TR 88.389 (H0048) January 2, 2001 Computer Science

IBM Research Report

Khnum - Data Management for a Dynamically Scalable Computing Utility

A. Azagury, G. Goldszdmidt, Y. Koren, B. Rochwerger, A. Tal
IBM Research Division
Haifa Research Laboratory
Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Khnum - Data Management for a Dynamically Scalable Computing Utility

Alain Azagury German Goldszmidt * Yair Koren
Benny Rochwerger
Arie Tal

IBM Research in Haifa
MATAM - Advanced Technology Center
Haifa, 31905 Israel
{azagury,rochwerg,yairk,ariet}@il.ibm.com

Abstract

Khnum is the applications and data management component of Océano, a dynamically scalable hosting infrastructure for e-business computing utilities. Océ ano dynamically adjusts the resources allocated to each domain to its actual demands, enabling multi- enterprise hosting on a collection of shared resources. The customer's data (including application binaries) is kept in a shared file system (AFS), such that installation and configuration of applications is done off-line. Application priming is done by mapping a remote shared subtree (or several) to the local file systems. Khnum supports a multicast cache prefetching model used to initialize the caches and prime the servers. We present experimental results which indicate that Khnum can significantly improve the scalability of such computing server farms.

1 Introduction

The Océano project [1] under development at IBM Research, is aimed at providing a dynamically scalable hosting infrastructure for e-business computing utilities. Hosted customers increasingly require support for peak workloads that are orders of magnitude larger than what they experience in their normal steady state. Thus, a faster turnaround time in adjusting the

*IBM Watson Research Center - gsg@us.ibm.com

resources (bandwidth, servers, and storage) assigned to each customer site to the actual workload is needed. The prevalent co-location web hosting model uses dedicated non-shared infrastructure and servers for each customer. In this model, enabling peak-load scale on demand would require large investments in standby, non-shared resources, which would be mostly under utilized and would occupy large amounts of physical space. Clearly, such a model is not well suited to efficiently mitigate the differences between average and peak load.

Océano modifies this model by pushing towards increased levels of sharing of standard components, and introducing high levels of automation to dynamically adjust web sites to actual traffic demands. It enables multi-enterprise hosting on a collection of shared resources where at any point in time, each resource is assigned for use by only a single customer. That is, the hosting environment is dynamically divided into smaller, secure domains, each supporting one customer. These domains are dynamic: the resources assigned to them may be augmented when workload increases and reduced when workload dips (see Figure 1)

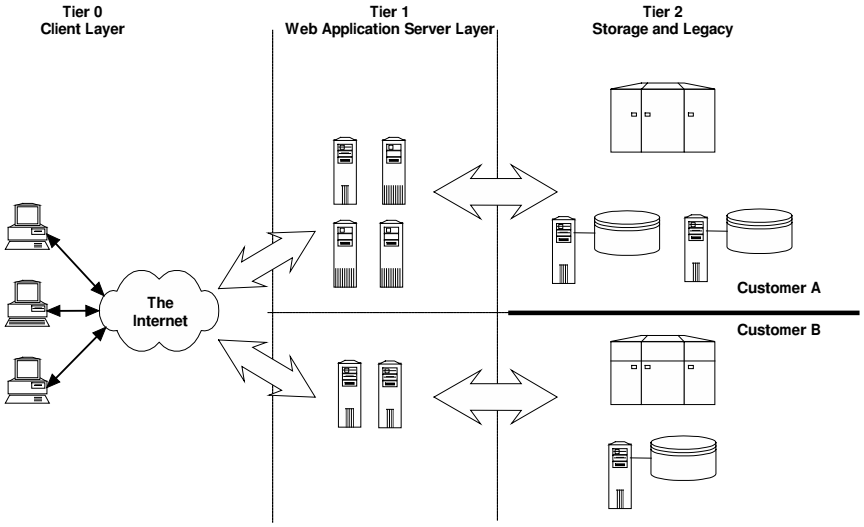
Once a node has been allocated to a customer it should assume the "new personality" as soon as possible. Bringing up a new server involves time consuming and labor intensive operations such as application installation, configuration and tuning. Many techniques are available to ease these tasks:

- RedHat's RPM [2] technology significantly reduces the installation complexity by packing the applications with installation scripts and a list of dependencies. The `rpm` utility perform 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 copies of the same data synchronize. On the other hand, shared file system (see Section 1.1.1 have been used to store application binaries in a common place. However architectures using this approach [4, 5, 6] limit the use of shared data to a small set of predefined locations to reduce the complexity of managing mount points and/or symbolic links.

