INFORMS Journal on Computing
Vol. 18, No. 2, Spring 2006, pp. 267–282
ISSN 1091-9856 | EISSN 1526-5528 | 06 | 1802 | 0267

DOI 10.1287/ijoc.1050.0142

A Data-Mining-Based Prefetching Approach to Caching for Network Storage Systems

Xiao Fang
Department of Information, Operations and Technology Management, University of Toledo, Toledo, Ohio 43606, USA, xiao.fang@utoledo.edu

Olivia R. Liu Sheng
School of Accounting and Information Systems, University of Utah, Salt Lake City, Utah 84112, USA, actos@business.utah.edu

Wei Gao
Department of Information and Communication Systems, Schools of Business Administration, Fordham University, New York, New York 10023, USA, gao_wei@hotmail.com

Balakrishna R. Iyer
IBM Silicon Valley Lab, San Jose, California 95141, USA, balaiyer@us.ibm.com

The need for network storage has been increasing rapidly owing to the widespread use of the Internet in organizations and the shortage of local storage space due to the increasing size of applications and databases. Proliferation of network storage systems entails a significant increase in the number of storage objects (e.g., files) stored, the number of concurrent clients, and the size and number of storage objects transferred between the systems and their clients. Performance (e.g., client-perceived latency) of these systems becomes a major concern. Previous research has explored techniques for scaling up the number of storage servers involved to enhance the performance of network storage systems. However, adding servers to improve system performance is an expensive solution. Moreover, for a WAN-based network storage system, the bottleneck for its performance improvement typically is not caused by the load of storage servers, but by the network traffic between clients and storage servers. This paper introduces an Internet-based network storage system named NetShark and proposes a caching-based performance-enhancement solution for such a system. The proposed performance-enhancement solution is validated using a simulation.

Key words: caching; data mining; simulation
History: Accepted by Amit Basu, Area Editor for Knowledge and Data Management; received October 2002; revised June 2003, September 2003; accepted April 2004.

1. Introduction

The need for network storage has been increasing rapidly owing to the widespread use of the Internet in organizations and the shortage of local storage space due to the increasing size of applications and databases (Gibson and Meter 2000). Proliferation of network storage systems entails a significant increase in the number of storage objects (e.g., files) stored, the number of concurrent clients, and the size and number of storage objects transferred between the systems and their clients. Performance (e.g., client-perceived latency) of these systems becomes a major concern.

Previous research (Lee and Thekkath 1996, Thekkath et al. 1997) has explored techniques for scaling up the number of storage servers involved to enhance the performance of network storage systems. These techniques worked well for LAN-based network storage systems as increasing storage servers decreased the load for each server. However, adding servers to improve system performance is an expensive solution. Moreover, for a WAN-based network storage system, the bottleneck for its performance improvement typically is not caused by the load of storage servers, but by the network traffic between clients and storage servers. A cost-effective way to improve its performance is to distribute storage servers and cache copies of storage objects at storage servers near the clients who request them (i.e., migrate copies of storage objects from storage servers that store them to storage servers near the clients who request them) (Gwertzman and Seltzer 1995, Barish and Obraczka 2000). This paper introduces an Internet-based network storage system named NetShark and proposes a caching-based performance-enhancement solution for such a system. Three major contributions made by this paper are the following.

We have built a caching-based network storage system to expand storage capacities and to enhance access performance of system users.

We have proposed and shown that a data-mining-based prefetching approach outperformed other popularly applied caching approaches. Applications of the proposed caching approach include network storage caching as well as general-purposed web caching.

We have developed a network-storage-caching simulator based on general and proven characteristics of trace logs (e.g., FTP logs) to test the performance of different caching approaches. Applications of this simulator are not limited to network storage caching. The simulator can be easily adapted to test different web-caching algorithms.

The rest of the paper is organized as follows. Features, implementation, architecture, and performance measurements of NetShark are briefly described in §2. We review related research in §3 and propose a data-mining-based prefetching approach in §4. Section 5 presents the simulator used to evaluate the performance of the proposed caching approach. We summarize and discuss the simulation results in §6, and conclude the paper with a summary and suggested future directions in §7.

2. Overview of NetShark

In this section, we briefly introduce features, implementation, architecture, and performance measurements of NetShark. Shark is a second-generation IBM Enterprise Storage Server with a maximum storage capacity of 11 TB. NetShark is an Internet-based network storage system built on Sharks and provides storage services to users at both IBM and the University of Arizona. By providing university users with extra storage space through the Internet, NetShark strengthens the shared IBM and university goals of providing better support for online learning communities. Besides routine functionalities, such as file transfer, file management and user management, NetShark also provides file-compression functionality, which allows users to compress files before transferring them, and file-encryption functionality, which enables users to encrypt critical files. Figure 1 shows a screen shot of NetShark installed at the IBM Almaden Research Center.

As shown in Figure 2, implementation of NetShark consists of three layers: presentation, application, and storage. NetShark was designed to be accessed easily anywhere in the world. Therefore, we chose web browsers (e.g., Netscape) as its presentation tool. The Apache web server was selected as the web server of NetShark for handling HTTP requests from clients and delivering HTTP responses to clients. The application layer consists of Java servlets managed by IBM WebSphere and a database maintained by IBM DB2, namely, Log-DB. The application layer is responsible for user authentication, transaction processing and session management. In addition to web logs collected by the Apache web server, all transactions between clients and NetShark are recorded

