UNIVERSITY OF WESTMINSTER

WestminsterResearch
http://www.wmin.ac.uk/westminsterresearch

Agent-based service management in large datacentres and grids.

Sophia Corsava
Vladimir Getov
Harrow School of Computer Science

Copyright © [2003] IEEE. Reprinted from 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (Ccgrid 2003) pp.633-640.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners. Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of WestminsterResearch. (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.

Agent-Based Service Management in Large Datacentres and Grids

Sophia Corsava and Vladimir Getov
Harrow School of Computer Science, University of Westminster, London, U.K.
Email: sophiac6@yahoo.com, V.S.Getov@westminster.ac.uk

Abstract. *Increased computational demands and data mining present the IT world with new challenges. Amongst others, the maturing grid technology aims to address them. To take full advantage of the grid capabilities and enhance its effectiveness in complex and dynamic computational environments, we must make service management more stable, less computationally expensive and more autonomic. In this paper, we propose a synthetic approach to deal with service management in large Unix datacentres that involves the employment of intelligent agents and ontologies. These agents can automatically detect and correct faults at run-time and manage services.*

1. Introduction

There are a number of definitions for agents. Of them, the Wooldridge-Jennings one is the most relevant to our work. According to them, agents are software or hardware-based systems that are autonomous, interact with other agents or humans, are reactive, proactive, can take initiative and exhibit goal-directed behaviour [15]. Agents have been used in service, task and resource mapping, security, event management, adaptation, information organisation and retrieval among other things [18]. Examples are: RETSINA that manages adaptation in open Internet environments; RASCAL that handles service mapping and arbitration services; JAM, a belief-desire-intention agent architecture that supports goal-based reasoning with priorities; JACK Intelligent Agents that provide the infrastructure for developing agents for distributed applications; OpenCybele for event, thread, and concurrency management, as well as internal event generation (send and receive). Agents for query optimisation have been used in grids [12], but to our knowledge, there are no references for agents used in service management or discovery in grids.

Currently, there are a number of methods to discover services in grids. These include [7]: 1) Flat decentralized discovery architectures (like LDAP), 2) Hierarchical Discovery architectures, 3) Relational Approaches with databases, 4) flat unicast and 5) Multicast. However, more than 80% of service requests end up without response [7], and in general there are a number of problems in the areas of service and resource discovery and management [7, 16], such as stability, timely resource discovery and security [2].

In this paper we shall discuss how intelligent agents and ontologies can be used to discover/present services in large Unix-based datacentres and improve service availability. In addition, we shall present a proposed implementation of our work for grids. The paper is organized as follows. Section 2 discusses our building methodology. Section 3 discusses how our work could be used in grid-enabled environments, while section 4 presents some results from one of our actual implementations in a production environment.

2. Building Methodology

2.1 Overview

Our approach involves: 1) Unix shell based intelligent agents that monitor, troubleshoot and manage services within the datacentre. 2) Dedicated administration servers that act as external agent co-ordinators in a high-availability failover configuration to avoid single points of failure. 3) A dedicated private network, where all agent related traffic goes through to avoid agent-related communication errors and congesting the public LAN and 4) Static and dynamic ontologies.

2.2 Ontologies

Ontologies have been widely used in philosophy, logic and lately in artificial intelligence and software engineering. Ontologies as described by John F. Sowa [13], study the categories of things that exist or may exist in a domain and are catalogues of things that are assumed to exist in that domain. Based on these principles, our ontologies describe the concepts (host and services) and their relations (e.g. a host A "owns" a service A, that exists or may exist – i.e. available for use or not). These ontologies include:

Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)
0--7695-1919-9/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

- Static local knowledge templates (SLKT) that contain information about what the server should be like hardware-wise, which applications it should run, as well as all application external and internal dependencies and requirements (file systems, path names, application component startup sequences, binary location, application type, version, name, IP address, port it listens to – if any, application process names and numbers etc). These are created manually whenever a new service is introduced to the datacentre and they reside on each host locally.
- Dynamic local service profiles (DLSP) that are generated dynamically and automatically by each agent controlled host in regular intervals. They contain information about server hardware, software, load, capacity and services at run-time. These ontologies are flat ASCII text files with 3 columns. The first column contains the description, the second the keyword corresponding to the description and the third the description/keyword value. Agents use standard Unix tools (such as *uname –a, ifconfig, uptime, who* etc) to generate these values dynamically.