(a) Two customers in the same Océano farm:



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

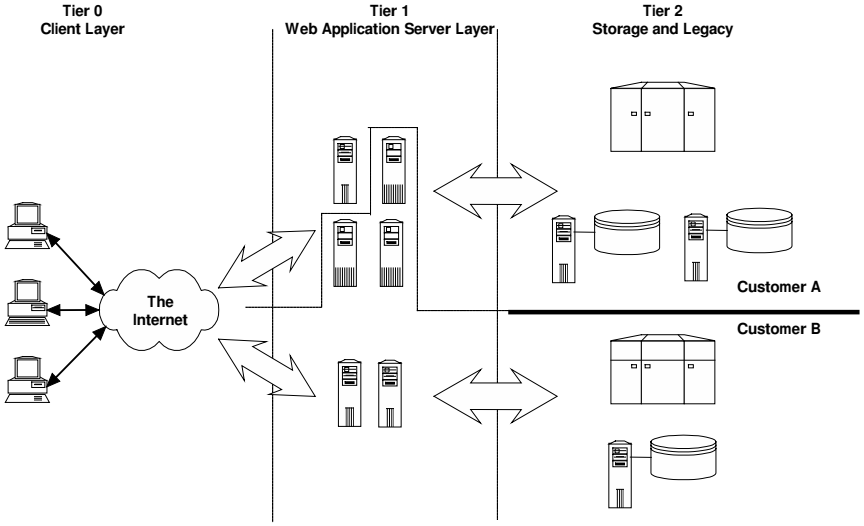


Figure 1: At any point in time machines at the Web Application Servers tier are allocated to a single user but as workload increases/decreases the allocation is changed automatically. In (a) 4 servers are allocated to customer A and 2 to customer B. A sudden increase in the workload of customer B cause the reallocation of some of the machines (b). This example shows an extreme situation: all servers are allocated and to fulfill the needs of customer B, the system took resources away from customer A; in the more typical case, "free" machines will be available for re-allocation.

In an Océano farm all the customer's data (including application binaries [1]) is kept in a shared file system; installation and configuration of applications is done off-line, and the process of bringing up all the applications on a new node (*application priming*) is reduced to mapping a remote shared subtree (or several) to the local file system of the new node. *Khnum*[2], the applications and data management component of the Océano architecture, is responsible for the automatic generation of this mapping which in the simplest case 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 aren't enough.

A sudden increase in workload may cause a site to quickly grow in computing power, as new nodes are allocated to the web application tier, however the number of back-end servers stays the same for longer periods of time, and hence they may become a bottleneck. This is in particular serious given the unique data access patterns expected in an Océano farm, where at priming time, it is likely that many new nodes will request access to the same data at the same time from the back-end servers. Clearly, over-provisioning for the maximum expected workload, as a way to avoid this problem, would be expensive and wasteful and in most cases not practical. Khnum avoids choking the back-end servers by pre-fetching "hot" cache data on newly allocated nodes. In addition Khnum uses multicast technology to "push" this hot data to all new nodes being allocated to a customer simultaneously.

1.1 Related work

1.1.1 Distributed File Systems for clusters

Khnum relies on a shared file system, namely the Andrew File System (AFS) [7, 8], for sharing the customer's content between multiple Web server nodes. AFS provides robust caching mechanisms that allow access to large amounts of cached data with speeds comparable with those of local file system accesses, contrary to the weak caching semantics of NFS [9]. As extentions to AFS's efficient file sharing model, which significantly reduces the workload on the file servers, Coda [10, 11] and Disconnected AFS [12] also allow access to the cached data even when the file server is inaccessible, while keeping their respective filesystem semantics.

