in an environment, which have been deployed through some initiator. The initiator registers a managent when it is booted and un-registers it when a managent becomes uninstalled.

2.4 Initiators

Any management application can act as an initiator. Or initiators can be exposed managents in the system whose purpose is to initiate the deployment of other managents. Initiators can also be implemented specifically for this purpose. Even human initiation shall be allowed, for instance to trigger the top-most initiator.

Based on information of the boot stub topology model, the initiator is capable of initiating deployment processes on potentially all boot stubs[2] in its environment.

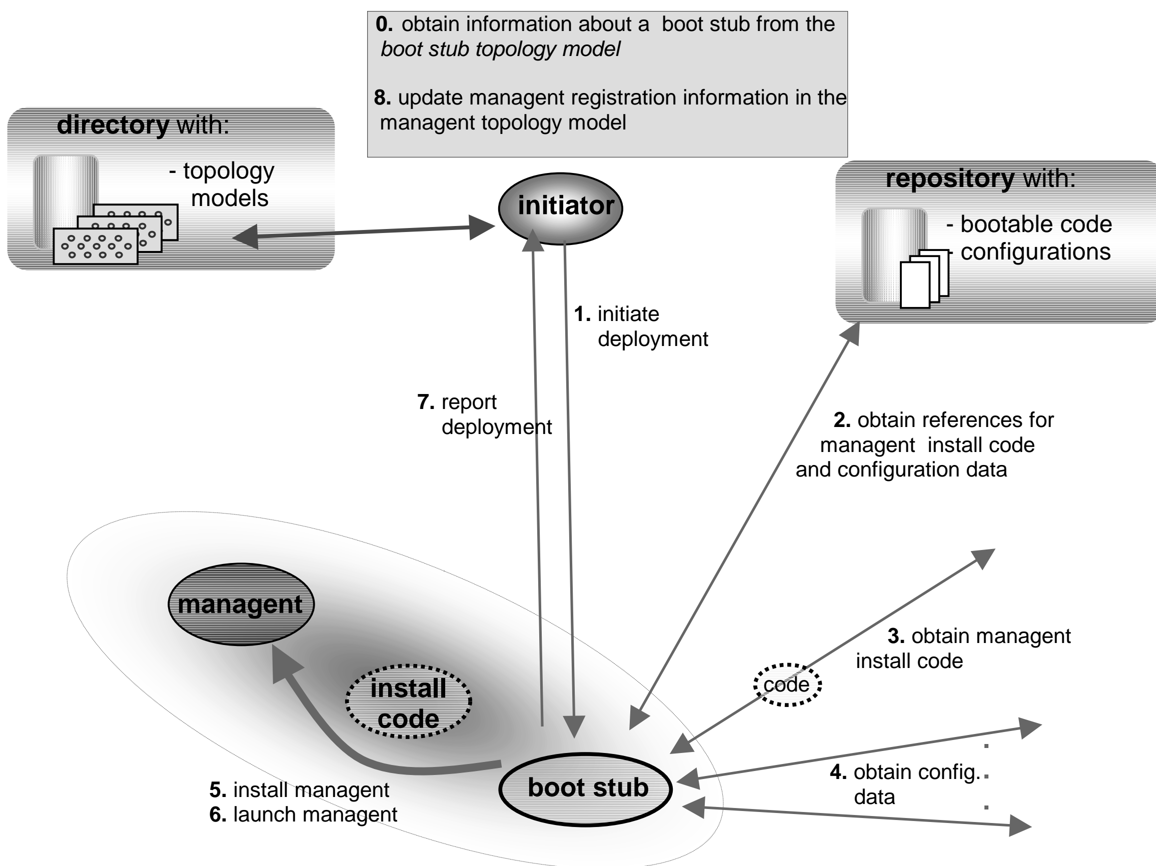

Figure 4: Deployment processes of a managent with obtaining software and configuring the managent.

[2] Some more sophisticated policy may further refine this deployment.

3 Deployment Process

The deployment process is sketched in Figure 4 and consists of the following steps:

1. Based on the information of the boot stub topology model, the initiator sends a message to each boot stub in its domain instructing it to initiate the deployment process.
2. The boot stub subsequently communicates with the repository in order to obtain references to the bootable code and configuration data based on the identity of the managent to be deployed.
3. The bootable installation code is obtained from the referenced location.
4. Configuration data are obtained from the referenced locations. It can consist of several XML documents from different locations.
5. The boot stub runs the installation code and provides it with the given installation configuration data. After completion of this step, all necessary managent software is installed and configured.
6. The boot stub launches the managent and provides it with its configuration data.
7. The successful deployment of the managent is reported to the initiator.
8. The initiator updates the managent topology model in the directory to reflect the new managent in the domain.

On request of the initiator, existing managents can be reconfigured after they have been deployed. Two basic means exist:

- *Managent configuration* meaning that the boot stub basically modifies configuration parameters of managents and triggers a managent to restart with the new configuration settings.
- *Managent environment configuration* means that the boot stubs terminates the managent, obtains a new configuration (maybe even new boot code) and re-runs the managent installation code with new configuration settings.

More details can be found in the next section that describes the boot stub.

3.1 The Boot Stub

The boot stub is a protocol handler that runs on all machines that might need to host a managent. It is capable of receiving control messages from some initiator that either requests deployment of a managent or some other control commands.