Based on these ontologies, agents create two main types of shortlists: index static service lists (ISSL) and dynamic global service profile lists (DGSP). ISSLs contain very basic information about each server or service or resource IP address and services. This shortlist is generated using SLKTs. It can contain up to 200 entries and looks like:

IP_1, port number, service_name_1.
IP_2, port number, service_name_2 etc

(i.e. each administration server can take care of up to 200 servers if the private agent LAN is 100BASE/T - much more if it is Gigabit Ethernet. This value has been experimentally proven to be optimum to avoid congestion in the private agent LAN). These lists are static once created and they reside on administration servers. DGSPs contain information about all running and available services across the entire datacentre. Available services are presented as *<Server type, OS, memory and CPUs, Application type and version, Current Load, Users logged in, Geographical Location, Site Name, IP Address>*. DGSPs are generated dynamically by administration servers. They are compiled together when administration servers collect all DLSPs from all servers in their care. DGSPs are sorted by service type, server hardware specification (model, CPUs and memory), load and current users. To this end, an intelliagent local and specific to administration servers only, uses "*awk*" commands to generate them from the DLSP keywords.

The ontological knowledge is automatically processed by intelliagents to determine host and service status. Agents using keywords mentioned in ontologies can "understand" if there are hardware or software problems. Agents come to this conclusion by comparing values in static ontologies (SLKTs) against those in dynamic ones (DLSPs). When it is determined that there is a fault (different values, i.e. services, hardware, software or resources are unavailable), they create a flag in the flags log directory that describes it appropriately and initiate "healing" actions to repair it. Various other intelliagents that monitor applications with multiple dependencies and required resources can later check these flags and take the desired course of action in order to achieve their ultimate goals (i.e. all services, resources and hosts available and in a good state).

2.3 Intelligent Agents

Intelligent agents or intelliagents are unix programs that monitor systems and services and wherever possible automatically correct run-time operational faults with as little downtime as possible. They are highly modular and use constraint-based causal reasoning to decide the best course of action [10]. A causal model is a triple that encodes the truth-values of sentences that deal with causal relationships. They include 1) action sentences such as A will be true if we do B, 2) counterfactuals such as A would have been different if it were not for B and 3) plain causal utterance such as A may cause B OR B occurred because of A.

Intelliagents are installed locally on each server they monitor, always at the same physical location *"/apps/intelliagents"* and are "awakened by local Unix crons every *X* minutes (every 5 minutes for example). Intelliagents do not use a relational database (to avoid corruptions and for simplicity), they use static and dynamic ontologies as discussed. Our intelliagents are mainly developed in bourne shell and are as likely to fail as any standard system startup script (in Unix-based systems most startup/shutdown scripts are written in bourne shell) [11]. They use Unix IPC and exit codes to communicate with the operating system. The reason why we used Unix shell is because it is very easy and fast to write scripts, change them when needed or troubleshoot them. In addition, we did not have to install additional compilers that would put more load to monitored systems, or that would compromise security in any way as is the case with JAVA or C++ based software [3, 4]. Finally, by experience, we know that when a system is failing or is overloaded, complicated measurement/troubleshooting tools tend to stop working altogether [5, 6]. It should be noted however that agents can be written in any other programming language that suits the programmer best.

Intelliagents use 2-phase locking which is a programming discipline that shows that no lock can be released, before the last lock has been obtained. This avoids them operating inconsistently by "healing" the same type of resource concurrently. All actions are logged and every time an intelliagent observes a problem and

takes an action, a message is sent to human operators (usually by email). On each server a full intelliagent suite is running locally that is read-only. On each external administration server intelliagent originals are kept in a secure location. Human administrators are allowed very limited access to administration servers. Access control procedures are in place to ensure that no modifications take place without detection. Password ageing is also implemented that forces users and administrators to change them on a daily basis. The SSH [1] protocol is used, while all other connection methods (such as remsh, rlogin, rsh, telnet) are disabled by default during the server build process. This protects intelliagents, hosts and services from unauthorised accesses, intrusions and changes. This type of security was deemed necessary after we discovered experimentally that human operators were responsible for a significant amount of errors that caused 1,536 hours of cumulative downtime within 16 months at a customer site. This confirms surveys [9] reporting that 40% of downtime is due to human errors.

Whenever local intelliagents run, they produce flags in the dedicated *"/logs/intelliagents/intelliagent_name"* directory on the local server disk to show the status of the run. A number of flags are produced with appropriate naming conventions that show what happened and exactly where the agent found a fault. Absence of these flags means that we either have an internal intelliagent problem or that they did not run at all. Administration servers monitor the creation of these flags every $X+5$ minutes, where X is the frequency intelliagents run, i.e. every 10 minutes (adjustable parameter). If these flags are not there, they start troubleshooting intelliagent processes. Whenever an agent detects an error it tries to fix it. All intelliagents run in parallel, in a distributed manner and do not depend on each other as each agent is started by the local Unix cron. At start-up each intelliagent checks to see if any other of the same type is running, if so it exits, i.e. you can never have two backup intelliagents running at the same time. It also removes its own flags from previous runs and old status profiles. For each application type there are customised error categories. Application health is determined by attempting to connect to them every Y minutes and run basic commands (such as a "get" on a web server process for example). This is essentially the way intelliagents communicate with applications – by trying to use them and by examining the resulting exit codes at the Unix shell.