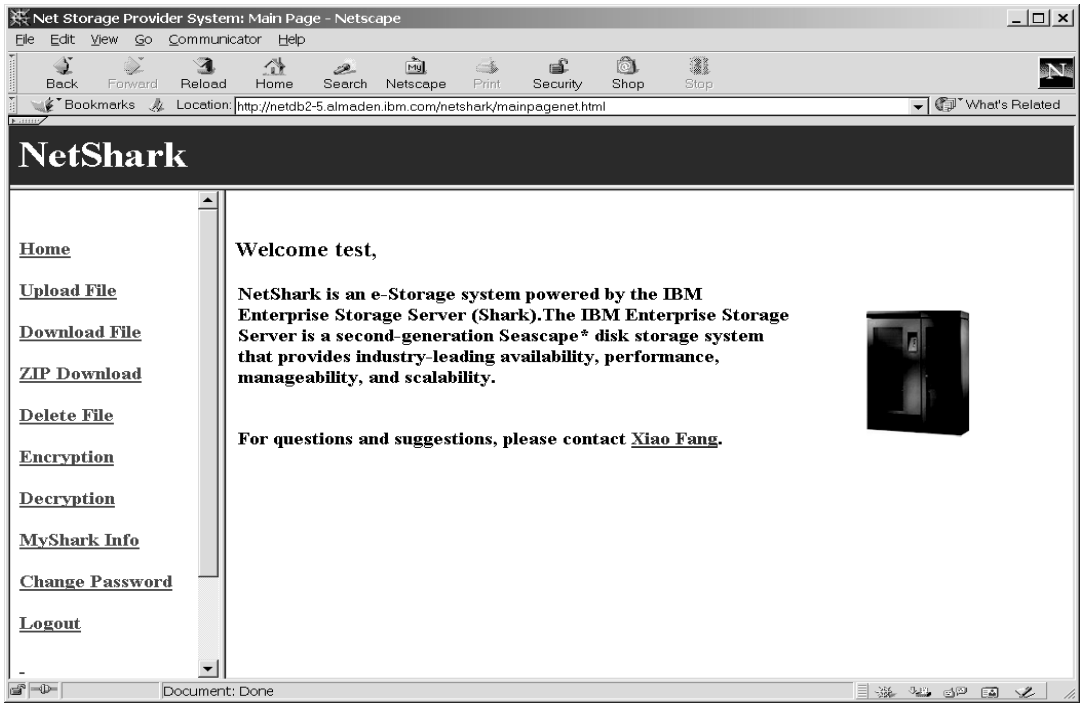

Figure 1 A Screen Shot of NetShark

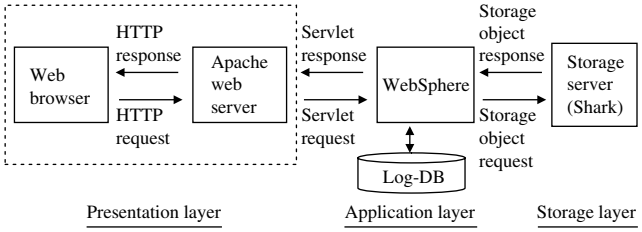

Figure 2 The Three-Layer Implementation Structure of NetShark

in Log-DB. Some critical information missing in web logs (Cooley et al. 1999), such as session identification, are stored in Log-DB. Log-DB provides an ideal source for the data-mining-based prefetching approach to be described in §4. The storage layer is located in Sharks and communicates with the application layer through the SCSI protocol.

NetShark employs a geographically distributed network storage architecture in which Sharks are distributed in several geographically separated regions where their clients reside. We call a Shark placed in a client's region the *Home Shark* of the client. To reduce client-perceived latency and to balance workloads among Sharks, clients' requests are always routed to and responded from their Home Sharks. If a storage object requested is not stored in a client's Home Shark, a copy of the object will be cached to the Home Shark from the Shark that contains the object (hereafter called the *Storage Shark*).

Figure 3 offers an example of the NetShark architecture. In this example, NetShark hosts a *knowledge-management-system* (KMS), which consists of modular knowledge objects in various file and storage formats. Clients of the KMS are distributed in three geographically separated regions, New York, San Jose, and Tucson. Sharks distributed in New York, San Jose, and Tucson are denoted as Shark NYC, Shark SJC, and Shark TUS respectively. A logical scenario of storage distribution could allocate the operating-system knowledge objects to Shark NYC, the database-knowledge objects to Shark SJC and the storage-server knowledge objects to Shark TUS. In addition to serving as the Storage Sharks of the KMS, these three Sharks are also the designated Home Sharks for the clients in their located regions. When clients in Tucson request remote objects stored at Shark SJC, the requested objects will be cached from Shark SJC to the clients' Home Shark, Shark TUS, and remain in Shark TUS until other cached objects replace them. In this example, clients connect to their Home Sharks through the Intranet/Internet and Sharks are connected using Public Switched Data Network (PSDN), a popular and high-performance method to connect servers in different regions.

To compare different caching approaches for NetShark, we introduced three performance measurements for NetShark: client-perceived latency, hit rate, and Shark-Shark response time. Client-perceived

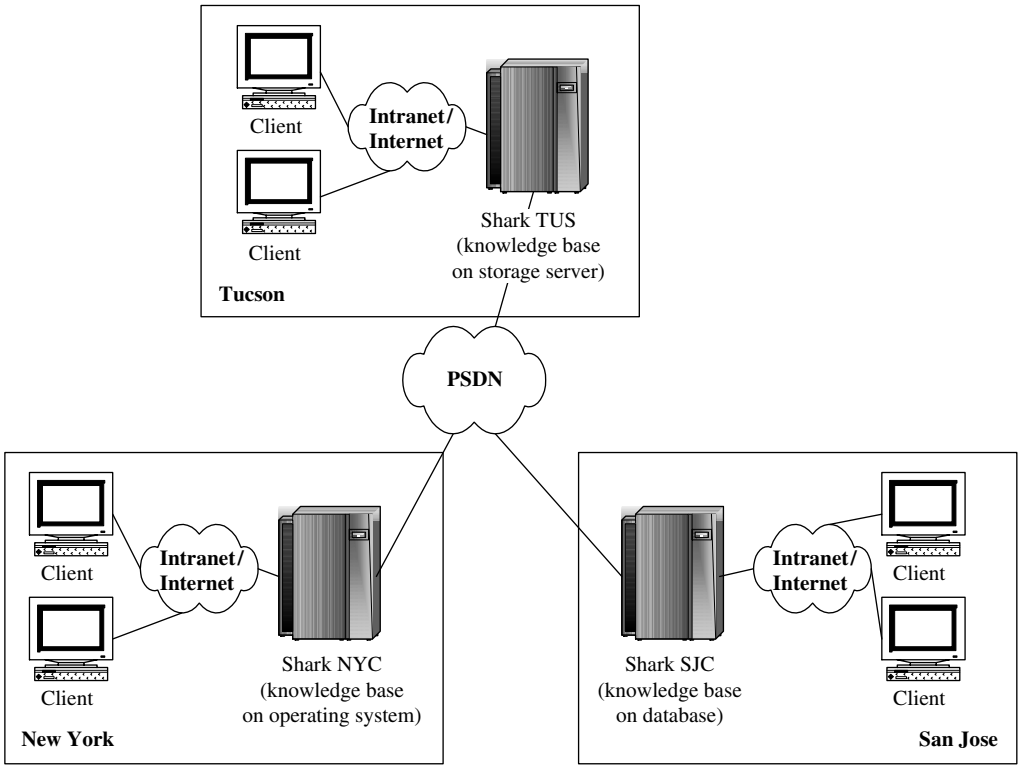

Figure 3 The Geographically Distributed Network Storage Architecture of NetShark

latency and hit rate are popular performance measurements for web caching.