The boot stub can be described by its internal behavior and its external interfaces defined by protocols. It communicates with three different entities:

- The **initiator** that sends control commands to the boot stub. An initiator could be some management application or another managent that recursively deploys and configures other managents.
- The **repository** that contains references to bootable managent installation code and configuration information.
- The **managent** itself, after the stub has launched it.

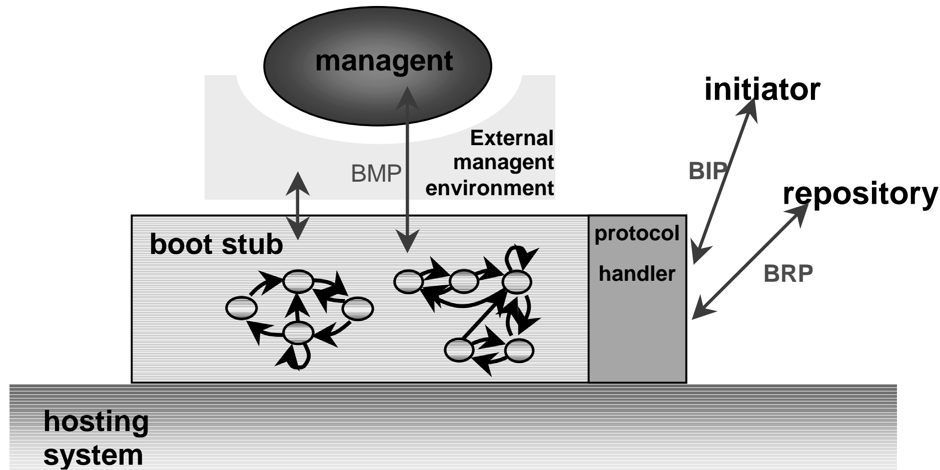

Figure 5: Structure of a boot stub with a managent.

The boot stub can have different implementations. It could be, for example, a daemon process listening on a certain port, or it could be a servlet behind a web server. It is assumed that the boot stub is running after its hosting machine was booted. We can assume this since the boot stub is a generic program that does not need any host specific configuration. The boot stub could, for example, even be part of the boot image of the hosting machine.

Following, the boot stub's internal structure and behavior will be described. The external protocols will be discussed in section 4.

Two state machines represent the boot stub's internal behavior. One controls the external managent environment and the other the managent itself.

3.2 Control of the External Managent Environment

The external managent environment comprises all the installation code, installation configuration data and actions that are necessary to deploy a managent. In general, the deployment process consists of the code distribution, configuration and installation. Figure 6 shows the states and transitions that are here discussed in more detail:

1.) *Load Install Code:* The boot stub sends a request to the repository in order to obtain references to the managent installation code (boot code) and configuration data. The repository determines the appropriate code and configuration data based on the managent's identity. The boot stub then obtains the boot code from the specified

location. The boot code could also already contain all the managent's code or it could be an installation program that obtains more code files and configuration data from further installation servers.

2.) *Configure:* As for the boot code, the repository provides references to the installation configuration data of the managent. The boot stub obtains the configuration data from the specified locations. Configuration data can, for example, include environment variables and initialization parameters that are passed to the managent installation program. It is possible to reconfigure the managent environment after the installation process. This implies that the boot stub requests new references for the configuration data from the repository, downloads the actual data and then needs to re-run the installation program with the new configuration.

3.) *Install:* The boot stub runs the managent installation program with the given configuration data. The installation process can include typical actions such as creating directories, updating application configuration files, downloading more program files or libraries, etc.

4.) *Uninstall:* The uninstall process removes all the managent and installation program code and data from the system and reverses all actions taken during the installation process. We assume that managent software and configuration data allow clean un-installation.

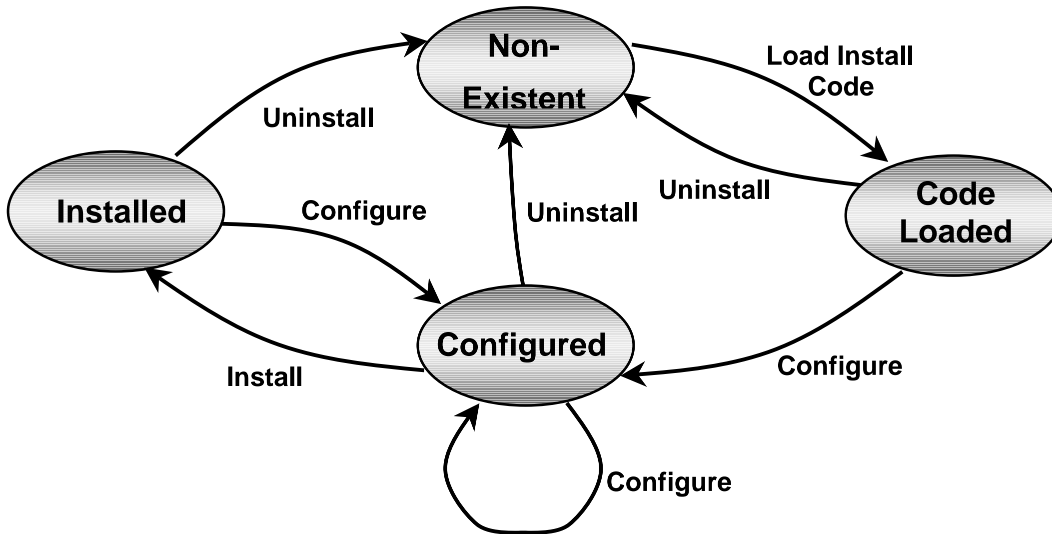

Figure 6: The external managent environment state diagram.

