KHNUM - A SCALABLE RAPID APPLICATION DEPLOYMENT SYSTEM FOR DYNAMIC HOSTING INFRASTRUCTURES

Alain Azagury[1], German Goldszmidt[1], Yair Koren[2], Benny Rochwerger[1] and Arie Tal[3]
[1]*IBM Research*
AZAGURY@il.ibm.com, gsg@us.ibm.com, ROCHWER@il.ibm.com
[2]*Technion Israel Institute of Technology*
yair_k@cs.technion.ac.il
[3]*IBM Toronto Lab*
arietal@ca.ibm.com

Abstract: In a dynamically scalable hosting infrastructure for e-business computing, servers need to be quickly allocated in order to satisfy a sudden demand for increased computing power for a hosted site.

Khnum is the applications and data management component of Océano - a dynamically scalable hosting infrastructure for e-business computing utilities. It is responsible for server reconfiguration and for application deployment. Application deployment involves all services, configuration directives, executables and data of the application. A hosted site may include several applications.

Khnum enables Océano to rapidly deploy multiple applications to tens of servers simultaneously in just a few minutes. It uses AFS as the infrastructure for secure storage, automatically mapping files and directories onto the new servers' local filesystems and multicasting hot AFS cache content to the new servers. To avoid overloading the AFS servers during the deployment process, the hot cache content is multicasted to all the new servers, avoiding the boot storming (or "rushing") effect. This, in turn, improves the scalability of the deployment process; experimental results attest to Khnum's scalability in simultaneously deploying applications to tens of servers.

1. INTRODUCTION

The Océano project aims at providing a dynamically scalable hosting infrastructure for e-business computing utilities. Current co-location hosting environments use dedicated computing resources for every hosted site; these resources determine the capacity of the hosted site's applications.

In these environments, the process of adding new resources is complex, time-consuming and labor-intensive. This process normally involves the physical installation of the machines and supporting infrastructure (space allocation, racks or shelves, electricity, cooling, routers, etc.), and the deployment of the hosted site's applications, data and middleware. This process may also require some downtime while the site is being reconfigured for the added servers.

In addition, hosted sites increasingly require support for peak workloads that, in some cases, could be an order of magnitude larger than what they experience in their normal steady state. During promotions, a successful marketing campaign or a high seasonal demand, the number of accesses to an e-commerce site can increase significantly compared to its normal steady state, requiring much more resources for handling the larger workloads. In this model, enabling peak-load scale on demand would require large investments in standby, non-shared resources, which would be mostly under-utilized, occupy large amounts of physical space and require regular maintenance. Furthermore, a site may still go down if faced with a larger than expected workload and lacks proper access throttling. Clearly, such a model is not well suited to efficiently mitigate the differences between average and peak workloads. Thus a faster turnaround time in adjusting the resources (bandwidth, servers, and storage) assigned to each hosted site to the actual workload is needed.

Océano modifies the prevalent hosting model by increasing the sharing of resources, and by dynamically adjusting the amount allocated to each hosted customer according to the current observed demand. The Océano model assumes that all servers are clustered together within the same glass-house or campus, and the hosting environment is dynamically divided into secure, single hosted site domains. These domains are dynamic: the resources assigned to them may be augmented when workload increases and reduced when workload dips. This dynamic allocation of resources is controlled by flexible Infrastructure Service Level Agreement (ISLA) contracts with hosted customers. Océano administers the available resources, so that each hosted customer is provisioned as specified by its contract.

To do that, Océano maintains a pool of unassigned (free) servers that can be dynamically assigned to hosted sites. When a higher than normal workload is expected for a hosted site, a group of servers can be pre-allocated for the hosted site in advance. If however, during the live operation of the hosted site, Océano determines that the workload is going to increase beyond the hosted site's current capacity, more servers are then dynamically allocated for the hosted site (in accordance with the ISLA for the hosted site), while the site is handling the current workload. On the other hand, if Océano determines that the workload is going to decrease and that servers are going to be under-utilized, servers are de-allocated from the hosted site and returned to the pool of unallocated servers. Underlying subsystems provide the mechanisms for determining expected workloads, throttling access, managing resources and shifting them to and from a hosted site domain in pseudo real time (minutes) without compromising security requirements.