[1] On this document the discussion on sharing binaries is purely technical, i.e., even if it is technically possible to share a particular application it may still be prohibited by the application licensing limitations.

[2] The Egyptian lord of the cool waters (*Océano*)

A rather different approach to managing a distributed file system is suggested by Grönvall et al. [13] in their JetFile multicast-based distributed file system. In JetFile, most of the operations, which are usually managed by the server in other file system, have been moved to the clients. JetFile also support 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.

1.1.2 Cache pre-loading schemes

Our model is based on the assumption of a symmteric relationship between nodes, that is, we assume that all the nodes for a certain customer segment, serve more or less the same content, and hence should have very similar cache contents. Therefore, when adding new nodes to a customer segment, the cache content of an active node is *multicast* to the added nodes, under the assumption that the new nodes will be asked to serve the same content. This model is different from SEER [14] and MFS [15], 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 nodes need different content.

Dedicated to caching Web content, LPC [16] employs multicasting in pushing cache content to cache replicas, and automatic tracking of popular web pages in determining hot cache items. When popular web pages are determined, their content is pushed to the Web server nodes, effectively minimizing the time it will take a node to serve that page for the first time. This is rather different from our approach, in which all nodes other than newly added ones, are quite independent of each other. However, in our model, if the nodes are configured with a large enough AFS cache, all popular static web pages should eventually be stored in the local cache of every node without imposing any additional complexity on the server [17]. On the other hand, if the web pages are continuously generated every few minutes (e.g. weather reports, stock reports, sports games results, etc.), LPC may be more scalable with large number of nodes, significantly reducing the workload on the filesystem server. A different, but related, approach was used by TriggerMonitor in the 1998 Olympic Games Web Site [18, 19]. TriggerMonitor was used for tracking updates in the databases of olympic events' results, and generate events for creating the updated Web pages and pushing them to the Web servers.

2 The Khnum Data Sharing Model

To reduce the time and complexity of the *application priming* process [3] on new nodes, all application data (applications executables, configuration files and data) sits on a shared file system and the local disk is used only for the basic OS and utilities and for temporary storage (swapping, caching, logging, etc). Essentially the nodes on each cluster become *almost dataless machines*, i.e., the local disk is used only for temporary data, machine specific configuration and the basic OS.

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/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 log of the local http activity. To overcome these problems and still keep the applications on the shared file system the *Khnum Application Enablement Model* suggests a three-phased process for enabling applications on an Océano server farm:

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

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 node. 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 which contain information relevant to a particular instance of the application and hence do not need to be shared. Log files are a good example of this type of files. When the application is relocated, these files are separated from

[3] In the context of Océano, application priming is the process of installing, configuring and starting applications

the application subtree into a local subtree (by modifying the application configuration file accordingly).

Application-wise read/write files which contain information relevant to all instances of the 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).

3. Mapping - The final step of bringing an application up in Océano is similar to the process of adding a new node to an existing customer: applications/services are stopped; the necessary links and directories are created on the node's file system; and applications/services are restarted.

The application analysis process may be difficult at first, but the knowledge acquired on each application can be re-applied and, although the file hierarchy structure varies from application to application the ongoing efforts to standardize the file system structure [20] will simplify the process. Note that the time-consuming parts of this process (installation and analysis) can be done off-line and are not part of the priming process.

2.1 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 priming process; and an execution sub-component - the *Khnum Daemon* (`KhnumD`), which is present on all nodes and is responsible for file system mapping (section 2.2) and cache pre-fetching (section 2.3).

The Khnum Manager waits for "priming instructions" from either *eClams*, the Océano component responsible for the management of servers [1]; or directly from an user when working in stand-alone mode. These priming instructions consist of a customer identifier, a list of nodes and a "commad": either add the nodes to the customer, or remove the nodes from the customer, i.e., restore the customer to the unallocated state. As a response to these instructions the Khnum Manager initializes the priming process on the new nodes by selecting from the nodes already allocated to the desired customer (the "old nodes") a *cache pre-fetch source* and then sending a `START-PRIMING` to all new nodes.