3.3 Control of the Managent

Figure 7 shows the state diagram of the managent itself. The managent remains in the state *non-existent* as long as the managent environment is not yet in the state *installed*. The *deploy* transition is made when the managent environment enters the *installed* state;

the *remove* transition is performed whenever the managent environment leaves the *installed* state.

Managent configuration includes the same actions as for the configuration of the installation process. References to configuration data are obtained from the repository based on the managent's identity. The actual configuration data are then obtained from the specified locations. Managent configuration data can include parameters like the port number where the managent will be listening at, identifiers and/or addresses of other managents in the control hierarchy, interval times for reporting etc. They can be passed to the managent in form of environment variables or as command line arguments passed to the managent when it is launched. Reconfigurations of the managent can also be performed at any later point in time.

Once the managent is deployed and configured, the boot stub can start the managent. The managent can be in an active or passive state. In the active state, the managent fulfills his specific management tasks. It receives and sends messages from or to other managents, sensors and management applications. The passive state is used to temporarily deactivate the managent without shutting it down.

State transitions for both, external managent environment and the managent itself, are caused by the execution of control commands received from the initiator.

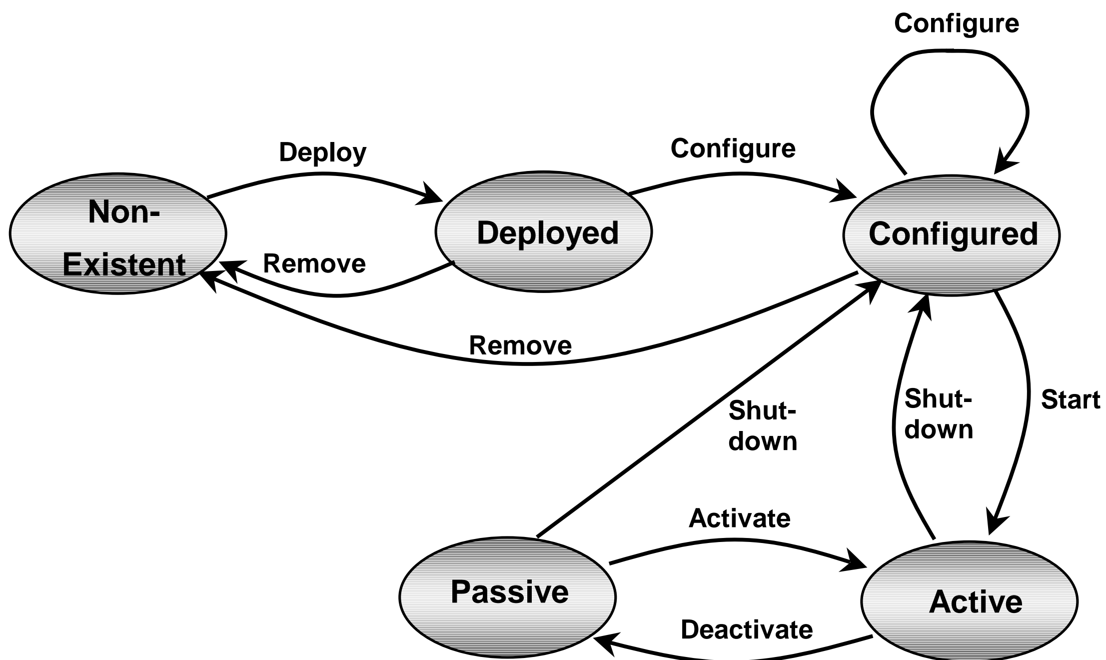

Figure 7: The managent state diagram.

4 Protocols

4.1 Boot Stub – Initiator – Protocol (BIP)

The boot stub receives messages that include control commands, which can change the state of the external managent environment or the managent itself. Control commands can be categorized into those that affect the external managent environment and those that affect the managent itself. However, there are some dependencies so that both can be affected. The first part of Table 1 describes the protocol messages for the external managent environment, the second the ones for the managent.

Control Command Requests	External Managent Environment State Machine	Managent State Machine
deploy repository-uri managent id ________________ *uri*: Universal Resource Identifier, RFC 1630 (and subsequent RFC's)	- condition: must be in state *non-existent,* - transitions: load install code, configure, install, - resulting state: *installed*	- condition: must be in state *non-existent,* - waits until external managent environment is in state *installed,* - transitions: deploy, configure, start, - resulting state: *active*
configure-env repository-uri managent id	- condition: must be in state *code loaded, configured* or *installed,* - transitions: configure, - Resulting state: *configured*	- condition: must be in state *non-existent,*
uninstall managent-id	- condition: none, - transitions: uninstall, - resulting state: *non-existent*	- condition: must be in state *non-existent, deployed* or *configured,* - transitions: remove, - state: *non-existing*
start managent-id	- condition: must be in state *installed*	- condition: must be in state *configured,* - transitions: start, - resulting state: *active*
shutdown managent-id	- condition: must be in state *installed*	- condition: must be in state *active* or *passive,* - transitions: shutdown, - resulting state: *configured*
restart managent-id	- condition: must be in state *installed*	- condition: must be in state *active* or *passive,* - transitions: shutdown, start, - resulting state: *active*

activate managent-id	- condition: must be in state *installed*	- condition: must be in *passive* state, - transitions: activate, - resulting state: *active*
deactivate managent-id	- condition: must be in state *installed*	- condition: must be in *active* state, - transitions: deactivate, - resulting state: *passive*
configure-managent repository-uri managent-id	- condition: must be in state *installed*	- condition: must be in state *deployed* or *configured,* - transitions: configure, - Resulting state: *configured*