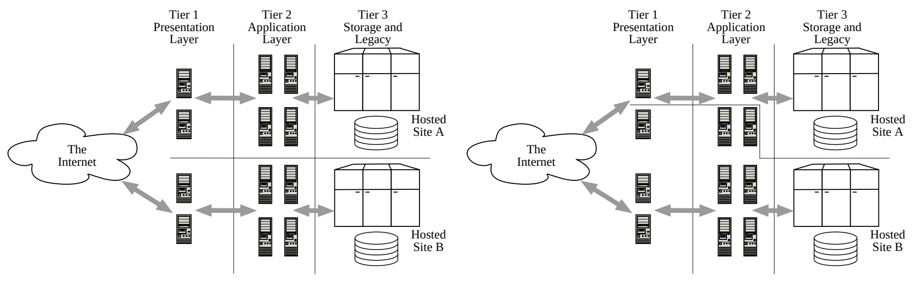
Figure1 illustrates this in a simple multi-tier scenario. Servers at Tier-1 and Tier-2 are dynamically reallocated as workload increases, while Tier-3 resources are assigned to hosted sites for very long periods of time. In Figure 1(a) 6 servers are allocated to hosted site A and 6 to hosted site B. Figure 1(b) illustrates the server allocation status following a detection of a significant increase in the workload of hosted site B and the reallocation of some of the machines. In the absence of "free" machines, to fulfill the needs of hosted site B, the Océano system took under-utlilized resources away from hosted site A[1].

Once a server has been allocated to a hosted site, it should be installed with the applications and data that will enable it to actively participate in the site's workload. In addition, the server's network configuration should be modified to identify it as belonging to the hosted site to which it was allocated. Setting up a new server involves time-consuming and labor-intensive operations such as application installation, configuration and tuning. Many techniques are available to ease these tasks (see Section 2).

In an Océano-hosted environment, the entire hosted site's data (including application binaries[2]) is kept in a shared file system; installation and configuration of applications is done off-line. *Khnum*[3], the applications and data management component of the Océano architecture, is responsible for the rapid deployment of applications (and data) on added servers. Application deployment, the process of setting up all the applications on a new server, is reduced to mapping a remote shared subtree (or several subtrees) to the local file system of the new server.

In the simplest case, the mapping involves only the creation of a few symbolic links, but it could also be relatively complex for applications that require system configuration changes where symbolic links are insufficient. In addition, Khnum is also responsible for configuring the machines to run the applications, and to pre-fetch application data and executables in order to bring them faster into the fully functional state.

Figure 1. Resources assigned are augmented when workload increases and reduced when workload dips



(a) Two hosted sites in the same Océano hosted environment

(b) Due to sudden workload increase the servers are reallocated

In a multi-tier environment, any considerable change in the computing power (that is, number of servers) of any of the tiers can, if not managed properly, severely impact the adjacent tiers. For example, in an Océano-like hosting environment, a sudden increase in incoming requests may cause a quick buildup of computing power at Tier-1 and Tier-2; however, the number of Tier-3 servers stays the same, hence they may become a bottleneck when multiple new Tier-1 servers and Tier-2 servers all request access simultaneously to the same data from the Tier-3 servers. Over-provisioning Tier-3 for the maximum expected workload, as a way to avoid this problem, would be expensive and probably impractical. Khnum avoids choking the Tier-3 servers by pre-fetching "hot" cache data from a running Tier-1 server and a running Tier-2 server into newly allocated Tier-1 servers and Tier-2 servers, respectively. Furthermore, Khnum uses multicast to "push" this hot data to all new servers being allocated to a hosted site simultaneously. The Khnum model proved to be a very efficient mechanism for rapidly deploying applications on new servers. Using this model, we have successfully deployed multiple applications, including Apache, Jakarta-Tomcat, Real Media, and iPlanet.

The Khnum model requires AFS distributed filesystem support and a local filesystem supporting symbolic links. Therefore, although our implementation was based on RedHat Linux v6.2, other UNIX variants could be used for implementing this model.

The rest of this paper is organized as follows. Section 2 describes related work in application installation, distributed file systems, and cache pre-loading schemes. Section 3 presents Khnum's data sharing model, its components, its file system mapping method, and its cache pre-fetching. Section 4 describes experimental results, and conclusions and future directions are presented in Section 5.

2. RELATED WORK

2.1 Installing and Cloning

The RPM utility by Red Hat Linux [2] reduces the installation complexity by packing the applications with installation scripts and a list of dependencies. The **rpm** utility performs the dependencies check, unpacks applications and runs the installation scripts. Disk cloning products, such as *Symantec Ghost* [3], tackle the installation problem by copying entire disks from a pre-configured machine into one or more new machines. All these approaches assume that applications and data are installed on each machine as independent copies, which makes the task of "content management" hard since there is a need to maintain many synchronized copies of the same data. On the other hand, shared file systems have been used to store application binaries in a common place. However, architectures using this approach (for example, [4]) limit the use of shared data to a small set of predefined locations to reduce the complexity of managing mount points or symbolic links, or both.