Each intelliagent has 5 major parts: a) Monitoring, b) Diagnosing, c) Self-Healing/Action/Repair, d) Communication/Logging, e) Self-maintenance. The monitoring part is tasked to look after one particular resource or system aspect. Whenever the monitored subject does not respond as expected, the diagnosing part is invoked and goes through a series of tests to determine the root of the problem. The diagnostic procedure is done in two ways; statically and dynamically. Statically from parsing and examining error logs and flags and dynamically by the use of Unix administration commands to ensure the best possible diagnosis. Findings are recorded automatically in the dynamic ontologies. Based on these findings the self-healing portion gets activated and starts repairing the faults.

The communication part is responsible for communicating with other intelliagents and human operators. It is also responsible for logging all intelliagent activities and results. Intelliagents are classified according to their functions and tasks. Intelliagent categories include: 1) Hardware agents that look after hardware components (CPU, memory, boards etc), 2) Operating system/network agents that look after all OS and network related aspects, 3) Resource intelliagents that are responsible for managing and configuring resources such as disks, network cards, virtual memory etc, 4) Application/Service intelliagents that manage and troubleshoot local and global application/services across the datacentre and 5) Status intelliagents that dynamically generate status profiles for servers, resources and services in terms of availability, load, capacity, geographical location and site name.

2.4 Administration servers

Administration servers are used as external checkpoints for all intelliagents. They are configured in a high-availability (HA) fail-over configuration using a commercial cluster software such as Veritas Cluster Server [14] and they share common disks mounted via NFS. These disks contain all agent related information, such as source code, static ontologies, profiles, logs etc. If any administration server crashes, the high-availability software automatically nominates another administration server in the group to take over activities (default HA behaviour).

Administration servers work in parallel all together. They ensure that local service and status server intelliagents run in a timely manner and that they generate DLSPs and status flags at the end of each run. They achieve that by checking if these exist and if so their creation timestamps. If a DSLP is not present or has an old timestamp, during their monitoring they start troubleshooting. They remotely connect to the service server that is causing the problem, remove any still running service intelliagents, overwrite the intelliagent source code and static ontologies from their local software depots, restart the cron (just in case the cron has stopped and therefore agents did not run as expected), and remotely "kick-start" status intelliagents. If intelliagents still do not run (they establish that by monitoring if a DLSP has been generated or not) they notify human

IEEE COMPUTER SOCIETY

administrators. The service server at that point is considered as being unable to offer any of its services to the agent community. Administration servers use ISSPs to sequentially copy DLSPs over the private network, if service servers are unable to access the administration server NFS disks and place their files directly there.

The collection/copy task is initiated with a 1-minute difference from each other, to avoid congestion in the private agent LAN. It takes about 1-2 minutes for each administration server to collect DLSPs from all 200 hundred servers in the shortlist, as each DLSP is less than 0.5 KB. If they cannot access a server, human operators are notified to look into it. ISSLs are used to confirm if all DLSPs are present and if so, their creation timestamps are checked to verify if they are up-to-date. If not, administration servers try to generate and copy them remotely. Even if they do not manage to collect all DLSPs, they still proceed to the classification step. The ones they have not managed to collect are sent to human operators via email usually, in the form of a report. Operators can then troubleshoot services manually.

2.5 Service management

Service management is handled in a what could be called unorthodox way. Each host in the datacentre is responsible for "knowing" and taking care of its own resources and services locally. This has been proven experimentally, to be the safest and less "resource" expensive way to manage hosts, services and resources. The local to each host status intelliagent is "awakened" by the host Unix cron and compiles dynamically the local DLSP. To confirm that local services are available on each server, status intelliagents invoke local service intelliagents who attempt to connect to running services and perform very simple queries (query types, connectivity tests and timeout baselines are provided by specialised application/service providers). If services that should be running on that host are not running, service intelliagents start troubleshooting. Their goal is to ensure that local services run at all times and if not restart them. Once they achieve this, they perform the prescribed tests again and if there is a problem they cannot resolve they notify human administrators (usually via email or SMS).