Measurement 1. Client-perceived latency refers to the delay between the time when a client submits a storage-object request and the time when the storage object is received by the client. Retrieving storage objects directly from clients' Home Sharks saves the time of transferring these storage objects from their Storage Sharks to clients' Home Sharks. Therefore, increasing the number of storage object requests that can be directly fulfilled from Home Sharks reduces client-perceived latency.

Measurement 2. Hit rate, $hr = n_{\text{home}}/n_{\text{req}}$, is used to measure the capability that Home Sharks can directly respond to storage object requests. Here n_{home} is the number of storage-object requests that can be directly fulfilled from Home Sharks, and n_{req} is the total number of storage object requests.

Measurement 3. To measure the impact of network traffic between Sharks, Shark-Shark response time is the length of the interval between the time when a Shark submits a storage—object request to a remote Shark and the time when the storage object is received by the requesting Shark.

3. Related Work

In this section, we review research on web caching, on which our caching solution is based. There has been research on file migration for hierarchical storage management and distributed systems since 1970s (e.g., Liu Sheng 1992). Much past research has been incorporated into web caching. Here, we present a review of web-caching research related to this paper. Web caching is the temporary storage of web objects for later retrieval (Barish and Obraczka 2000). Major issues in web caching include deciding what web objects need to be cached, how to replace old web objects when there is not enough space for newly cached web objects, and how to keep consistency among copies of web objects. According to the aforementioned issues, web-caching research can be divided into three parts: caching, replacement, and consistency.

A simple caching approach, called caching on demand, caches only currently requested web objects. More efficient caching approaches, called prefetching approaches, cache both currently requested web objects and web objects predicted to be requested in the near future (Kroeger et al. 1997). Popularity-based prefetching approaches (Dias et al. 1996, Markatos and Chronaki 1998, Kim et al. 2000) predicted and prefetched future requested web objects based on their past request frequencies. Another type of prefetching approaches discovered and utilized access relationships between web objects in making prefetching decisions (Padmanabhan and Mogul 1996, Horng et al. 1998, Fan et al. 1999, Lou and Lu 2002). For example, Padmanabhan and Mogul (1996) modeled access relationships between web objects using a dependency graph in which nodes represented web objects and arcs between nodes represented access relationships between web objects (i.e., how likely one web object will be requested after another web object). Lou and Lu (2002) extended the work in Padmanabhan and Mogul (1996) and considered not only access relationships between web objects, but also access relationships between websites when making prefetching decisions. Chinen and Yamaguchi (1997) proposed a different approach to predicting and prefeching future requested web objects based on structural relationships between web objects. The proposed approach parsed HTML files and prefetched embedded images and web objects pointed to by embedded links. However, this approach greatly increased network traffic between servers and proxies, and it is not practical to prefetch web objects solely using this approach. The prefetching approach proposed in Davison (2002) predicted the next requested web page by analyzing the content of the web pages requested recently. Padmanabhan and Mogul (1996) and Crovella and Barford (1998) showed that prefetching approaches increased network traffic between servers and proxies, which could affect client-perceived latency negatively. However, none of the past prefetching approaches addressed the network traffic increase problem. In Padmanabhan and Mogul (1996), Horng et al. (1998), and Fan et al. (1999), access relationships between web objects were extracted based on an arbitrarily chosen look-ahead window, and these approaches neglected the session—a natural unit to extract access relationships between web objects. Aware of the limitations of past prefetching approaches, the proposed prefetching approach will:

predict and prefetch future requested objects based on sequential object request patterns within sessions and between sessions, and

divide future requested objects into two categories, urgent and wait, and cache objects in the urgent category immediately while caching objects in the wait category only when the network traffic is below a user-defined threshold.

Least-recently-used (LRU) and *least-frequently-used* (LFU) are two widely used cache—replacement approaches. LRU is concerned with recency of object requests, while LFU is concerned with frequency of object requests. LRU, which was previously used for page replacement in memory management (Tanenbaum and Woodhull 1997), is based on the observation that web objects that have been heavily requested recently will probably be heavily requested in the near future. Conversely, web objects that have

not been used for a relatively long time will probably remain unused for a long time. Hence, the web object that has not been used for the longest time is replaced by LRU. LFU replaces the web object with the lowest request frequency. Cache-replacement approaches proposed in Cao and Irani (1997) and Jin and Bestavros (2000) considered the cost of transferring a copy of a web object from a server to a proxy, namely the cache cost. Obviously, it is desirable to replace the web object with the lowest cache cost, if everything else is equal. In Chang and Chen (2002), the replacement decision is based on a profit function of objects, and the profit function is defined based on expected request frequencies of objects and sizes of objects, etc. We adopt LRU, a widely used replacement approach, in NetShark. Description of LRU can be found elsewhere, such as Tanenbaum and Woodhull (1997).