4.2 Boot Stub – Repository – Protocol (BRP)

The protocol between the boot stub and the repository consists of the boot stub's requests for references to managent installation code and/or configuration data. Four different request types exist:

1. request managent installation code reference,
2. request managent installation configuration references,
3. request managent configuration references,
4. request all references (1. to 3.) for one managent.

An example of a request to the repository and its response is shown in Figure 8.

```
Request:

<request>
    <type>install-configuaration</type>
    <id>/dc_paloalto/customer_x/cluster_12</id>
</request>


Response:

<reply>
    <type>install-configuaration</type>
    <id>/dc_paloalto/customer_x/cluster_12</id>
    <ref type="url">http://sos.hpl.hp.com/repository/customer_x/config.xml</ref>
    <ref type="url">http://sos.hpl.hp.com/repository/dcmgmt/paloalto.xml</ref>
    <ref type="url">http://sos.hpl.hp.com/repository/clustermgmt/basic.xml</ref>
</reply>
```

Figure 8: Example of a message exchange in the Boot Stub – Managent – Protocol (BMP).

4.3 Boot Stub – Managent – Protocol (BMP)

The boot stub must know how to configure, start and shutdown a managent's program. There could be different implementations how the boot stub determines how to start the managent, for example:

- The boot stub recognizes the file format (e.g. binary executable, perl script, java program),
- the managent installation program returns a string that represents the command line that starts the managent when subsequently being executed by the boot stub,
- or the initiator explicitly tells the boot stub the start command.

Configuring the managent can be done by passing environment variables and initialization (command line) parameters to the managent program.

Additionally, the managent program must be able to receive the following control commands from the boot stub:

- *activate:* start or continue work (receiving requests, processing, sending event and decision notifications),
- *deactivate:* intercepts the managent's current work in a consistent way such that no data will be lost and work can be resumed after reactivation.

In the prototype, a the boot stub is realized as a java servlet. The communication between initiator and boot stub is based on HTTP. The message format is XML-based, files are attached as binary MIME extensions to XML messages. An example control message sent to a boot stub that initiates the deployment of a managent is:

```
<?xml version="1.0" encoding="UTF-8"?>
<header>
    <to addr="http://palmtree.hpl.hp.com:8080/cs/bootstub"/>
    <from addr="http://sos.hpl.hp.com/ControlService"/>
    ...
</header>

<?xml version="1.0" encoding="UTF-8"?>
<body>
   <cmd name="deploy">
      <repository uri="http://sos.hpl.hp.com:8080/cs/repository"/>
      <initiator uri="http://sos.hpl.hp.com/ControlService"/>
      <boot-file type="java" name="SimpleAgent.class"/>
   </cmd>
</body>
```

Figure 9: Example control message that initiates the deployment of a managent.

5 Recursive Deployment of Managents

Full automation of the managent deployment process is inevitable in order to deal with massive amounts of managents. Individual instances cannot be installed or configured manually any more. The goals of our approach is:

- total automation of the managent deployment process as a requirement to address large numbers of managents,
- compact configuration descriptions applying to whole (potentially large) sets of managents rather than individual descriptions per instance, and
- decentralization of the deployment process by deploying sub-branch “deployment managents” leading to a recursive, hierarchical structure.

In this section we explain how the mechanisms introduced before will be used to deploy large numbers of managents using a recursive method. The proposed method automates all three stages of the deployment process for managents: software distribution, installation, and configuration under some assumptions explained in section 1.4

- managents form a hierarchical topology, and
- groups of managents are “similar” to each other meaning they run the same software using the same or similar configurations. The more configuration information can be shared, the higher is the compactness in configuration descriptions.