2.2 Distributed File Systems for clusters

Khnum relies on the Andrew File System (AFS) [5], for sharing the hosted site's content between its multiple servers. Contrary to the weak caching semantics of the popular NFS [6], AFS provides robust caching mechanisms that allow access to large amounts of cached data with speeds comparable to those of local file system accesses. As extensions to AFS's efficient file sharing model, which significantly reduces the workload on the file servers, Coda [7] and Disconnected AFS [8] also allow access to the cached data even when the file server is inaccessible, while keeping their respective file system semantics. In the JetFile [9] multicast-based distributed file system, most operations, which are usually managed by the server, have been moved to the clients. JetFile also supports large caches (in the order of gigabytes), and uses dynamic replication as a means to localize traffic, contrary to AFS's static read-only replication.

2.3 Cache pre-loading schemes

The Khnum model is based on the assumption of a symmetric relationship between servers, that is, we assume that all the servers for a certain hosted site, serve more or less the same content[4], and hence should have very similar cache contents. Therefore, when adding new servers to a hosted site, the cache content of an active server is *multicasted* to the newly added servers, under the assumption that the new servers will be asked to serve roughly the same content. This model is different from SEER [10] and MFS [11], which propose sophisticated hoarding mechanisms for detecting which files should be stored in the cache, as a preparation for disconnected operation, under the assumption that different servers need different content.

Dedicated to caching Web content, LPC [12] employs multicasting for pushing cache content to cache replicas, and automatic tracking of popular Web pages for determining hot cache items. When popular Web pages are determined, their content is pushed to the Web servers, effectively minimizing the time it will take a server to serve that page for the first time. This is rather different from our approach, in which all servers, other than newly added ones, are quite independent of each other. However, in our model, we assume all servers are configured with a large enough AFS cache, so that all popular static Web pages should eventually be stored in the local cache of every server without imposing any additional complexity on the server [13]. On the other hand, if the Web pages are continuously generated every few minutes (for example weather reports, stock reports, sports results), LPC may be more scalable with large numbers of servers, significantly reducing the workload on the shared file system server. TriggerMonitor [14] uses a different, but related, approach by "sensing" changes in database entries, which are used to compose the dynamically generated Web pages. Generated pages are stored and used until a change is detected that requires a page regeneration. The newly generated pages can then be "pushed" to the participating Web servers that serve them as "static" pages.

3. THE DATA SHARING MODEL

To reduce the time and complexity of the *application deployment* process on new servers, all application data (applications executables, configuration files and data[5]) reside on a shared file system and the local disk is used only for temporary data (swapping, caching, and so on), machine specific configuration, and the basic operating system. Essentially, the servers on each cluster become *almost "data less" machines.* When a server either fails or is removed from a hosted site, all the data on its local disk gets wiped out (except for the basic operating system and machine specific configuration)[6].

Ideally, a single symbolic link into a subdirectory in the AFS tree would be sufficient to fully enable applications on the cluster nodes. However, this is doable only for a limited set of applications. In many cases applications require files in system directories such as /etc. Moreover, symbolic links to a shared directory should not be used for directories and files that are, by definition, local to a particular machine. For example, the installation of the Apache Web server creates the /var/log/http subdirectory to keep a local log of http activity. To overcome these problems and still keep the applications on the shared file system, the Khnum model suggests a three-phase process for deploying applications on an Océano-hosted environment:

1 Standard installation - The application is installed locally using the standard application procedure on an off-line machine designated as the *installation staging server*.

2 Analysis and relocation - Once an application has been installed, configured and tested, it is relocated to an area on the shared file system that mirrors the local disk of the installation staging server. Some applications can be directly installed on the shared file system, while others need manual classification of all its files according to their access:

Read-only files which can always be relocated to the shared file system as long as the actual path to them is kept (through symbolic links). In most cases this includes configuration files since these are typically modified once (or sporadically).

Instance read/write files contain data relevant to a particular instance of the application, and hence cannot be shared. Log files are a good example of this type of files. When the application is relocated, these files are separated from the application subtree into a local subtree (by modifying the application configuration files accordingly).