There are two types of cache-consistency approaches: *weak cache consistency* and *strong cache consistency* (Cao and Liu 1998). Weak cache-consistency approaches might return stale web objects to users while strong cache-consistency approaches guarantee return of up-to-date web objects to users. Client polling and TTL (i.e., time to live) are weak cache consistency approaches (Barish and Obraczka 2000). With client polling, cached objects are periodically compared with their original copies. Out-of-date cached objects are dumped and their newest versions are fetched. In TTL, each cached object has a time to live (TTL). When expired, these objects are discarded and their newest versions are fetched. Invalidation callback is a strong cache-consistency approach (Barish and Obraczka 2000) requiring that a server keep track of all the cached objects. The server will notify all the proxies to invalidate their copies if the original object has been modified in the server. Cao and Liu (1998) showed that strong cache-consistency approaches can be realized with little or no extra cost compared to weak cache-consistency approaches. In addition, in a recent study by Yin et al. (2002), strong cache-consistency approaches increase the hit rate by a factor of 1.5 to 3 for large-scale dynamic websites, compared to weak cache-consistency approaches. We adopt a strong cache-consistency approach, the invalidation callback approach, to maintain consistency among storage objects in NetShark. Description of the approach is in Barish and Obraczka (2000).

4. A Data-Mining-Based Prefetching Approach

Similar to web caching, major issues in network storage caching include deciding what storage objects need to be cached (i.e., caching), how to replace old storage objects when there is not enough space for newly cached storage objects (i.e., replacement), and how to keep consistency among copies of storage objects (i.e., consistency). For NetShark, we propose a data-mining-based prefetching approach as the caching approach, and we adopt LRU and invalidation callback as replacement and consistency approaches, respectively.

The proposed prefetching approach consists of two algorithms: offline learning and online caching. Client-request patterns are extracted from Log-DB periodically, using the offline learning algorithm. Based on the patterns extracted, the online caching algorithm caches storage objects. Table 1 summarizes the important notation to be referenced in §§4 and 5.

Table 1 Notation Summary

Notation	Description
p_{intra}, p^i_{intra}	an intrasession pattern
P_{intra}	a set of intrasession patterns, where $p_{\text{intra}} \in P_{\text{intra}}$
$S(p_{\text{intra}})$	the set of all storage objects in p_{intra}
p_{inter}, p^i_{inter}	an intersession pattern
P_{inter}	a set of intersession patterns, where $p_{\text{inter}} \in P_{\text{inter}}$
$S(p_{\text{inter}})$	the set of all storage object sets in p_{inter}
SO, SO_i	a set of storage objects
so_d	a storage object on demand
so_r	a related storage object of so_d, predicted from intra/intersession patterns
SO_S	a set of storage objects requested so far in a session, where $so_d \in SO_S$
$pattern(SO_S)$	an intra/intersession pattern that contains SO_S
$Pattern(SO_S)$	a set of intra/intersession patterns that contain SO_S
$ti(so_r)$	the predicted time interval between so_d and so_r
$\sup(so_r)$	the support of so_r, which measures how likely so_r will be requested after so_d
$nw(so)$	the network over which a storage object, so, is transmitted
$transmission(so, nw(so))$	the time to transmit a storage object, so, over network $nw(so)$, which can be estimated using (1)
$HD(nw(so))$	the average hop delay of network $nw(so)$
$HN(nw(so))$	the number of hops on network $nw(so)$
$D(nw(so))$	the distance of network $nw(so)$
$v(nw(so))$	the signal velocity of network $nw(so)$
$B(nw(so))$	the bandwidth of network $nw(so)$
$size(so)$	the size of a storage object, so
$slack(so_r)$	the slack value of so_r, which can be calculated using (2)
n_{NS}	the total number of storage objects in NetShark
SO_{NS}	the set of all storage objects in NetShark
$fre(so_i)$	the request frequency rate of so_i, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$
$fanout(so_i)$	the fan-out number of so_i, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$
$random(so_i)$	the random factor of so_i, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$

Table 2 Critical Fields in Log-DB

ClientID	SessionID	FileName	RequestTime	
CT1	SN1	*A*	09:19:34	07/25/2001
CT1	SN1	*B*	09:20:34	07/25/2001
CT1	SN1	*C*	09:24:34	07/25/2001
CT1	SN2	*F*	10:57:34	07/25/2001
CT1	SN2	*E*	11:01:34	07/25/2001
CT2	SN3	*F*	09:57:34	07/25/2001
CT2	SN3	*D*	10:00:34	07/25/2001
CT3	SN4	*A*	09:29:34	07/25/2001
CT3	SN4	*B*	09:30:34	07/25/2001
CT3	SN4	*C*	09:32:34	07/25/2001
CT3	SN5	*D*	11:07:34	07/25/2001
CT3	SN5	*E*	11:09:34	07/25/2001

4.1. The Offline Learning Algorithm

We briefly illustrate the structure of Log-DB before introducing the offline learning algorithm. A record in Log-DB has the following fields: ClientID, ClientIP, SessionID, FileName, FileType, FileSize, and RequestTime. As shown in Table 2, critical fields for offline learning include ClientID, SessionID, FileName, and RequestTime, which respectively specify who requested which storage object (e.g., file) at what time and in which session. Here, a session is a sequence of storage-object requests between a client's log-into and log-out of NetShark. It can be easily seen from Table 2 that a client (e.g., CT1) may have several sessions (e.g., SN1 and SN2) and a session (e.g., SN1) may include several storage-object requests (e.g., *A*, *B*, and *C*).

Instead of an arbitrarily chosen look-ahead window used in Padmanabhan and Mogul (1996), the offline learning algorithm uses the session as the basic processing unit to discover client request patterns. Two types of the patterns can be extracted from Log-DB: intrasession patterns P_{intra} and intersession patterns P_{inter}.

Definition 1. An intrasession pattern, p_{intra}, where $p_{\text{intra}} \in P_{\text{intra}}$, reflects a request behavior within a session. It has the format: $\langle so_1 \xrightarrow{\text{time interval}} so_2 \cdots\cdots so_{n-1} \xrightarrow{\text{time interval}} so_n \text{ support}\rangle$. Here $so_1, so_2, \ldots, so_n$ are storage objects requested within the same session. The time interval between two storage objects is the average time interval between the requests of the two storage objects in sessions where the pattern can be found. The support of an intrasession pattern is the fraction of sessions in which the pattern can be found.

For an intrasession pattern, p_{intra}, with the format $\langle so_1 \xrightarrow{\text{time interval}} so_2 \cdots\cdots so_{n-1} \xrightarrow{\text{time interval}} so_n \text{ support}\rangle$, we denote $S(p_{\text{intra}})$ as the set of all storage objects in it, i.e., $S(p_{\text{intra}}) = \{so_i\}$ for $i = 1, 2, \ldots, n$.