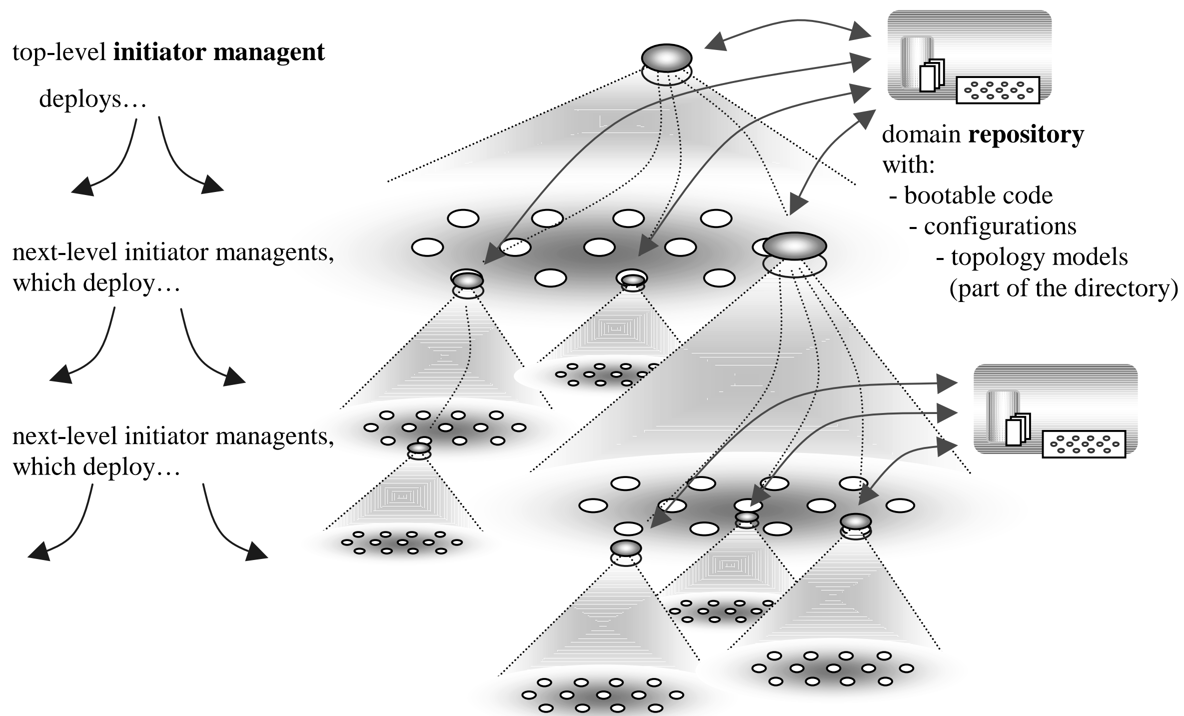

Figure 10: The recursive deployment process.

Domain repositories contain information about managents and relationships among managents in form of topology models (in the directory), references to the bootable code and respective configuration information for managents. The repository information thus entirely controls the managent deployment process. References to bootable code may include scripts of any suitable scripting language (`sh`, `perl`, `python`) or Java classes as used in the prototype. References to configuration information include environment settings, paths, command line arguments etc. Topology models maintain the information where boot stubs are residing in an environment and where managents have already been installed and started. After activation, initiators register the deployed managents with their directory. The repository information is outside and thus independent of the managed system, though repositories may well reside inside the managed systems. Repository information can thus be managed and maintained separately at a location where it is best suited. A variety of repositories can exist in order to provide means to structure configuration information and maintain it separately for different regions.

5.1 The Recursive Deployment Process

The recursive deployment process of managents is based on the classification of managents into two categories (see Figure 10):

- managents with initiator roles (non-leaf node managents) – these are managents which obtain code and configuration information during their own deployment process providing them with initiator roles for deploying sub-ordered managents, which on their part can again be assigned roles as initiators for sub-sub-ordered managents;
- managents with no initiator role (leaf-node managents) – these are typically managents performing the actual managent tasks of monitoring, data processing and dissemination, event detection, decision-making and actuation. The recursive deployment process stops here.

A hierarchy builds up during this recursive generation process controlled by providing managents at each level with appropriate configuration information. Configuration information also determines the role of either of the two categories. Controlling the overall deployment process is thus reduced to configuration information in the repositories passed to managents providing them with their roles during their deployment.

In Figure 10, the top levels of the hierarchy are controlled by information contained in the top-level repository. Each initiator managent obtains the repository reference as part of its configuration information from the parent initiator managent. A sub-ordered repository then controls lower levels of the hierarchy. It is part of the configuration information, which repository a managent will refer to for deployment and control of sub-ordered managents. This allows establishing separate repositories, see also section 5.6.

5.2 Assigning Identity to Managents During the Deployment Process

A managent's identity is the information that uniquely identifies a managent when it communicates with other instances. Management agents in existing management systems are often identified by IP addresses or hostnames of systems where they reside since the

management agent can automatically obtain this information. Another approach is to assign a unique identification string to each managent instance during deployment. This identification information can be pre-configured for each managent, or it can be generated while building the managent hierarchy.

Since we assume a hierarchy of managents, we can apply a total enumeration schema in this hierarchy in order to generate hierarchy-wide unique identification strings for each managent instance. Concatenated, these strings represent paths starting from the top level to all individual managent instances. Paths are unique for all managents, and such paths can easily be generated during the deployment process.

Initiator managents at each level of hierarchy know all their direct children. This information is passed to them as part of their deployment configuration information. Children are enumerated uniquely among all siblings within the scope of the parent managent. The initiator managent then initiates deployment processes for all children and concatenates its own identification path obtained during its deployment with respective children identifiers and passes the result to each child as identification string:

```
childPath ::= parentPath + "/" + local child enumeration.
```

Path expressions are unique within the hierarchy and can thus be used as managent identifiers. Using path expressions also provides the advantage that these strings can easily be matched in order to obtain proper configuration information from repositories as shown in the following section. Figure 11 shows an example.