FLAG NAME	MEANING
SRV_name. UP	All service components run
DEP.dep_name.SRV_name.FAULTED	One or more service dependencies are not up or cannot be contacted.
REQ.req_name.SRV_name FAULTED	Local service requirements are not present
SRV_name.STOPPED	Local service is stopped
SRV_name.CLIM.MAX	Maximum number of application connections has been reached
dep_name.SRV_name.WAITING	Our software waits for a service dependency to become available
Absence of any flag or OLD flags	Internal error or our software has not run

Table 1. Service intelliagent flag names and their meaning

Service intelliagents, compare local SLKTs and DLSPs to monitor and troubleshoot application/service-related issues (SLKTs contains information about how the service SHOULD be and DLSPs about what the service currently IS). Every time they run, they create flags to indicate the service status (up or not) and any problems they have encountered. In Table 1, we can see in more detail some examples of flag naming conventions and their meaning.

Let us suppose that we need to support a multi-component application, whose components run on physically separate servers. We shall call the global application "EXAMP". EXAMP has three components, a web server called "WEB1", a Sybase database called "SYB_DB1" and an interface process between the two called "INTER1". The IP addresses server/application components are "IP_WEB1", "IP_SYBDB1", and "IP_INTER1" respectively. The web server listens at IP_WEB1 port 80, the database at IP_SYBDB1 port 5000 and the interface process at port IP_INTER1 port 6721. The correct functionality of the database depends on both the web server and the interface process being available in that order. For the sake of this example, we assume that this service intelliagent is called "*exampintelli*" and runs every 15 minutes. The directory where it creates the flags is called "/logs/exampintelli/FLAGS" (the $LOG_DIR). The service ontology is shown in Table 2.

SERVER NAME	HOST1
SERVER IP ADDRESS	IP_SYBDB1
APPLICATION RUNNING	SYB_DB1
LOCAL APPLICATION PART OF	EXAMP
CONFIGURATION DIRECTORY	/apps/Sybase/11.9.2
DATA_DIR_1_NAME	/apps/Sybase
DATA_DIR_1_MNT	tiger:/vol/volroot/sybdir
CONFIGURATION FILE	Syb.conf
PROCESS NAME 1	sybase
PROCESS NAME 2	XXXX
MIN NUMBER OF APPL PROC RUNNING	2
MAX NUMBER OF APPL PROC RUNNING	30
MAX No FOR APPL CONNECTIONS	30
STARTUP SCRIPT	/etc/rc3.d/s99sybase
REQUIRES_1	/apps/sybase
REQUIRES_2	/opt/sybase
DEPENDS_1	WEB1
DEPENDS_2	IP_WEB1 80
DEPENDS_3	INTER1
DEPENDS_4	IP_INTER1 6721
STOP SCRIPT	/etc/rc3.d/k99sybase

Table 2 Service ontology example for EXAMP.

This ontology contains information about the Sybase requirements such as directories that need to be present, mount points (e.g. tiger:/vol/volroot/sybdir) etc. It also contains the other two application components Sybase needs to be up and running before it starts (i.e. dependencies). The sequence by which they are defined in the ontology signifies their precedence in the dependency

0--7695-1919-9/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

checks. If any single one dependency or requirement is not satisfied, the database will not start.