Application-wise read/write files contain information relevant to all instances of an application. To avoid inconsistencies, applications may choose to lock entire files or only portions of them, and to lock only during the write operations or during the entire "life" of the application (in which case the application becomes non-shareable).

Some applications may require an additional effort of creating customized configuration scripts. The purpose of these configuration scripts is to modify configuration files and perform additional tasks that cannot be achieved by simply replacing the existing files with application-specific ones (for example files containing server-specific configuration information). These configuration scripts will be invoked at a post-mapping phase (see below).

3 Mapping - The final step of bringing an application up includes: (a) stopping the processes running on the server undergoing application deployment; (b) creating the necessary links and directories on the server's file system; (c) running configuration scripts (when needed); and (d) restarting the system and application processes.

The application analysis process may be difficult at first, but the knowledge acquired on each application can be re-applied. Although the file hierarchy structure varies from application to application, the ongoing efforts to standardize the file system structure [15] will eventually simplify the process. Note that the time-consuming parts of this process (installation and analysis) are done off-line, and hence they are not part of the rapid deployment of applications on new servers. In addition, when a new server is allocated to a hosted site, previously allocated servers continue to work normally without interruption.

3.1 System Overview

As with most Océano components, Khnum's functionality is divided between a management sub-component - the *Khnum Manager* (**KhnumM**), which oversees and coordinates the application deployment process; and an execution sub-component - the *Khnum Daemon* (**KhnumD**), which is present on all servers and is responsible for file system mapping, cache pre-fetching and configuration. **KhnumM** waits for "application deployment instructions" from either *eClams*, the Océano component responsible for the management of servers [1], or directly from a user when working in standalone mode. These application deployment instructions consist of a hosted site identifier, a list of servers and a "command": either add the servers to the hosted site, or remove the servers from the hosted site, that is, restore the servers to the unallocated state. As a response to these instructions, **KhnumM** initializes the application deployment process on the new servers by selecting from the servers already allocated to the desired hosted site (the "old servers") a *cache pre-fetch source*[7] and then sending a `START-DEPLOYMENT` message to all new servers. We currently pick an arbitrary old server as the cache pre-fetch source. However, it is not clear whether we should pick older servers whose cache might have reached a level of stability, or newer servers that may still hold the files required for initialization.

Figure 2. The life cycle of a server

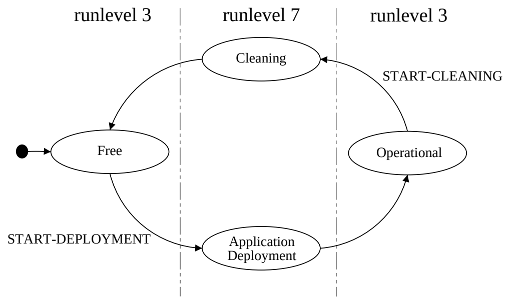


Figure 2 describes the life cycle of a server from Khnum's perspective. **KhnumD** on new servers is in the *Free* state where it waits for the START-DEPLOYMENT message from **KhnumM.** When **KhnumD** receives a request from **KhnumM** it responds by changing the system's *runlevel*[8] to a specially tailored runlevel 7[9]. This transition will cause all the services on the server undergoing applicationed deployment to be stopped and the initialization program *KhnumSetup* to run. *KhnumSetup* will create symbolic links to the shared file system, initialize the cache, optionally run configuration scripts[10], and initiate another runlevel transition, which will cause the new services to be started. A START-DEPLOYMENT message includes information on new AFS cell configuration files, a designated cache pre-fetch server identifier and a setup file (Khnum.setup) containing information on how to setup the required applications.

Upon receiving a START-DEPLOYMENT message, **KhnumD** enters the *Application Deployment* state, and performs the following procedure:

1 Stop all services on the server undergoing deployment (by switching to runlevel 7).
2 Stop AFS daemon[11] services.
3 Copy new AFS configuration files to a standard location.
4 Restart AFS daemon with the new configuration (to gain access to the new hosted site).
5 Find and set up all the symbolic links and directories on the hosted site's AFS cell required by the server (by running KhnumSetup) and run configuration scripts according to the definitions in Khnum.setup.
6 Stop AFS daemon (to prepare for cache pre-fetching).
7 If there are other servers undergoing application deployment for this hosted site, pre-fetch the AFS cache content (from the server designated by **KhnumM** as part of the START-DEPLOYMENT message).
8 Switch back to runlevel 3, which also restarts the AFS daemon and (system and application) services (new services are now started because in step 5 symbolic links were created in /etc/rc.d/init.d and /etc/rc.d/rc3.d).
9 Send a DEPLOYMENT-COMPLETE message back to KhnumM.