The Khnum Daemon on new nodes is in the *free state* where it waits for the `START-PRIMING` message from `KhnumM` (see Figure 2). When this message arrives, `KhnumD` enters *priming state* starts the "personality change" as follows:

1. Stop all services.
2. Stop AFS daemon [4]. services
3. Copy new AFS configuration files to standard location.
4. Restart AFS daemon (by switching to `runlevel 7`).
5. Find and setup all the symbolic links and directories on the customer's AFS cell required by the node (see Section 2.2).
6. Stop AFS daemon.
7. Pre-fetch AFS cache content (see Section 2.3).
8. Restart AFS daemon (by switching back to `runlevel 3`).
9. Restart services (new services are now started because in step 5 symbolic links were created in texttt/etc/rc.d/init.d).
10. Send a `PRIMING-COMPLETED` message back to `KhnumM`.

After priming, the node automatically enters the *operational state* in which it listens for `START-CLEANING` or `PREPARE-CACHE-SNAPSHOT` (not shown in Figure 2) messages. The "cleaning process" that `KhnumD` goes when it enters the *cleaning state* is very similar to the priming process:

1. Stop all services.
2. Stop AFS daemon.
3. Clean up AFS cache.
4. Remove all the files created by applications since the node was primed.
5. Remove all symbolic links and directories created by `KhnumD` when the node was primed.

[4] The `afsd` is stopped last since in the application enablement model describe in this document, services run off AFS and `afsd` refuses to shutdown when there are remote files open.

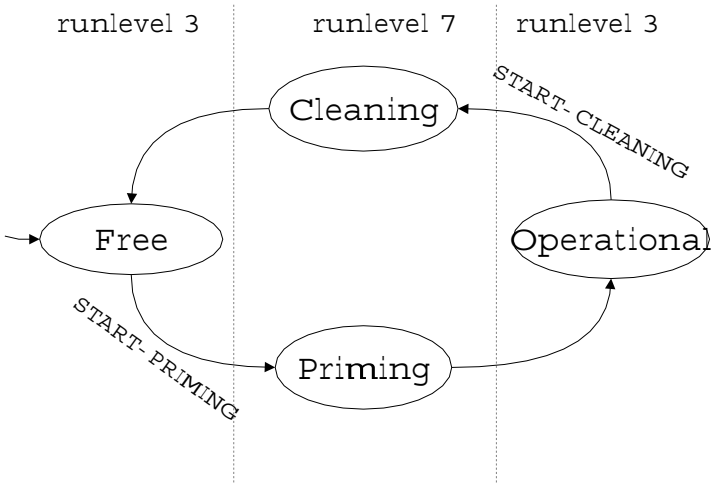


Figure 2: The life cycle of a dolphin from the Khnum: when `KhnumD` receives a request from `KhnumM` it responds by changing the system's `runlevel` to the special level 7. This transition will cause all the services to be stopped, then while in `runlevel 7`, `KhnumD` will create/remove the symbolic links to the shared file system, initialize the cache, and perform another `runlevel` transition which will cause the new services to be started

6. Copy administrative AFS configuration files to standard location.
7. Restart AFS daemon.
8. Find and setup all the files and directories on the administrative AFS cell (see Section 2.2).
9. Stop AFS daemon.
10. Restart AFS daemon (by switching back to `runlevel 3`).
11. Restart services (at this point only basic services will be started)
12. Send a `CLEANING-COMPLETED` message back to `KhnumM`.

Both transitions will be encapsulated by a a System V `init` startup script; when `KhunmD` receives either a priming or cleaning command from `KhnumM` it causes the system to change the runlevel.

2.2 File System Mapping

The Khnum mapping process automatically creates: 1) symbolic links for read-only and application-wise read-write data , and 2) 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, i.e., where in the shared file system the image of additions to the local file system is rooted.