The local service intelliagent that monitors the health of the database component behaves as follows (note that the flag timestamp is determined by the flag creation date, flags are created with the Unix command "`touch flag_name`"): ***Removes*** all its past flags from directory $LOG_DIR at startup. ***Tries*** to connect to SYB_DB1 port 5000 and runs a simple query such as "`select count from table_name`". ***If successful*** it creates the flag "$LOG_DIR/SYB_DB1.UP". ***If not***, checks to see if all application processes are running (all this information is retrieved from the ontology). If none are running, it creates the flag "$LOG_DIR/SYB_DB1.STOPPED". If all Sybase processes are running, it checks to see if the maximum connection limit has been reached. ***If not*** and it still cannot connect to it, it creates the flag "$LOG_DIR/SYB_DB1.FAULTED", stops any running Sybase processes and informs OPS (human operators). ***If more*** Sybase processes are running than they should, it tries to kill the obsolete ones (zombies or application processes that should not be there – any application process not in the ontology is considered illegal). ***It then*** tries to connect to Sybase again. ***If it succeeds*** it creates the flag "$LOG_DIR/SYB_DB1.UP" and informs OPS ***else*** the flag "$LOG_DIR/SYB_DB1.FAULTED", stops any running Sybase processes and informs OPS. ***If some*** Sybase processes are running, it tries to start the missing ones. It tries to connect again and if it succeeds it creates the flag "$LOG_DIR/SYB_DB1.UP, ***else*** the flag "$LOG_DIR/SYB_DB1.FAULTED", stops any running Sybase processes and informs OPS. ***If all*** Sybase processes are running and the maximum connection limit has been reached, it tries to clear any old Sybase connections using Sybase commands provided by an experienced Sybase DBA. ***It then*** tries to connect again. ***If it succeeds*** it creates the flag "$LOG_DIR/SYB_DB1.UP" and informs OPS about what it did. ***If it cannot connect*** it creates the flag "$LOG_DIR/SYB_DB1.CLIM.MAX" and informs OPS. In any case it continues to the next step as it cannot be sure yet why there is a problem if at all. ***It tries to connect*** to the first dependency, in this case IP_WEB1 80 (the web server and does an http get), if it is unable to do this successfully for 3 consecutive times, ***it stops*** the Sybase database if it is not stopped already. It goes into an infinite loop, where it tries to connect to the web server every 5 minutes and emails OPS at every iteration. If/When it finally succeeds, ***it continues*** likewise with the second dependency defined in the ontology (connection and a small query on IP_INTER1 6721). It behaves likewise waiting for the connectivity test to succeed. If not it stays in the infinite loop and emails accordingly OPS. If/when it successfully goes through all dependency checks, it proceeds with the local requirement checking. (Note that it is not necessary for the loop to be infinite, in some cases the loop can be finite, by incorporating a small counter. As Unix shell loops of this type hardly consume any system resources, we were happy to let it run until the rest of the application components became active). ***It then checks*** if the local application requirements are present, i.e. if the "/apps/Sybase" directory is mounted, ***if not*** it tries to mount it with a command such as "/usr/sbin/mount –F nfs tiger:/vol/volroot/sybdir /apps/Sybase". Likewise, it checks sequentially if the rest of the dependencies are present. If not and it is unable to satisfy them it creates the flag "$LOG_DIR/REQ.*requirement_name*.FAULTED". It then ***emails*** OPS and ***exits***.

At that point human operators need to intervene and resolve the problem. They can easily add the resolution procedure to the intelliagent code so that next time it knows what to do to acquire the resource(s). If the requirement check succeeds for all Sybase requirements, it uses the startup script to start the database and checks the exit code. If the exit code is 0, it performs another Sybase connectivity test and if successful, it creates the flag "$LOG_DIR/SYB_DB1.UP". Finally it emails OPS. At this point it has successfully started the database and all dependencies are running as well. If the Sybase startup script returns a non-zero exit code, it stops any Sybase process that may have started, it creates the flag "$LOG_DIR/SYB_DB1.FAULTED" and emails OPS so they can intervene manually and fix the problem. Please note that similar types of checks run on the rest of the servers that are part of the "EXAMP" application. To avoid time related errors all hosts are time synchronized via NTP [11].

3. Agents and Grids

We propose that administration servers are used as site-grid gateways/service brokers and intermediaries amongst different datacentres. In addition, they can be used to co-ordinate service management within the local datacentre by grids. Administration servers have inside knowledge of all agent-controlled devices as described, that they can very well present to the grids via their DGSPs. Administration servers located at different geographical sites use the same service representation types to "talk" with each other at the global level. Local site references can be "aliased" to ontology keywords for intra-site communication and information exchange. Unix-based sorting and search algorithms can be used to select appropriate services based on semantic representations of services (see EXAMP example). To load balance potential incoming/outgoing grid requests, we propose that load balancing network devices are used as the second point of entry of a grid service request originating from another geographical site (first point being the

IEEE COMPUTER SOCIETY

router/firewall, second the load balancers and third the servers themselves).

The load-balancing algorithm ensures that only one gateway deals with a single global grid request, to avoid logical processing errors. The algorithm that can be used for this purpose is an MD5 [8] dynamic hashing algorithm based on both the source and destination IP address (for extra security we can use both IP addresses – if not possible we can use the administration server IP addresses for both). Dynamic hashing aids in improving global utilisation, by mapping each object to a single server. Objects are sorted deterministically between hosts and content synchronisation and duplication are eliminated. This type of hashing has been effectively utilised in NetCache [19], by Netapps, while the MD5 authentication algorithm is widely used by CISCO [20] amongst others, to verify that TCP requests originated by a peer are indeed so and not by other devices spoofing that peer's IP address. These proposed grid gateways can therefore be used as brokers for global grids amongst service requestors and service providers, while they protect the internal structure of the local sites. The grid gateways can in turn, distribute the requests based on the current load of each service server. DGSPs are by default sorted by the least busy server so the first available service server can be used.