Example 1. An intrasession pattern, $p_{\text{intra}} = \langle A \xrightarrow{\text{1 minute}} B \xrightarrow{\text{3 minutes}} C\ 0.4\rangle$, can be discovered from the Log-DB example in Table 2. According to this pattern, storage objects *A*, *B*, and *C* are sequentially requested within a session; such a pattern can be found in 40% (i.e., SN1 and SN4) of the total sessions; and the average time intervals between the requests of *A* and *B* and between the requests of *B* and *C* are one minute and three minutes respectively. In this example, $S(p_{\text{intra}}) = \{A, B, C\}$.

Definition 2. An intersession pattern, p_{inter}, where $p_{\text{inter}} \in P_{\text{inter}}$, reveals a request behavior between sessions. It has the format: $\langle SO_1 \xrightarrow{\text{time interval}} SO_2 \cdots\cdots SO_{n-1} \xrightarrow{\text{time interval}} SO_n \text{ support}\rangle$. Here $SO_1, SO_2, \ldots, SO_n$ are storage-object sets requested in different sessions by the same client. The time interval between two storage-object sets is the average time interval between the requests of the two storage-object sets among clients demonstrating such a pattern. The support of an intersession pattern is the fraction of clients demonstrating such a pattern.

For an intersession pattern, p_{inter}, with the format $\langle SO_1 \xrightarrow{\text{time interval}} SO_2 \cdots\cdots SO_{n-1} \xrightarrow{\text{time interval}} SO_n \text{ support}\rangle$, we denote $S(p_{\text{inter}})$ as the set of all storage object sets in it, i.e., $S(p_{\text{inter}}) = \{SO_i\}$ for $i = 1, 2, \ldots, n$.

Example 2. An intersession pattern, $p_{\text{inter}} = \langle \{A, B\} \xrightarrow{\text{100 minutes}} \{E\}\ 0.66\rangle$, can be learned from the Log-DB example in Table 2. According to this pattern, storage object sets $\{A, B\}$ and $\{E\}$ are sequentially requested in different sessions by the same client; 66% (i.e., CT1 and CT3) of total clients demonstrate such a pattern; and the average time interval between the request of $\{A, B\}$ and the request of $\{E\}$ is 100 minutes (i.e., the average time interval between the request of *B*, the last requested storage object in $\{A, B\}$, and the request of *E*, the first requested storage object in $\{E\}$). In this example, $S(p_{\text{inter}}) = \{\{A, B\}, \{E\}\}$.

We adapt sequential pattern mining to extract intrasession patterns and intersession patterns from Log-DB. Agrawal and Srikant (1995) used a database of customer transactions to define sequential pattern mining. In this database, a customer had k transactions, where $k \geq 1$, and a transaction included the purchases of m items, where $m \geq 1$. As shown in the example in Table 3, each record in the database

Table 3 Customer Transactions for Sequential Pattern Mining

Customer-id	Transaction-id	Item-id	Transaction-time	
CUST1	T1	1	09:19:34	07/25/2001
CUST1	T1	2	09:19:34	07/25/2001
CUST1	T1	4	09:19:34	07/25/2001
CUST1	T2	5	11:21:34	07/26/2001
CUST1	T2	3	11:21:34	07/26/2001
CUST2	T3	1	09:29:34	07/25/2001
CUST2	T3	2	09:29:34	07/25/2001
CUST2	T4	6	10:21:34	07/28/2001
CUST2	T4	3	10:21:34	07/28/2001
CUST3	T5	8	13:21:34	07/30/2001
CUST3	T5	3	13:21:34	07/30/2001

consists of customer-id, transaction-id, item-id, and transaction-time. The database is sorted by increasing customer-id and then by increasing transaction-time.

In sequential pattern mining, an *itemset* is a nonempty set of items. A *sequence* is an ordered list of itemsets and it reveals an intertransaction purchasing behavior among customers. For example, $\{1, 2\}$ is an itemset and $\langle\{1, 2\}, \{3\}\rangle$ is a sequence with the meaning that item 3 was purchased in a transaction after a transaction that had purchased items 1 and 2. A customer supports a sequence if and only if the sequence can be found in the transactions of the customer. The support of a sequence is defined as the fraction of total customers who support the sequence. For example, sequence $\langle\{1, 2\}, \{3\}\rangle$ is supported by customers CUST1 and CUST2 in the example given in Table 3 and its support is 66%. Sequential pattern mining is conducted to discover all large sequences, which are sequences with support larger than a user-defined threshold.

Before extracting intra/intersession patterns from Log-DB, the database is sorted first by increasing ClientID and then by increasing RequestTime. Table 2 gives an example of a sorted Log-DB. To extract intersession patterns from Log-DB, large sequences are discovered using sequential pattern mining. In this case, ClientID, SessionID, and FileName in Log-DB (see Table 2) correspond to customer-id, transaction-id, and item-id (see Table 3) respectively. Every large sequence discovered from Log-DB is an ordered list of storage-object sets and it reveals an intersession request behavior among clients. During the process of discovering large sequences, time intervals between storage object sets in sequences are calculated and incorporated into these sequences. The discovered large sequences with calculated time intervals are intersession patterns.

To discover intrasession patterns, we create pseudo-transactions by assigning a Pseudo-TransactionID for each storage object request (i.e., each record) in Log-DB, as shown in Table 4. Similarly, large sequences are discovered using sequential pattern mining. In this case, SessionID, Pseudo-TransactionID, and FileName in Log-DB (see Table 4) correspond to customer-id, transaction-id, and item (see Table 3) respectively. Every large sequence discovered is an ordered list of storage-object sets with only one element because each pseudo-transaction has only one storage-object request. Large sequences reveal interpseudo-transaction-request behaviors among sessions (i.e., intrasession request behaviors). Similarly, time intervals between storage objects in sequences are calculated and incorporated into these sequences. The discovered sequences with calculated time intervals are intrasession patterns.