`LocalDir` specifies where a subdirectory on the local file sytem 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`), or 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, i.e, 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 special cases of files that need to be modified instead of replaced are handled by the post-mapping script. The mapping algorithm is described in Appendix A.

2.3 AFS Cache Initialization

The aggressive caching mechanism implemented in AFS ensures that in "steady state", accessing frequently used files is almost as fast as if the files where local to the machine. However the penalty for reaching this state can be quite high in terms of client performance and network traffic in particular when many machines try to initialize their AFS cache simultaneously. Hence, a method to bring many machines simultaneously to a known steady state is needed.

For persistence, 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 created by the daemon itself. Based on this observation and on our desire to achieve a steady state fast, the AFS cache on new nodes is initialized, before the daemon is started, with the contents of the AFS cache of a node already in the cluster (the *cache source*). Furthermore, to reduce network traffic and priming time, this prefetching is done by multicasting the cache contents to all the new nodes simultaneously. The cache pre-fetch protocol is divided into several phases:

1. The Khnum Manager selects a node as the *cache source* and sends it a message to initialize the *cache service*: the cache source node takes a snapshot of the cache content starts the multicasting daemon, and sends a message back to the Khnum Manager notifying that it is ready for multicasting the cache content.

2. The Khnum Manager sends a `START-PRIMING` request to the new node with IP the address of the cache source as a parameter. The new nodes will go through the application priming sequence, creating the directory structures and links according to the customer's segment as described in the previous section.

3. The new nodes request the cache snapshot from the cache source and use it to initialize the AFS cache and resume the priming sequence.

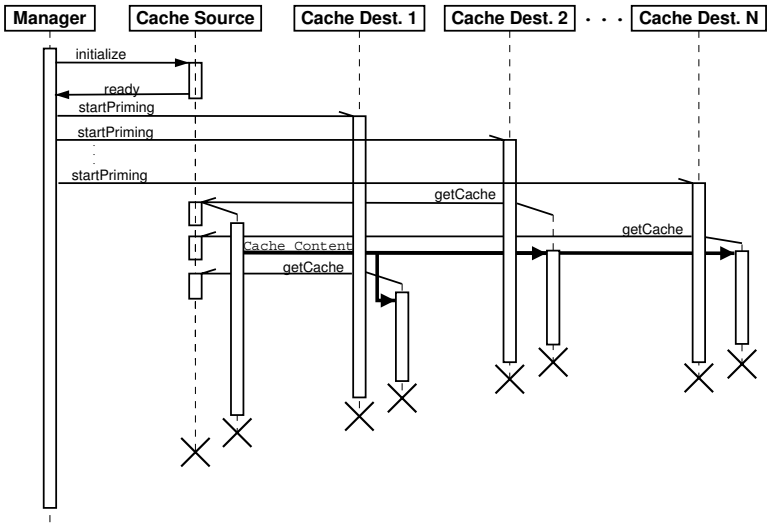


Figure 3: Cache Pre-fetching: The Khnum Manager selects a *Cache Source* and sends it a `initialize-cache` request and waits until the cache source is ready. Then it sends to all joining machines a `startPriming` request. The new machines request from the cache source the cache content and in response the cache source multicasts a snapshot of the cache.

3 Experimental Results

To validate the ideas presented here, we conducted experiments a 20 node cluster. Each node was a 600 MHz IntelliStation with 512 megabytes of RAM running RedHat 6.2. The machines were interconnected with a 100 Mbps fast ethernet switch. In these experiments the *total priming time* was measured, i.e., the total of time from when a `START-PRIMING` message was sent to the first node, to the time the last node has replied with a `PRIMING-COMPLETED` message. There are many factors that may affect the measured results considerably:

Number of nodes The number of nodes impacts the total amount of data that, under normal circumstances, needs to be sent over the network.

Every node independently accesses the shared file system. Thus, when a file is requested, which is not in the requester's shared file system cache, the file's content will need to be retrieved from the shared file system server.

Network bandwidth The amount of data bits that can traverse the network in a given time unit (e.g. 10^8 bits per second, which is, incidentally, the network bandwidth of our experiment's setup).