AFS cache pre-fetching (step 7) is only needed as a performance optimization. After application deployment, the server automatically enters the *operational state* in which it listens for `START-CLEANING` or `PREPARE-CACHE-SNAPSHOT` (not shown in Figure **2**) messages. The "cleaning procedure" that **KhnumD** performs when it enters the *cleaning* state is very similar to the application deployment procedure:

1 Switch to runlevel 7, which stops all services.

2 Stop AFS daemon.

3 Clean up AFS cache.

4 Remove all the files that were created locally by applications since the applications were deployed on the server.

5 Remove all symbolic links and directories created by **KhnumD** when the server underwent application deployment.

6 Copy administrative AFS configuration files to standard location.

7 Restart AFS daemon.

8 Find and set up all the files and directories on the administrative AFS cell.

9 Switch back to runlevel 3, which restarts the system services (at this point only basic services will be started)

10 Send a `CLEANING-COMPLETE` message back to **KhnumM**.

When servers are in the "free" pool, they can potentially go though hard-drive re-imaging (that is, operating system re-installation) to produce a totally "clean" filesystem. As mentioned before, the time limitations during the cleaning stage are not as tight as during the application deployment stage, and therefore more time-consuming processes such as hard drive re-imaging and machine reboot are possible in the time allowed.

3.2 File System Mapping

The Khnum mapping process automatically creates: (a) symbolic links for read-only and application-wise read-write data, and (b) entire subtrees needed for instance read-write data. This process is driven by a configuration file consisting of *mapping 4-tuples* **(SharedDir, LocalDir, policy, script)** where:

SharedDir: specifies the remote root of the mapping, that is, where in the shared file system the image of additions to the local file system is rooted.

LocalDir: specifies a subdirectory on the local file system where the links/directories found in SharedDir are to be created (recursively).

Policy: specifies what to create on the local file system: subdirectories only (mktree), subdirectories and symbolic links to remote files (mkdir), or symbolic links to remote subdirectories and remote files (mklink).

Script: points to a post-mapping configuration script, that is, after the line in the configuration file is processed, this script will be called.

The different policies together with the post-mapping script allow for maximum flexibility with minimal changes to the local file system. The essence of the process is to create an image of the remote file system structure on the local file system using symbolic links as the preferred mechanism and creating entire subtrees whenever symbolic links are not appropriate. At the end of the process, the minimal number of new subdirectories will have been created, while most of the data will be accessible through symbolic links and the post-mapping script handles the special cases of files that need to be modified instead of replaced.

3.3 AFS Cache Initialization

The AFS aggressive caching mechanism ensures that in the “steady state” accessing frequently used *read-mostly*[12] files is almost as fast as if the files were local to the machine[13] [16]. However, the penalty for reaching this steady state can be high in terms of performance (for example, network traffic), in particular when many machines try to initialize their AFS cache simultaneously. Hence, a method to get many machines simultaneously to a known steady state, without choking the AFS file servers, is needed.

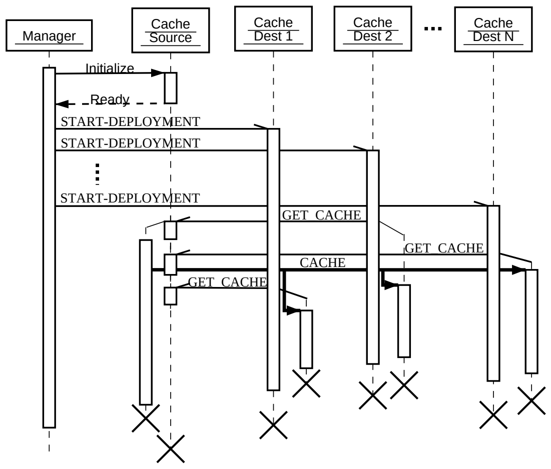
When the AFS daemon is started, it validates all the entries in the local cache. This means that if the cache contents are replaced with a valid content before the daemon is started, this new content will be used as if it was received by the daemon itself. Based on this observation and on our desire to achieve a steady state quickly, the AFS cache on new servers is initialized, before the daemon is started, with the contents of the AFS cache of a server already in the cluster (the *cache source*). Furthermore, to reduce network traffic and application deployment time, this pre-fetching is done by multicasting the cache contents to all the new servers simultaneously, using a multicast-enabled version of the TFTP protocol [17]. Figure 3 shows the message flow of the cache pre-fetch protocol, which is divided into the following three phases:

1 **KhnumM** selects a server as the *Cache Source* and sends it a `INITIALIZE-CACHE` request message to initialize the *cache service* and waits until the cache source is ready; the Cache Source server takes a snapshot of the cache content, starts the multicasting daemon, and sends a message back to **KhnumM** notifying that it is ready for multicasting the cache content.