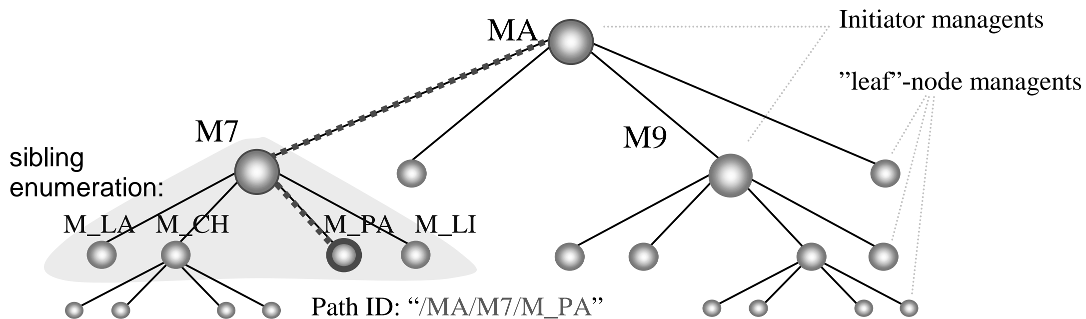

Figure 11: Generating managent identifiers based on paths in the hierarchy.

5.3 Obtaining Managent Deployment Data from Repositories

Repositories maintain the deployment information as shown in section 2.2. The boot stub receives a 'deploy' request from an initiator (see BIP in section 4.1)

initiator → boot stub:

```
deploy
    repository-uri,
    managent id.
```

The initiator passes a reference to the repository and the new generated managent identifier path to the boot stub. The boot stub then contacts the indicated repository providing the managent identifier path to the repository where it is matched against patterns in the three lists containing references to managent code, installation

configuration and managent configuration data (Figure 2 on page 9). Matching managent identification paths with patterns in the repository is based on regular expressions. References matching the managent's identifier path are returned from the repository to the boot stub. The boot stub then obtains the actual data from these references.

5.4 Controlling the Recursive Deployment Process

Managent deployment data obtained during deployment determines the role a managent will have, whether it will be assigned a role of an initiator for sub-ordered managents or it will be a leaf-node managent with no further deployments. Since the role of an initiator is entirely reduced to managent's configuration data, its control is also provided by configuration data. The recursion stops at a hierarchy level where no configuration data for initiator managents is obtained.

The termination of the recursion is defined by patterns of entries, which do not represent further initiator roles in repositories (see Figure 12). Patterns are matched against managent identifier strings growing through the levels of hierarchy. This ensures that the recursion can be controlled entirely through patterns and can be terminated at designated points indicated by patterns. Multiple levels of initiator managents can easily be described by patterns like:

```
<repository-information>

  ...

  <managent-config>
    <entry>                <!—termination of recursion -->
        <pattern xpath="/*/*/*/*/*"/>    <!—level 5 -->
        <ref url="http://sos/regular_managent_config.xml">
    </entry>
    <entry>                    <!—recursive generation -->
        <pattern xpath="/*/*/*/*"/>      <!—level 4 -->
        <pattern xpath="/*/*/*"/>        <!—level 3 -->
        <pattern xpath="/*/*"/>          <!—level 2 -->
        <pattern xpath="/*"/>            <!—level 1 -->
        <ref url="http://sos/initiator_config.xml">
    </entry>
  </managent-config >

</repository-information>
```

Figure 12: Example for controlling a recursive deployment process.

Figure 12 shows an example of a hierarchy growing across four levels of initiator managents. The recursion terminates when at level five a regular (non-initiator) configuration is matched. Assuming that each initiator managent deploys 10 child managents each, the shown description would spawn a five-level, symmetric hierarchy of altogether 10k managents (1+10+100+1000+10000).

Managent identifier paths growing through the levels of hierarchy could be:

initiator managent at level 1:	"/region1"
initiator managent at level 2:	"/region1/zone3"
initiator managent at level 3:	"/region1/zone3/dc6"

initiator managent at level 4: “/region1/zone3/dc6/core7”
regular managent at level 5: “/region1/zone3/dc6/core7/sos.hpl.hp.com”

with the level 5 managent identifier path matching the termination pattern.

When differences in configuration data among levels or even among managents within one level exist, these differences would reflect in separate entries in the repository information with patterns matching only the appropriate managent identifier paths.

The proposed repository description is very effective for managent hierarchies with a high degree of similarity and symmetry. The more configurations become different, the more extensive the description will become. It is our belief that very-large scale systems will have a high degree of similarity and symmetry in order to be controllable and maintainable.

5.5 Parallel Deployment Processes

Each initiator managent initiates (spawns) a deployment process for child managents, which on their part may spawn further deployment processes. All branches starting from each initiator managent are independent of each other and can thus proceed in parallel.

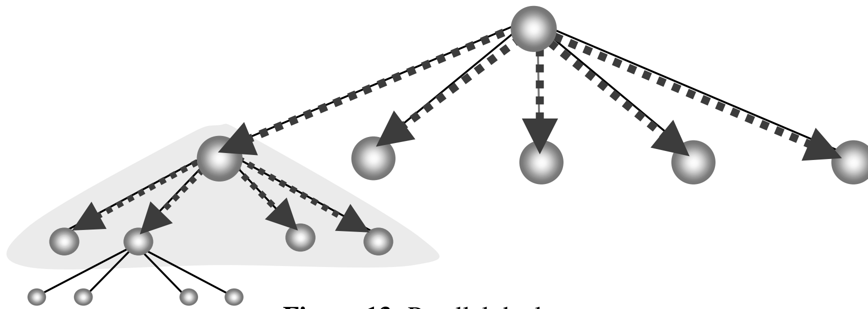

Figure 13: Parallel deployment processes.