Table 4 Adding the Pseudo-TransactionID Field into Log-DB

ClientID	SessionID	Pseudo-TransactionID	FileName	RequestTime	
CT1	SN1	1	*A*	09:19:34	07/25/2001
CT1	SN1	2	*B*	09:20:34	07/25/2001
CT1	SN1	3	*C*	09:24:34	07/25/2001
CT1	SN2	4	*F*	10:57:34	07/25/2001
CT1	SN2	5	*E*	11:01:34	07/25/2001
CT2	SN3	6	*F*	09:57:34	07/25/2001
CT2	SN3	7	*D*	10:00:34	07/25/2001
CT3	SN4	8	*A*	09:29:34	07/25/2001
CT3	SN4	9	*B*	09:30:34	07/25/2001
CT3	SN4	10	*C*	09:32:34	07/25/2001
CT3	SN5	11	*D*	11:07:34	07/25/2001
CT3	SN5	12	*E*	11:09:34	07/25/2001

We summarize the offline learning algorithm in Figure 4. We denote the number of records in Log-DB as n_{rec}. The offline learning algorithm includes four parts: sorting Log-DB, assigning Pseudo-TransactionID, discovering P_{inter}, and discovering P_{intra}. Sorting Log-DB needs $O(n_{\text{rec}} \log n_{\text{rec}})$ time. Assigning Pseudo-TransactionID requires $O(n_{\text{rec}})$ time. During the discovery of P_{inter}, sequential pattern mining goes through Log-DB α rounds, where α is determined by the maximum size of storage-object sets and the maximum size of large sequences (Agrawal and Srikant 1995). Therefore, the time complexity of discovering P_{inter} is $O(n_{\text{rec}} \times \alpha)$. Usually, α is much less than n_{rec}. Hence, the time complexity of discovering P_{inter} is $O(n_{\text{rec}})$. Similarly, the time complexity of discovering P_{intra} is $O(n_{rec})$. Combining the time complexities of all these parts, the time complexity of the offline learning algorithm is $O(n_{\text{rec}} \log n_{\text{rec}})$. The main I/O cost of the algorithm is the I/O of Log-DB. Hence, the I/O cost of the algorithm is determined by such factors as the maximum size of storage-object sets and the maximum size of large sequences.

4.2. The Online Caching Algorithm

A storage object on demand, so_d, is one that currently has been requested by a client. The online caching algorithm caches so_d as well as its related storage objects, $\{so_r\}$, predicted from intrasession patterns (e.g., storage objects predicted to be requested right after so_d within a session) and intersession patterns (e.g., storage objects predicted to be requested in

Input: Log-DB
Output: P_{intra}: intrasession patterns,
P_{inter}: intersession patterns.

sort Log-DB by increasing ClientID and then by increasing RequestTime;
assign a Pseudo-TransactionID for each storage object request (i.e., record) in Log-DB;
apply sequential pattern mining to Log-DB to discover P_{inter};
apply sequential pattern mining to Log-DB to discover P_{intra}.

Figure 4 The Offline Learning Algorithm

a session after the session requesting so_d). Although prefetching algorithms are more efficient than caching on demand (Kroeger et al. 1997), these algorithms have a well-known shortcoming of increasing network traffic between servers and proxies (e.g., between Sharks in NetShark) compared with caching on demand (Padmanabhan and Mogul 1996). The online caching algorithm addresses the network-traffic-increase problem by dividing related storage objects $\{so_r\}$ into two categories, urgent and wait, based on time intervals in intra/intersession patterns. Related storage objects in the urgent category, denoted as $\{so_{r_u}\}$, where $\{so_{r_u}\} \subseteq \{so_r\}$, are cached immediately, while related storage objects in the wait category, denoted as $\{so_{r_w}\}$, where $\{so_{r_w}\} \subseteq \{so_r\}$, are cached when the network traffic between Home and Storage Sharks is below a user-defined threshold.

As shown in Figure 5, the online caching algorithm consists of five parallel procedures: Process_Requests, Predict_Related_Objects, Process_Urgent_Objects, Process_Wait_Objects, and Transfer_Objects.

Procedure Process_Requests is triggered whenever there is a storage-object request. It is responsible for returning the storage object on demand, so_d, to the requesting client. It also caches so_d if it is not in the Home Shark of the requesting client.

Procedure Predict_Related_Objects is also triggered by a storage-object request. Input of the procedure includes the set of storage objects requested so far in a session, SO_S, which contains the storage object on demand, so_d, i.e., $so_d \in SO_S$, intrasession patterns P_{intra} and intersession patterns P_{inter}. The procedure first predicts related storage objects $\{so_r\}$ of so_d by searching for intrasession patterns P_{intra} and intersession patterns P_{inter} that contain SO_S.

Definition 3. SO_S is *contained* in an intrasession pattern, p_{intra}, iff,

$SO_S \subset S(p_{\text{intra}})$; and

the request sequence of storage objects in SO_S is the same as their request sequence in p_{intra}.

Definition 4. SO_S is *contained* in an intersession pattern, p_{inter}, iff $SO_S \in S(p_{\text{inter}})$.

We denote an intra/intersession pattern that contains SO_S as $pattern(SO_S)$ and the set of all intra/intersession patterns that contain SO_S as $Pattern(SO_S)$. The storage objects requested right after SO_S in $Pattern(SO_s)$ are related storage objects $\{so_r\}$ of so_d. Two attributes of a related storage object so_r, $ti(so_r)$ and $\sup(so_r)$, can also be predicted from $Pattern(SO_S)$. $ti(so_r)$ is the predicted time interval between so_d and so_r, which is the time interval between so_d and so_r in the $pattern(SO_S)$ that includes so_r. $\sup(so_r)$ is the support of so_r, which measures how likely so_r will be requested after so_d; $\sup(so_r)$ is the support of the $pattern(SO_S)$ that includes so_r.