2 **KhnumM** sends a `START-DEPLOYMENT` request (with the IP address of the Cache Source server) to each joining server. Each server then requests from the Cache Source the cache content. Once all requests are received (or a time-out expires), the Cache Source multicasts its cache contents to the new servers. Multicast TFTP multicasts cyclically all nonreceived blocks until the cache has been transmitted successfully to all the joining servers. Since pre-fetching is done for performance, we also considered multicasting in a *best effort* manner, that is, populating potentially only parts of the caches. **KhnumD** on the new servers performs the application deployment procedure, creating the directory structures and links according to the hosted site’s definition as described in the previous section.

3 The new servers request the cache snapshot from the cache source and use it to initialize the AFS cache and resume the application deployment procedure.

Figure 3. Cache Pre-fetching



4. EXPERIMENTAL RESULTS

As mentioned previously, the prevailing hosting model at the time of writing is that of statically allocated servers. Adding additional servers to the initial setup is both expensive and time-consuming (a matter of days for adding additional machines, provided that they are physically available). Therefore, there is no value in a direct experimental comparison of dynamic allocation times and static allocation times.

On the other hand, we mentioned that adding many servers at once in our model may stress the AFS server (or servers), and we proposed a cache multicast pre-fetch model as a possible solution to this problem. We also conducted experiments that show the difference in "total application deployment time" (the time it takes to add a number of servers to a hosted site) with or without using multicast cache pre-fetching. The results of our experiment show that multicast cache pre-fetching significantly improves the scalability of our solution (in terms of the number of servers that can be added "at once" in single-digit number of minutes).

Our experiments were conducted using a set of 20 servers. Each server was a 600 MHz IBM IntelliStation with 512 MB of RAM running Linux RedHat 6.2. The machines were interconnected with a 100 Mbps fast Ethernet switch. In these experiments the *total application deployment time* was measured, that is, the time from when a `START-DEPLOYMENT` message was sent to the first server, to the time the last server has replied with a `DEPLOYMENT-COMPLETE` message.

Therefore, as described earlier, all the servers, that undergo application deployment, stop their current services and clear their "old" AFS caches before the application deployment for the newly hosted site begins. There are a number of factors that may considerably affect the measured results:

Number of servers: Impacts the total amount of data that, under normal circumstances, needs to be sent over the network. Every server independently accesses the shared file system. Thus, when a file is requested, that is not in the requester's AFS cache, the file's content will need to be retrieved from the shared file system server.

Network bandwidth and load: In many occasions, data from different sources may traverse the network, and affect the experiment's results. In our setup we use a dedicated private network for these experiments, minimizing the possibility that data other than our experiments' data will traverse the network (besides the usual "network noise").

Data set size: The total size of all the content for a given installation. We assume a server is already installed with the base operating system services and utilities, thus an installation refers only to any additional data that may be needed by the server, in order to serve as an equal member of the set of servers it is joining (for example Web servers of a specific Web site).

Cache size: The use of a cache depends heavily on the design and implementation of the shared file system. In our experiment, we use AFS. To *simplify our experiment*, we define our entire data set as "hot" (that is the most needed data, that needs to reside in the cache)[14]. That is, we define the cache size to be equal to or larger than the data set size, which should give us the effect of having our entire data set in the cache.

Shared file system server capacity: The more servers that need to be served from a shared filesystem server the greater the potential workload on the shared server, more so when new servers undergoing application deployment try to access the same content simultaneously. The capacity of the shared filesystem server could easily become a bottleneck when dealing with many servers.