Network load In many occasions, data from different sources may traverse the network, and affect the experiment's results. Since network bandwidth is limited, any additional data traversing the network may narrow the actual bandwidth. 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 node 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 node, in order to serve as an equal member of the set of nodes it is joining (e.g. 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 the Andrew File System (AFS). To simplify our experiment, we define our entire data set as "hot" (i.e. the most needed data, that needs to reside in the cache). That is, we define the cache size to be equal to or larger than the data set size, which should give us with the effect of having our entire data set in the cache.

Shared file system server capacity The shared file system server is supposed to serve a number of nodes. Naturally, the more nodes that need to be served, the greater the load on the shared file system server, more so when priming nodes, that would be trying to access the same content simultaneously. The capacity of the shared could easily become a bottle neck when dealing with many nodes.

3.1 Description of the experiments

In order to get a clearer picture on the improvement that is provided by our cache multicast prefetching model, we tested on two different data sets:

- The "Mayflower" experiment provides an example of a graphic intensive web site, that consists of lots of small size (3k to 10k) images. The site consists of 5421 files adding up to 112 megabytes. The AFS cache size was set to 120 megabytes.

- The "ShowBoat" experiment provides an example of a streaming video site, that consists a few relatively long movie clips (3 minutes to 4 minutes long). The site consists of 1855 files adding up to 162 megabytes. In this case the AFS cache size was set to 180 megabytes.

The experiments consist of priming up to n nodes simultaneously ($0 < n < 20$) and measuring the total priming time as defined above. Our primary interest was to measure the effects of the multicast cache pre-fetching, hence tests were tests were done without pre-fetching and with pre-fetching. To simulate a "busy web application server" after priming all new nodes were "forced" 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 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 priming time (see Figures 4 and 5). Both figures clearly show that with multicast pre-fetching, the total priming time behaves as $3.5 \times x + K_1$, where x is the number of simultaneous priming nodes, and K_1 is the "setup penalty" which depends on the size of the data set. Without pre-fetching the total priming time behaves as $21 \times x + K_2$, i.e., we've achieved a $6:1$ ratio, and easily overcame the somewhat higher setup penalty. Note that even though cache data is multicasted, each AFS clients still need to validate the AFS cache content validity with the AFS server, which causes a slight delay that increases as the number of priming nodes increases, hence the small slope.

4 Conclusions

The Khnum Application Enablement Model proved to be a very efficient mechanism for bringing up new nodes fast. Both open source applications,

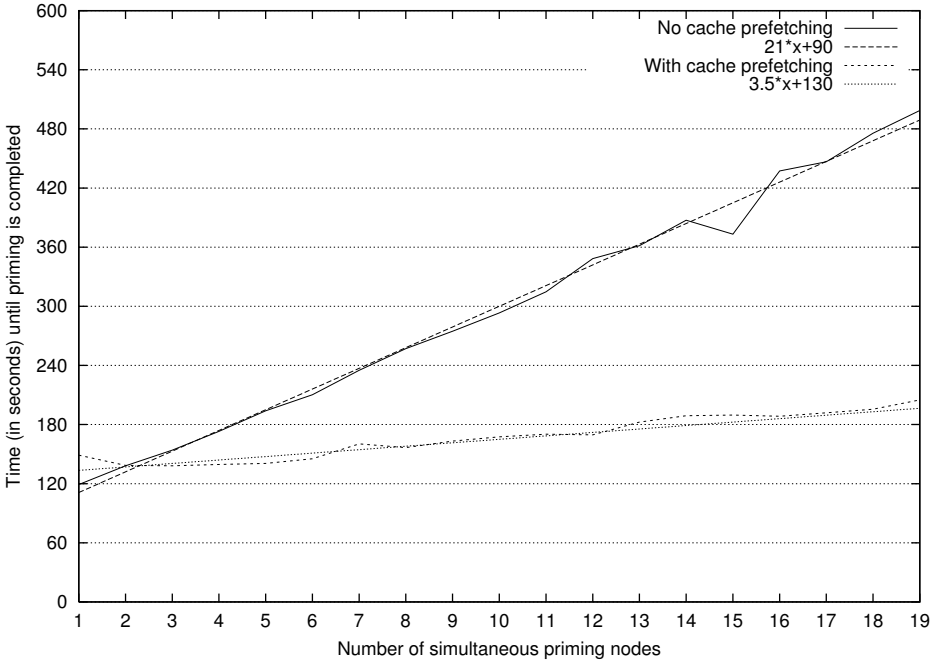


Figure 4: The "Mayflower" application priming experiment.