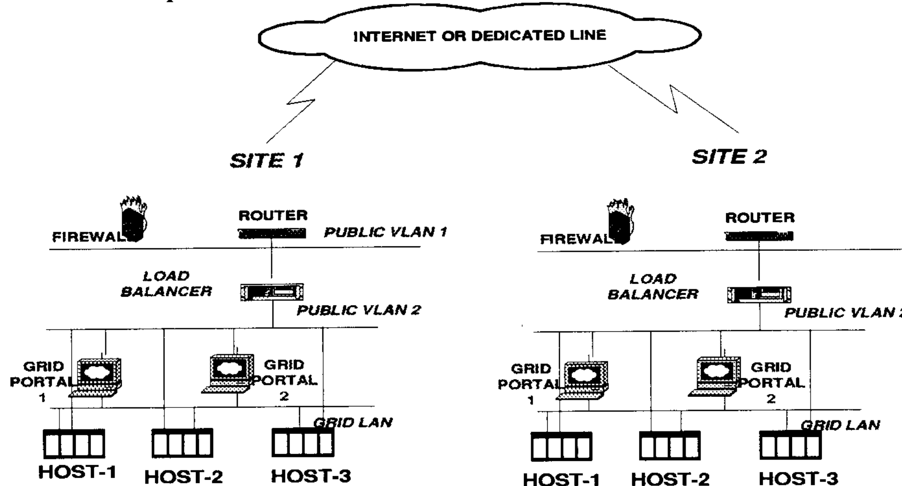

Figure 1. Example of a proposed network infrastructure topology for global grids.

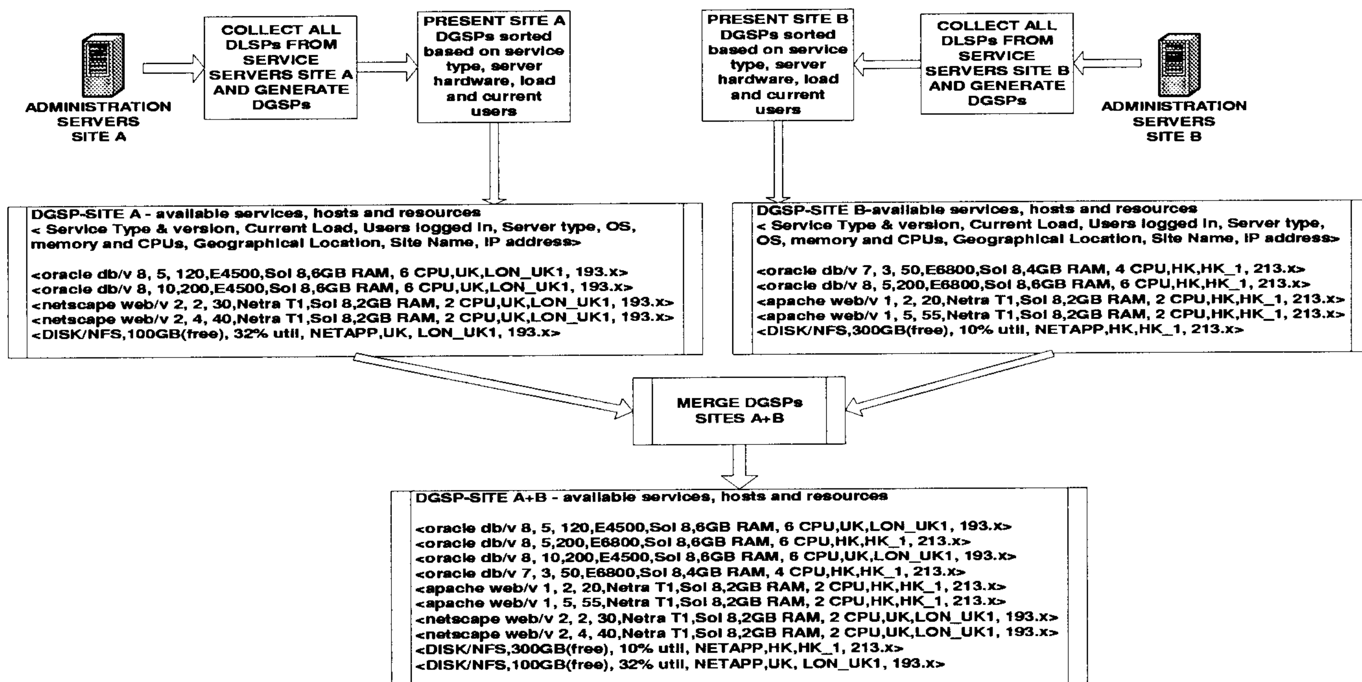

Figure 2. The role of the administration servers in service presentation.