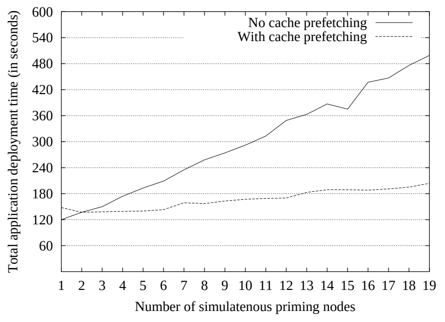
4.1 Description of the experiments

In order to get a clearer picture on the improvement that is provided by multicast cache pre-fetching, we tested our environment using two different data sets:

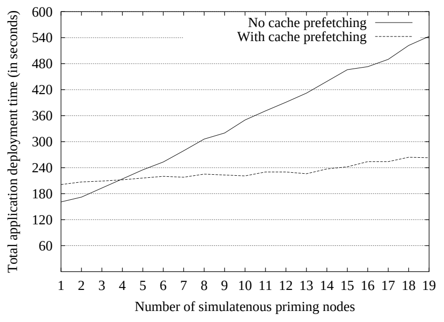
- The "Mayflower" experiment provides an example of a graphic intensive Web site. The site consists of 5421 files of small size (3 KB to 10 KB), adding up to 112 MB. The AFS cache size was set to 120 MB.
- The "ShowBoat" experiment is an example of a streaming video site, with a few relatively long movie clips (3 minutes to 4 minutes long). The site consists of 1855 files adding up to 162 MB. In this case the AFS cache size was set to 180 MB.

The experiments consist of deploying applications and data on up to n servers simultaneously ($0 < n < 20$) and measuring the total application deployment time as defined above. Our primary interest was to measure the effects of the multicast cache pre-fetching; hence tests were done with and without pre-fetching. From the hosted customer's perspective, the interesting measure is the time it takes for a new server to serve its first request (that is, the server becomes fully operational). To approximate this value, we "forced" all new servers to retrieve the entire content from the AFS server such that the AFS cache fills up. Since in both experiments the cache size was set to be larger than the data set size, the data set is exhausted before the AFS cache is

Figure 4. Application deployment experiments



(a) “Mayflower“



(b) “Showboat“

completely full, and thus, the entire data resides in the AFS cache (nearly simulating the effect of having the data set locally installed).

The measured results show that the multicast cache pre-fetching significantly reduces the total application deployment time (see Figures5(a) and5(b)). Both figures show that the application deployment time is a linear function on the number of servers simultaneously undergoing application deployment, where pre-fetching imposes a somewhat costlier setup time (that is preparing the cache content to be sent and sending it to a large number of nodes). However, the difference in slope (roughly a 6:1 ratio) shows that we easily overcame the somewhat higher setup penalty. Note that even though cache data is multicasted, each AFS client still needs to validate the AFS cache content with the AFS server, which causes a slight delay that increases as the number of servers undergoing application deployment increases, hence the small slope. The multicast cache pre-fetching model considerably improves the scalability of the application deployment process. This is especially important for “bursty environments”[15], where the system is expected to react fast to sudden peaks in demands, by adding a large number of servers simultaneously. It is also interesting to note that there is a certain threshold (which varies depending on the data size and number of files) under which it would be much better to avoid using multicast cache pre-fetching in order to obtain the best results. However, since the time difference in these numbers of servers is not large compared to the total time it takes to deploy applications on the servers, a simpler approach may mandate the use of multicast cache pre-fetching on any number of servers. The main reason for this approach being the fact that other than conducting experiments similar to the ones we have conducted, it is quite hard to estimate the threshold for each and every setup. Thus it may be practical to always use multicast cache pre-fetching, even for a small number of servers.

5. CONCLUSIONS

Océano manages a single collection of servers to provide hosting for multiple hosted sites, and dynamically reconfigures these servers to suit the current demand. Khnum manages the application deployment and server reconfiguration of each hosted site. An additional feature is Khnum's cache management, which supports fast concurrent addition of multiple servers for the same hosted site. By using multicast, Khnum achieves significant performance gains relative to earlier methods. We believe this model is flexible enough to host large families of applications that can serve requests independently on separate servers, for example, search engines that access a shared index from the shared file system.

So far, Khnum has focused on management of file system data. In the future, we would like to investigate similar pre-fetching techniques for databases as well. Another interesting direction for future investigation is geographical distribution of server farms, where the Océano system might dynamically allocate servers in the farm closer to the end-users (for example, Web sites transparently migrate from the US to Japan when the level of activity from Japanese users increases).

Trademarks

IBM and IntelliStation, are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Acknowledgments

We would like to thank Liana Fong and Srirama Krishnakumar for their helpful comments and ideas.

Notes

1. The methods by which to shut off a server from incoming requests until it becomes inactive, and the ways by which to determine whether a server is active or not are beyond the scope of this paper.

2. The discussion on sharing binaries ignores any licensing constraints.

3. All Océano component names allude to the ocean. According to Egyptian mythology, *Khnum is* the lord of the cool waters.