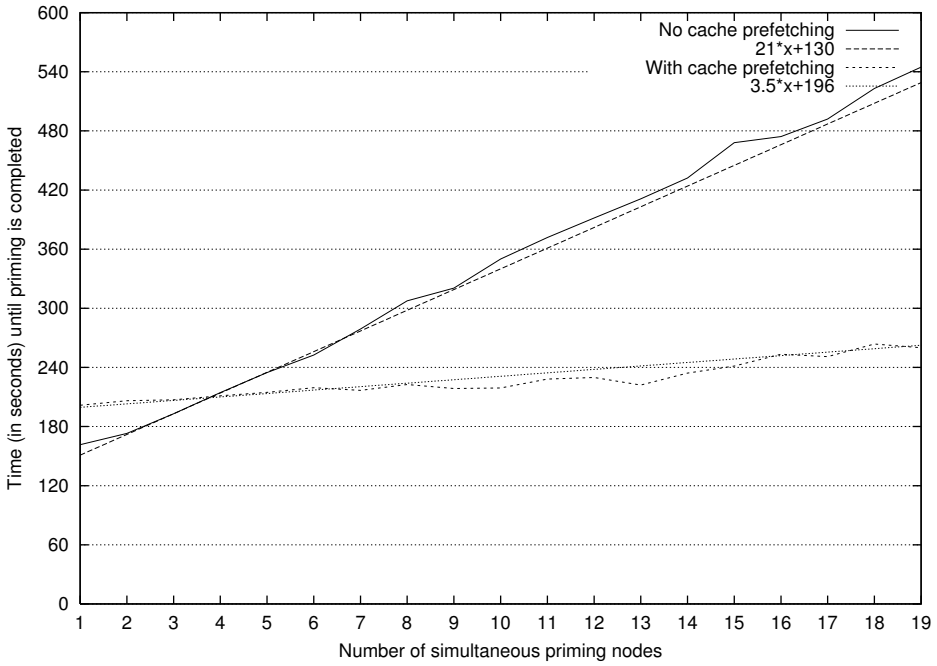


Figure 5: The "ShowBoat" application priming experiment.

such as the Apache Web Server [21] and the Jakarta-Tomcat Web Application Server [22]; and commercial applications such as the Real Media Server [23] and the iPlanet Web Server [24], have been successfully deployed using our model. In addition the multicast cache prefetching model was shown to significantly improve the scalability of the server farm by enabling the priming of large numbers of nodes in about the same time that it would take to prime a small number of nodes.

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 prefetching in order to obtain the best results. However, since the time difference in these numbers of nodes is not so big compared to the total time it take to prime the nodes, a simpler approach may be to mandate the use of multicast cache prefetching on any number of nodes. 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 tell where the threshold for each and every setup lies, thus it would be much simpler to use multicast cache prefetching at all times (even for a small number of nodes).

Acknowlegments

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

References

[1] K. Appleby et al. Océano - SLA Based Management of a Computing Utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[2] *The Official Red Hat Linux Reference Guide*, chapter 6: Package Management with RPM. Red Hat Inc., 2000. `http://www.redhat.com/support/manuals/RHL-6.2-Manual/ref-guide/ch-rpm.h%tml`.

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

[4] G. Goldszmidt and G. Hunt. Scaling internet services by dynamic allocation of connections. In *Proceedings of the 6th IFIP/IEEE Inter-*

national Symposium on Integrated Network Management, Boston, MA, 1999.