5.6 Domain Repositories

The information from which repository a boot stub will obtain the deployment data for a managent is part of the initial deploy request (see section 4.1):

initiator → boot stub:
```
deploy
    repository-uri,
    managent id.
```

The reference to a repository issued with a deploy request is part of an initiator’s own set of configuration data. References to multiple repositories can be configured. This allows to organized configuration information in separate, domain-specific repositories.

Figure 14 shows an environment where two repositories are used. The top-level repository maintains the configuration information for the two top-level initiator managents (MA and M7 in the figure). The managent M-PA, sub-ordered to M7, then refers to its own domain repository. This selection was made based on configuration data passed to initiator managent M7, that includes also the reference to the repository used

for further deployed sub-managents, in order to initiate a deployment of managent M-PA using the sub-domain repository.

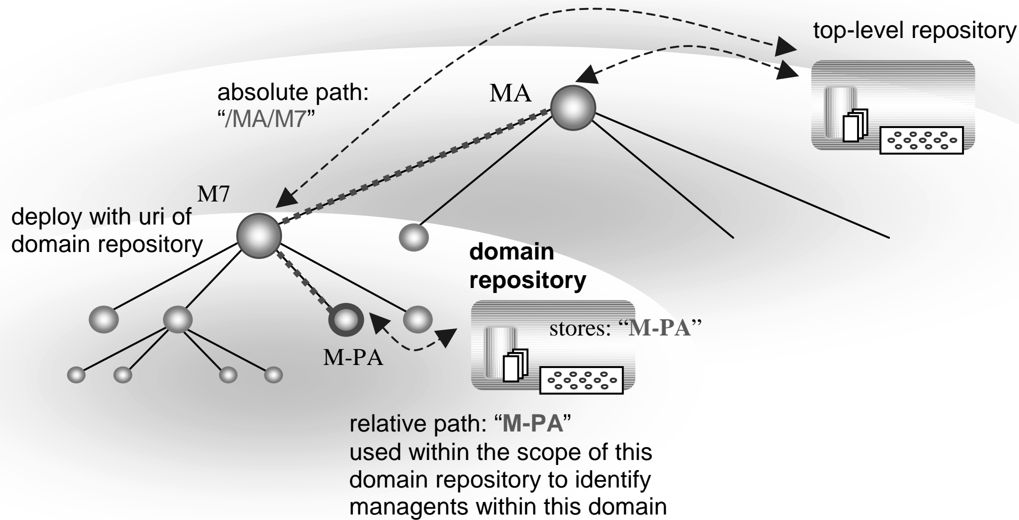

Figure 14: Domain repositories.

Managent identification is always based on full paths through the hierarchy. In order to avoid dependencies from the overall hierarchy occurring in domain repositories, identifiers and patterns are used relatively to the scope of the domain repository. This means that the full path of managent M-PA

/MA/M7/M-PA

is cut by the path of the position of the domain repository in the hierarchy indicated by /MA/M7. The actual identifier of this managent within the scope of the sub-domain repository then is

/M-PA.

Relative paths can only be used when domain repositories fit into the hierarchy that can be specified by a path such as /MA/M7 in the figure. Otherwise domain repositories must be managed with full path identifiers.

6 Summary

In this report, we propose a schema for the automated deployment of massive amounts of management agents (managents) used in next-generation data center management and control systems anticipating that these environments will become two orders of magnitude larger than they are today and will be characterized by the total virtualization of resources, platforms and execution environments. From this space, we have addressed the problem of how massive amounts of managents can be deployed, a process that includes the distribution of software for managents and the installation and configuration of this software. We compare our approach with configuration engines used today for managing software and configurations in larger environments. Shortcomings are outlined and addressed by our solution.

The *infrastructure* consists of so-called boot stubs responsible for handling the initial deployment protocols, repositories forming the information base for references to all needed deployment and configuration data, and directories maintaining topology models, which is the information about where boot stubs reside in a system and where existing managent instances are registered. The *deployment process* for individual managent instances is described in detail in form of state diagrams and three protocols: BIP, BRP and BMP. We assume that managents will be organized in hierarchies, and we exploit this property to automatically generate and assign unique identifiers to managent instances during their deployment.

In section 5, the *recursive deployment process* for massive amounts of managents is then presented built on the primary mechanisms of the deployment infrastructure to deploy theoretically unlimited amounts of managents forming hierarchies. It is explained how this process is controlled and how the recursion is terminated. Control, configuration and termination is described in a very dense format using patterns formulated as regular expressions that are matched against path identifiers of individual managents in order to obtain the proper deployment information for them. We assume large groups of managents and whole hierarchies of them will share software and configuration data.

Advantages of the proposed solution:

The proposed solution overcomes problems by automatically assigning identifiers to managent instances based on their position in the managent topology rather than using their current physical hosting address. We thus achieve the automation of assigning identity to managent instances independently of the physical location of managents.