IEEE COMPUTER SOCIETY

Figure 1 shows an example of the proposed network topology for global grids, where site 1 and site 2 exchange all resource/service related information via grid gateways in a high-availability fail-over configuration that are kept in sync. In addition to the public LANs, there is a dedicated grid network that is used for all grid-related communications. The load balancer is using a dynamic MD5 hashing algorithm based on both the origin and destination address. In this way both security and load balancing are achieved. Grid gateways hold all resource and status information for all grid members and can be used as service brokers amongst different sites.

Administration servers from each site collect all DLSPs from all agent-enabled servers and each compile a site-specific DGSP (see Figure 2). Each DGSP contains information sorted using the same keywords in exactly the same way, irrespective of differences in the site-specific DLSPs. Site DGSPs are merged together and both administration servers have a common DGSP that contains information for both sites, sorted by service type, load, users, server type, geographical location and site name. Services that are less loaded are presented first. The constant update of site-specific DGSPs ensures that the common DGSP is always up-to-date.

4. Results

Our agent software and proposed topology have been successfully used between two sites, one in the UK and one in Hong-Kong, in a peer-to-peer configuration. Administration servers were used as gateways to exchange service availability and status information between the two sites. Our agents were used to automatically perform system administration, check and maintain multi-component service availability and health and present available services between the sites whenever a service/application became unavailable on any one of them. They were also used to monitor (with "ping" requests) servers and notify administrators if any of them became unavailable. Each site runs the complete spectrum of customer services. Services were duplicated in this way, but the users wanted to be able to fail-over to either site if services/applications became unavailable for any reason, or when the entire site failed altogether. Services that needed to fail-over, included customer registration, e-mail, web services and customer billing. The two sites were connected with dedicated T1 lines. All databases for the above mentioned services were replicated between them using commercial software called Veritas Volume Replicator (VVR) [17], which kept all databases between the two sites in sync. All databases contained cumulative information for both sites. Site-specific DGSPs were used to present services. Whenever the billing database in HK was unavailable, the UK database was used instead and vice versa. When the HK database was available again, the VVR software was responsible for automatically updating it with the latest information from the UK database (default VVR behavior). The UK site had 300 servers dedicated to these services, while the HK site had about 150. Both had 2 administration servers each in a fail-over configuration, clustered with Veritas Cluster Server 1.3.0 [14]. Servers were mainly SUNs running Solaris and PCs running Linux. Database sizes ranged between 100-250 Gbyte each.

To explain better how agents were used, we shall consider the management of customer's billing services. Customer bills were requested manually by users on a daily basis. A Java graphical user interface presented users with all available databases offering these services and the load/capacity of the machines they were running on. Java programs collected this information from administrator servers from both sites. Users were able to choose which database was less loaded from the list they were presented and use it. Access control was relative to the user name each user was using. Users (starting from the Unix level) had access to specific machines and applications and could only execute a restricted set of commands. There were additional access control lists at the database level. Wherever the operating system had role specification tools (e.g. Solaris 8) we used it, otherwise we used the sudo tool and detailed sudoers lists [11]. Intelliagents compiled DGSPs using service name keywords as previously described. For each site the same keywords were used. Administration servers maintained alias lists that linked site specific service names with DLSP and DGSP keywords. Database naming conventions were service and site specific (such as "*Billing_dbase1.UK*"). We did not have to move any data across the two sites manually, only the jobs. Input and output data were present in each site's databases and the VVR software kept them in sync automatically whenever any transactions were committed. Intelliagents made sure services were available and presented to the Java user interface software in the same way, irrespective of the site they were located at. Users did not know anything about internal site structures this way. Job scheduling was handled by in-house developed programs.

The problems we had were mainly network-related. More specifically, on many occasions we either lost connectivity between the two sites or latency was bad. As a result, databases were not properly synchronized. The second more frequent fault was that databases crashed while processing a job mid-way. Intelliagents were able to restart the databases but could not help with the continuation of the job processing much initially. Oracle recommended we use Oracle Parallel Server [21] to deal with these types of problems. However, implementation costs were prohibitive so the customer did not choose this option. To deal with this problem, we had to develop job

IEEE COMPUTER SOCIETY

scheduling intelliagents that monitored job queues and automatically re-submitted jobs to databases, using always the first database in the list of available resources. This shortcut was successful, but it could not deal with network related problems effectively across the two sites.

Figure 3 demonstrates how intelliagents have improved service availability before and after they were implemented. The graph shows service downtime caused by all failures for 32 months in total, 16 months before any of our work was implemented and 16 months after. The first 16 months prior to our work, cumulative service downtime was 168 hours. This downtime was caused by host, resource and network related problems. During the subsequent 16 months our work was implemented, service downtime caused by the same type of faults went down to 12 hours in total.