[5] G. Goldszmidt and A. Stanford-Clark. Load distribution for scalable web servers: Summer olympics 1996 - a case study. In *Proceedings of the 8th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Sydney, Australia, October 1997.

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

[7] H. J. Kazar et al. Scale and performance in distributed file system. *ACM Trans. on Computer Systems*, 6, February 1988.

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

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

[10] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

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

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

[13] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system, 1999.

[14] 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.

[15] Terry D. B. et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

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

[17] Kwan Thomas T., McGrath Robert E., and Reed Daniel A. NCSA's World Wide Web Server: Design and Performance. pages 28(11):68–74, November 1995.

[18] Challenger Jim, Dantzig Paul, and Iyengar Arun. A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites. In *Proceedings of ACM/IEEE SC98*, November 1998.

[19] Challenger Jim, Iyengar Arun, and Dantzig Paul. A Scalable System for Consistently Caching Dynamic Web Data. In *Proceedings of IEEE INFOCOM '99*, March 1999.

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

[21] B. Laurie and P. Laurie. *Apache: The Definitive Guide*. O'Reilly and Associates, 2nd edition, February 1999.

[22] The Tomcat Documentation. `http://jakarta.apache.org/tomcat/jakarta-tomcat/src/doc/index.html`.

[23] RealSystem Server Basic. `http://www.realnetworks.com/products/basicserver/info.html`.

[24] iPlanet Web Server, Enterprise Edition 4.1 Datasheet. `http://www.iplanet.com/products/infrastructure/web_servers/iws/index.ht%ml`.

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

A The Mapping Algorithm

1. `SubTreesToMake` $\leftarrow \emptyset$
2. for each 4-tuple `(SharedDir,LocalDir,policy,script)` in the configuration file where `policy` $\neq$ `mklink`:
   (a) if `LocalDir` doesn't exists create it.

(b) if `policy` = `mktree` then

- `SubTreesToMake` ← `SubTreesToMake` $\bigcup$ {`LocalDir`}

3. for each 4-tuple `(SharedDir,LocalDir,policy,script)` in the configuration file where `SharedDir` ≠ `any`:

   (a) if `LocalDir` ∈ `SubTreesToMake` then `policy` ← `mktree`

   (b) `sdir` ← `SharedDir`

   (c) `ldir` ← `LocalDir`

   (d) for each directory entry `e` in the directory `sdir`:

      i. if `ldir/e` doesn't exists then:

         A. if `sdir/e` is a directory and `policy` ≠ `mklink` then "recurse":

            - create subdirectory `ldir/e`
            - ldir ← `ldir/e`
            - sdir ← `sdir/e`
            - go back to step 3d

         B. else if `sdir/e` is a file and `policy` ≠ `mktree` then

            - create link `ldir/e` → `sdir/e`

      ii. else (`ldir/e` exists):

         A. if `ldir/e` is a directory and `sdir/e` is also a directory then "recurse":

            - `ldir` ← `ldir/e`
            - sdir ← `sdir/e`
            - go back to step 3d

         B. else if `ldir/e` is a symbolic link to a remote directory and `sdir/e` is a directory then:

            - rename `ldir/e` (say to `ldir/e.backup`)
            - create directory `ldir/e`
            - for each directory entry `x` in the directory pointed to by `ldir/e.backup`
              – create symbolic link `ldir/e/x` → `ldir/e.backup/x`
            - "recurse":

- `ldir` ← `ldir/e`
- `sdir` ← `sdir/e`
- go back to step 3d

C. else (`ldir/e` is neither a directory nor a link to a remote directory):

- rename `ldir/e`
- if `sdir/e` is a directory and `policy` ≠ `mklink` then "recurse":
  - create subdirectory `ldir/e`
  - `ldir` ← `ldir/e`
  - `sdir` ← `sdir/e`
  - go back to step 3d
- else if `sdir/e` is a file and `policy` ≠ `mktree` then
  - create link `ldir/e` → `sdir/e`

4. for each 4-tuple `(SharedDir,LocalDir,policy,script)` in the configuration file:

   (a) execute `script`