Example 3. Given the following intra/intersession patterns:

p^0_{intra}: $\langle B \xrightarrow[2\text{ minutes}]{} A \xrightarrow[5\text{ seconds}]{} I\ 0.08\rangle$,
p^1_{intra}: $\langle A \xrightarrow[1\text{ minute}]{} B \xrightarrow[5\text{ seconds}]{} C \xrightarrow[1\text{ minute}]{} K\ 0.12\rangle$,
p^2_{intra}: $\langle A \xrightarrow[1\text{ minute}]{} B \xrightarrow[4\text{ minutes}]{} F\ 0.1\rangle$,

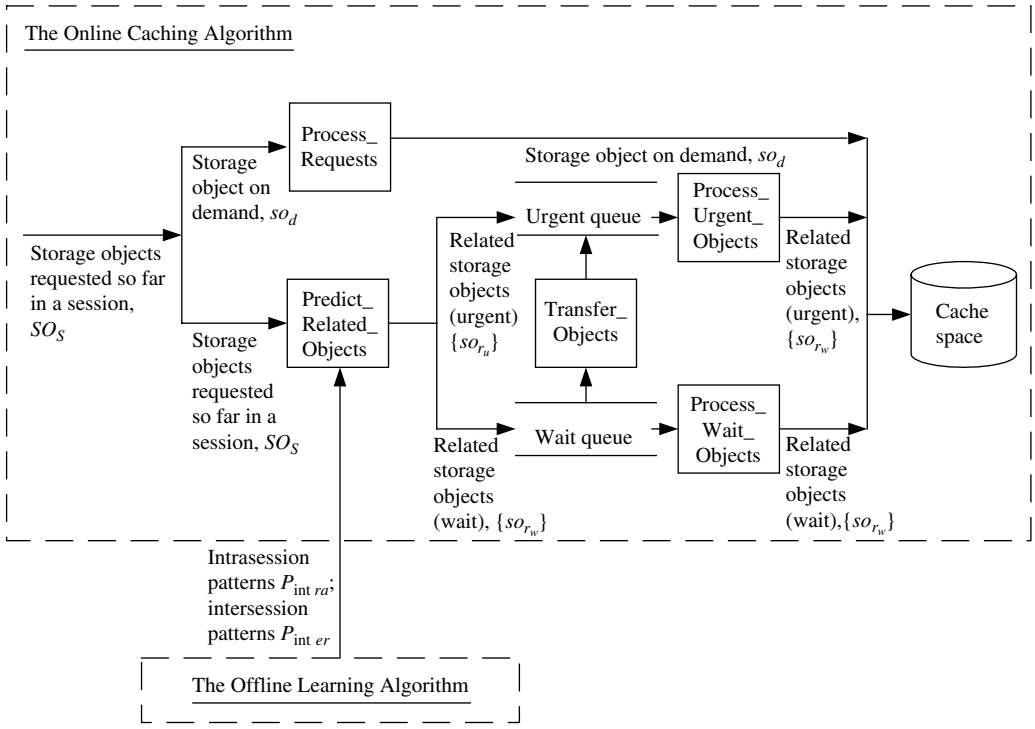

Figure 5 The Online Caching Algorithm

p^0_{inter}: $\langle \{A, B\} \xrightarrow[100 \text{ minutes}]{} \{E, H\}\ 0.12 \rangle$, and
p^1_{inter}: $\langle \{D, F\} \xrightarrow[10 \text{ minutes}]{} \{G\}\ 0.1 \rangle$,
if $SO_S = \{A, B\}$, $so_d = B$ and the request sequence of storage objects in SO_S is first A then B (i.e., $A \to B$), then SO_S is contained in p^1_{intra} by Definition 3, as,

$SO_S \subset S(p^1_{\text{intra}})$, where $S(p^1_{\text{intra}}) = \{A, B, C, K\}$;

the request sequence of storage objects in SO_S, $A \to B$, is the same as their request sequence in p^1_{intra}. Similarly, SO_S is also contained in p^2_{intra} by Definition 3. SO_S is contained in p^0_{inter} by Definition 4, as $SO_S \in S(p^0_{\text{inter}})$, where $S(p^0_{\text{inter}}) = \{\{A, B\}, \{E, H\}\}$. Therefore, $Pattern(SO_S) = \{p^1_{\text{intra}}, p^2_{\text{intra}}, p^0_{\text{inter}}\}$. The storage objects requested right after SO_S in p^1_{intra}, p^2_{intra}, and p^0_{inter} are C, F, E, and H. Hence, $\{so_r\} = \{C, F, E, H\}$. From $Pattern(SO_S)$, we also get:

so_r	$ti(so_r)$	$\sup(so_r)$
C	5 seconds	0.12
F	4 minutes	0.1
E	100 minutes	0.12
H	100 minutes	0.12

Related storage objects $\{so_r\}$ are distributed into the urgent queue or the wait queue based on their slack values $slack(so_r)$. We introduce the following before defining $slack(so_r)$. We denote $nw(so)$ as the network over which a storage object so is transmitted. The time to transmit so over network $nw(so)$, $transmission(so, nw(so))$, can be estimated using the following formula (Bertsekas and Gallagher 1995):

$$transmission(so, nw(so)) = HD(nw(so)) \times HN(nw(so)) + \frac{D(nw(so))}{v(nw(so))} + \frac{size(so) \times 8}{B(nw(so))}. \quad (1)$$

Here $HD(nw(so))$ is the average hop delay of network $nw(so)$, $HN(nw(so))$ is the number of hops on network $nw(so)$, $D(nw(so))$ is the distance of network $nw(so)$, $v(nw(so))$ is the signal velocity of network $nw(so)$ (see Table 5 for some examples of $v(\cdot)$), $B(nw(so))$ is the bandwidth of network $nw(so)$, and $size(so)$ is the size of so.

$slack(so_r)$ is defined as

$$slack(so_r) = ti(so_r) - transmission(so_r, nw(so_r)). \quad (2)$$

In (2), $nw(so_r)$ refers to the network between the Storage Shark of so_r and the Home Shark of the client who could request so_r, while $slack(so_r)$ reflects the degree of urgency to cache a related storage object so_r. Related storage objects with nonpositive slack values are placed in the urgent queue and need to be cached immediately; related storage objects with positive slack values are placed in the wait queue and cached when the network traffic between Home and Storage Sharks is below a user-defined threshold. Related storage objects in both the urgent queue and the wait queue are sorted by their descending supports (i.e., $\sup(so_r)$) then by their ascending slack values (i.e., $slack(so_r)$).