Another addressed issue is how managent topologies are established. A managent topology defines communication relationships among managents. Sub-ordered managents need to know to which supervisor managent they report. Typically, this is configuration information passed to managents during configuration. This information needs to be described and maintained in the configuration base separately for each managent instance. We avoid the need for configuring topologies separately by using the recursive deployment process for establishing a default communication topology. Links between parents and children are kept in managents throughout the hierarchy. This approach is slightly less flexible since it assumes a deployment hierarchy as the only managent

communication topology, but it avoids the need for additional configuration on the other hand. However, other topologies may be established as well. They then need to be configured separately.

To summarize, the proposed method has advantages over prior art regarding

- the automatic assignment of identity to managent instances independently of physical hosting addresses;
- very dense deployment descriptions for large amounts of “similar” managents;
- regular expressions used to effectively formulate pattern used in repositories to match managent’s identifier paths;
- using only references to managent software as well as configuration data allows rather than this data itself provides the reduction and simplification of data maintained in repositories (XML and XML references can be used);
- maintenance in repositories is simplified and reduced to maintain references rather than particular versions of software and configuration data;
- entirely relying on references in repositories also allows the “webification” (establishing hyper-linked structures) of configuration data and software incorporating offerings from a multitude of vendors (it can also be seen as a new way for distributing and managing software and configuration data based on a web -model);
- the self-establishment of a default communication topology among managents based on the recursive generation of the deployment hierarchy;
- domain structures can easily be established using domain repositories;
- the isolation of configuration data form the management system as well as from the managed system by maintaining them in separated repositories;
- requirements to potential managent hosting systems are reduced to a minimum by only assuming the initial bootstrap protocol handler (boot stub) on those systems;
- the explicit representation of topology information in the directory, and finally
- the complete automation of the deployment process for managents applied in massive amounts.

The principles presented in this paper for managents can be generalized for common services or software systems.

The proposed solution has been prototyped in Java, and we experimented with it in a 524-node Linux cluster, the largest system available to us at that time this work was done.

Table of Contents

Massive Deployment of Management Agents in Virtual Data Centers 1
1 Introduction 1
1.1 Impact of Scale on Systems of Management Agents 3
1.2 Structure of this Report 4
1.3 Comparison With the State of the Art 4
1.4 Assumptions 5
2 Deployment Infrastructure 7
2.1 Topology of Boot Stubs 8
2.2 Repositories 8
2.3 Directories 10
2.4 Initiators 11
3 Deployment Process 12
3.1 The Boot Stub 12
3.2 Control of the External Managent Environment 13
3.3 Control of the Managent 14
4 Protocols 16
4.1 Boot Stub – Initiator – Protocol (BIP) 16
4.2 Boot Stub – Repository – Protocol (BRP) 17
4.3 Boot Stub – Managent – Protocol (BMP) 18
5 Recursive Deployment of Managents 19
5.1 The Recursive Deployment Process 20
5.2 Assigning Identity to Managents During the Deployment Process 20
5.3 Obtaining Managent Deployment Data from Repositories 21
5.4 Controlling the Recursive Deployment Process 22
5.5 Parallel Deployment Processes 23
5.6 Domain Repositories 23
6 Summary 25
Table of Contents 27
References 28

References

[1] Kotov, V.: *On Virtual Data Centers and Their Operating Environments*, HP Labs Technical Report, HPL-2001-44, March 8th, 2001.

[2] *HP Utility Data Center (UDC)*, http://www.hp.com/go/hpudc, or http://www.hp.com/go/always-on, November 2001.

[3] Rolia, J., Singhal, S., Friedrich, R.: *Adaptive Data Centers*, Proceedings of SSGRR 2000 Computer and eBusiness Conference, L'Aquila, Italy, Proceedings on CD-ROM, ISBN 88-85280-52-8, or http://www.ssgrr.it/en/ssgrr2000/proceedings.htm, August 2000.

[4] Kotov, V.: *Towards Service-Centric System Organization*, HP Labs Technical Report, HPL-2001-54, March 21st, 2001.

[5] Graupner, S., Kotov, V., Trinks, H.: *A Framework for Analyzing and Organizing Complex Systems*, Proceedings of the 7th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2001), pp. 155-165, Skövde, Sweden, June 11-13, 2001.

[6] Mansuri-Samani, M.: *Monitoring of Distributed Systems*, Ph.D. Thesis, 165 pages, University of London, UK, December 1995.

[7] Graupner, S., Kotov, V., Trinks, H.: *Distributed System Management with System Factory*, HP Labs Technical Report, HPL-2000-152, November 17, 2000.

[8] Andrzejak, A., Graupner, S., Kotov, V., Trinks, H.: *Self-Organizing Control in Planetary-Scale Computing*, submission to the IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2nd Workshop on Agent-based Cluster and Grid Computing (ACGC), May 21-24, 2002, Berlin.

[9] HP: OpenView Network Node Manager (NNM), http://www.openview.hp.com/products/nnm.

[10] HP: OpenView, http://www.openview.hp.com.

[11] Chemnitz University of Technology, *CLiC 524-node Linux Cluster*, http://www.tu-chemnitz.de/urz/anwendungen/CLIC.

[12] http://cfengine.org.

[13] HP ITSM, http://www.openview.hp.com/products/itsmchange/.

[14] Croft, B., Gilmore, J.: *The Bootstrap Protocol (BOOTP)*, RFC 951, September 1985, with related DHCP extensions, see RFC 15[32-34], http://www.ietf.org/.