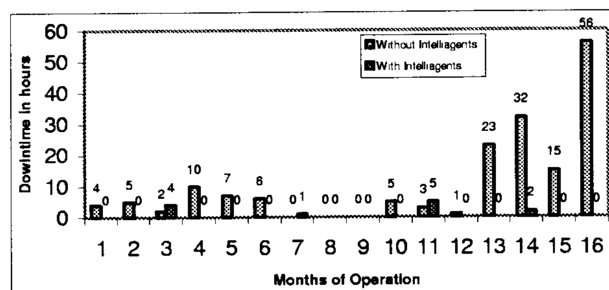

Figure 3. Cumulative service downtime in hours from all reasons per month for 32 months in total; 16 months before intelliagents and 16 months after.

5. Conclusions and Future Work

Unix based intelligent agents and ontologies can and have been used successfully to manage services in production environments. Much work remains to be done, to adapt them to work successfully in fully dynamic environments and improve their troubleshooting capabilities and service presentation policies. We hope that the mechanisms by which they manage services can help develop further grid service management. Our future work, involves integrating a grid toolkit such as Globus [16] with our agents. We also think that intelliagents should be able to allocate services using some form of prediction related to the anticipated service load. Although statically defined maximum load baselines were attempted to be used towards that end, the job scheduling software could not successfully calculate the anticipated load overhead the server would have to take. Faults caused by database overload resulted in databases and servers crashing. Service intelliagents were able to restart services and make them available for use again or present alternative services when they were not. Human administrators were always appraised of the situation, as they received frequent notifications from intelliagents about the status of all activities.

References

1. Barrett, Daniel J., Silverman Richard, "SSH, The Secure Shell: The Definitive Guide", O'Reilly & Associates, February 2001.
2. Chuang Liu, Lingyun Yang, Angulo Dave, Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman,C., "Design and Evaluation of a Resource Selection Framework for Grid Applications", 11th IEEE International Symposium on High-Performance Distributed Computing, Edinburgh, July 2002.
3. Corsava Sophia, Getov Vladimir, "Self-Healing Intelligent Infrastructure for Computational Clusters", ACM Proceedings, SHAMAN Workshop, New York, June 2002.
4. Corsava Sophia, Getov Vladimir, "Intelligent Fault-Tolerant architecture for cluster computing", IASTE, PDCN03, Innsbruck, Austria, Feb 2003.
5. Corsava Sophia, Getov Vladimir, "Intelligent Architecture for Automatic Resource Allocation in Computer Clusters", IPDPS/CAC03 workshop, April 2003, Nice France, IEEE CS press.
6. Corsava Sophia, Getov Vladimir, "Improving Quality of Service for Application Clusters", IPDPS/PDSECA03 workshop, April 2003, Nice France, IEEE CS press.
7. Dinda P, Plale D, "A Unified Relational Approach to grid Information Services", Grid Forum Informational Draft GWD GIS-012-1, GGF5, 2002.
8. Devine. R, "Design and Implementation of DDH: Distributed Dynamic Hashing" (FODO'93), Chicago, Illinois, Springer-Verlag, October 1993.
9. Patterson D., "A new focus for a new century: availability and maintainability >> performance," Keynote speech at USENIX FAST, January 2002.
10. Pearl Judea, "Reasoning with cause and effect", IJCAI Award Lecture, 1999.
11. Quigley, Ellie, "Unix Shells by example", Prentice Hall, 1999.
12. Serafini, Luciano, Stockinger Heinz, Stockinger Kurt, Zini Floriano. "Agent-Based Query Optimisation in a Grid Environment", IASTED International Conference on Applied Informatics (AI2001), Innsbruck, Austria, February 19 - 22, 2001.
13. Sowa, John F., "Knowledge Representation: Logical, Philosophical, and Computational Foundations", Brooks Cole Publishing Co, 2000.
14. Veritas Cluster Server, release 1.3.0, Veritas Software Corporation, 2000.
15. Wooldridge, Michael and Nicholas R. Jennings, "Agent Theories, Architectures, and Languages: a Survey," in Wooldridge and Jennings Eds., Intelligent Agents, Berlin: Springer-Verlag, 1-22, 1995.
16. http://www.globus.org The Globus Project,
17. http://www.veritas.com/us/products/volumereplicator/
18. http://www.agentbuilder.com/AgentTools/commercial.php
19. http://www.netapp.com/products/netcache/
20. http://www.cisco.com
21. http://www.oracle.com

IEEE COMPUTER SOCIETY