4. Our notion is that for many Web sites that serve static (or pseudo-static) data, there is a portion of the data which is hot (i.e. accessed by most users). With a good enough workload balancing component, most of the participating servers will need to access that hot data.

5. In the rest of the paper, we use interchangeably "applications" and "data", since both are stored as files in the shared file system, and treated in the same manner by Khnum.

6. Contrary to the requirement of rapid application deployment due to increased workload on a working site, removing a server from a hosted site does not require the same speeds. Therefore, re-imaging the server's disks is one option for a thorough "cleaning" of any residues or for installing a different version of the operating system, or a different operating system.

7. A *cache pre-fetch source* is an active server in the hosted site, which is instructed by Khnum to multicast the contents of its AFS cache to new servers being added to the hosted site. For more information on cache pre-fetching in Khnum, see section 3.3.

8. A UNIX System V term for the set of services and kernel state (single/multi user).

9. Every Khnum controlled server is configured to include the additional runlevel 7, which is essentially similar to runlevel 6 (reboot) with few modifications for running the setup scripts and automatically switching back to runlevel 3 (multiuser).

10. Some applications may require an additional configuration, which is achieved by customized configuration scripts. This is needed in those cases where there is a need for server-specific configuration.

11. The afsd daemon is stopped last as services run off AFS and afsd refuses to shutdown when there are remote files open.

12. We are assuming, for simplicity, that *read-mostly*− files may be updated by Tier-2 servers (or services) such as a daily forecast update, etc. Updating these files by a Tier-1 server can be handled by AFS, however, it may cause unpredictable results when applied to applications that are ill equipped to deal with sharing of files by multiple instances.

13. We assume that read-write data is stored on shared databases, which are beyond the scope of this paper.

14. In practice, data sets for hosted sites are much larger than the cache size.

15. Examples of such “bursty environments” are an on-line stock reports site during a stock rally, or a news site during a large scale event.

References

[1] K.Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. *Océano - SLA Based Management of a Computing Utility*. In Proc. of the 7th IFIP/IEEE International Symposium on Integrated Network Management, May 2001.

[2] *The Official Red Hat Linux Reference Guide*, chapter 6: Package Management with RPM. `http://www.redhat.com/support/manuals/RHL-6.2-Manual/ref-guide/ch-rpm.html`.

[3] *Symantec Ghost - Product Information*. `http://www.symantec.com/ghost`.

[4] Z. Wensong. *Linux virtual server for scalable network services*. In Linux Symposium, Otawa, Canada, July 2000. `http://www.LinuxVirtualServer.org/ols/lvs.ps.gz`.

[5] R. Campbell. *Managing AFS: The Andrew File System*. Prentice Hall PTR, 1998.

[6] Sun Microsystems. *NFS:Network File System Version 3 Protocol Specification*, February 1994.

[7] P. J. Braam. *The coda distributed file system*. Linux Journal, June 1998.

[8] L.B. Huston and P. Honeyman. *Disconnected Operation for AFS*. In Proceedings of the USENIX Mobile and Location-Independent Computing Symposium, August 1993.

[9] B. Gronvall, A. Westerlund, and S. Pink. *The design of a multicast-based distributed file system*. In Proceedings of the Third Symposium on Operating Sytsems Design and Implementation, Feb 1999.

[10] G. Kuenning. *The Design of the SEER Predictive Caching System*. In Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 1994.

[11] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser. *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System.* In Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995.

[12] J. Touch. *The LSAM proxy cache: a multicast distributed virtual cache.* In Proceedings of the Third International WWW Caching Workshop, Manchester, England, June 1998.

[13] T. T. Kwan, R. E. McGrath, and D. A. Reed. *NCSA’s World Wide Web Server: Design and Performance*. pages 28(11):68-74, November 1995.

[14] J. Challenger, A. Iyengar, and P. Dantzig. *A Scalable System for Consistently Caching Dynamic Web Data.* In Proceedings of IEEE INFOCOM ’99, March 1999.

[15] D. Quinlan. *Filesystem Hierarchy Standard (FHS) - Version 2.1*. April 2000. `http://www.pathname.com/fhs`.

[16] M. Spasojevic and M. Satyanarayanan. *A Usage Profile and Evaluation of a Wide-Area Distributed File System*. In Proceedings of the USENIX Winter 1994 Technical Conference, 17–21, San Francisco, CA, USA, 1994.

[17] A. Emberson. *TFTP Multicast Option*. RFC 2090, Lanworks Technologies Inc., February 1997.