Table 5 Signal Velocity of Different Media (Steinke 2001)

Medium	Signal velocity (km/second)
Phone line	160,935
Copper (category 5)	231,000
Optical fiber	205,000
Air	299,890

Example 4. For related storage objects C, E, F, and H, predicted in Example 3, $size(C) = 500$ kbytes, $size(E) = 50$ k bytes, $size(F) = 100$ k bytes, and $size(H) = 100$ k bytes. Given the network over which these storage objects were transferred, $nw(so_r)$, and its characteristics: $HD(nw(so_r)) = 0.015$ second, $HN(nw(so_r)) = 18$, $D(nw(so_r)) = 1000$ km, $v(nw(so_r)) = 231000$ km/second, and $B(nw(so_r)) = 500$ k bps, slack values of these storage objects can be calculated as follows:

$$\begin{aligned} &transmission(C, nw(so_r)) \\ &\quad = 0.015 \times 18 + \frac{1000}{231000} + \frac{500 \text{ k} \times 8}{500 \text{ k}} \\ &\quad = 8.2743 \text{ seconds} \\ &slack(C) = ti(C) - transmission(C, nw(so_r)) \\ &\quad = 5 - 8.2743 = -3.2743 \text{ seconds}; \end{aligned}$$

similarly,

$$\begin{aligned} slack(E) &= ti(E) - transmission(E, nw(so_r)) \\ &= 5998.926 \text{ seconds} \\ slack(F) &= ti(F) - transmission(F, nw(so_r)) \\ &= 238.1257 \text{ seconds} \\ slack(H) &= ti(H) - transmission(H, nw(so_r)) \\ &= 5998.126 \text{ seconds}. \end{aligned}$$

According to the slack values calculated, related storage object C is placed in the urgent queue, while related storage objects E, F, and H are put in the wait queue.

Procedure Predict_Related_Objects is summarized in Figure 6. The time complexity of the procedure is $O(n_{\text{intra}} + n_{\text{inter}})$, where n_{intra} is the number of intrasession patterns and n_{inter} is the number of intersession patterns.

Procedure Process_Urgent_Objects is triggered whenever the urgent queue is not empty. The procedure caches related storage objects in the urgent queue $\{so_{r_u}\}$. Procedure Process_Wait_Objects is triggered if the wait queue is not empty and the network

```
Event: a storage object request /*event that triggers the
         procedure*/
Input: P_intra: intrasession patterns,
       P_inter: intersession patterns,
       SO_S: the set of storage objects requested so far in a
         session, where so_d ∈ SO_S.

For each intrasession pattern p_intra, where p_intra ∈ P_intra
  If (SO_S is contained in p_intra)
    so_r = the storage object in p_intra requested right after SO_S;
    If (so_r not in clients' Home Shark)
      calculate slack(so_r);
      If (slack(so_r) ≤ 0)
        urgent.add(so_r); /*add so_r into the urgent queue*/
      else
        wait.add(so_r); /*add so_r into the wait queue*/
      End if
    End if
  End if
End for
For each intersession pattern p_inter, where p_inter ∈ P_inter
  If (SO_S is contained in p_inter)
    {so_r} = the storage object set in p_inter requested right after SO_S;
    For each so_r, where so_r ∈ {so_r}
      If (so_r not in clients' Home Shark)
        calculate slack(so_r);
        If (slack(so_r) ≤ 0)
          urgent.add(so_r); /*add so_r into the urgent queue*/
        else
          wait.add(so_r); /*add so_r into the wait queue*/
        End if
      End if
    End for
  End if
End for
```

Figure 6 Procedure Predict_Related_Objects

traffic between Sharks is below a user-defined threshold. The procedure caches related storage objects in the wait queue $\{so_{r_w}\}$. Procedure Transfer_Objects transfers a related storage object in the wait queue so_{r_w} to the urgent queue whenever its wait expiration time, $wet(so_{r_w})$, has passed the current system time. Here, $wet(so_{r_w})$ is

$$wet(so_{r_w}) = \text{time}(so_d) + slack(so_{r_w}), \tag{3}$$

where $\text{time}(so_d)$ is the request time for the storage object on demand, so_d.

5. The Network-Storage-Caching (NSC) Simulator

Simulation is a major tool to evaluate different caching approaches (Davison 2001). Simulations used in previous caching research, such as Kroeger et al. (1997), and simulators developed for caching performance evaluation, such as NCS (Davison 2001) and PROXIM (Feldmann et al. 1999), are trace-driven simulations based on specific trace logs. Using trace-driven simulations, it is hard to isolate, examine, and explain the impacts of system characteristics such as the size of cache space on the performance of different caching approaches. Unlike trace-driven simulators, the NSC simulator is based on general and proven characteristics of trace logs and provides researchers with flexibility to manipulate parameters of these characteristics. The NSC simulator simulates how NetShark provides storage services for clients distributed in m geographically separated regions. At the beginning of an experiment, storage objects are randomly allocated to m Sharks.

According to Law and Kelton (2000), the abstraction level of a simulation model is determined by factors such as the purpose of simulation. In this research, the purpose of simulation is to compare the performance between the proposed data-mining-based caching algorithm and the two existing caching algorithms. All these caching algorithms run on the application level of a network. They are used to determine which storage objects (e.g., files), instead of which TCP segments, IP packets, or ATM cells, need to be cached. Therefore, as shown in Figure 7, the NSC simulator simulates each network in the simulation model as a G/G/1 queue, instead of a network of queues that simulates each network switch as a queue. The NSC simulator simulates request generation and three types of servers: Client-Shark networks, Shark networks, and Shark-Shark networks. There may be more than one server in each server type. These servers serve requests and deliveries of storage objects according to the first-come-first-served (FCFS) queuing discipline. In §5.1, we explain the request-generation model, and §5.2 describes the server queuing models.

5.1. The Request-Generation Model

A request-generation procedure consists of two steps: (1) generating session-begin messages and session-end messages; and (2) generating requests within sessions. To generate session-begin/end messages, we assume the interarrival times between sessions and the durations of sessions to be exponentially distributed. Request generation within a session begins as soon as the session-begin message has been generated and the request-generation process stops as soon as the session-end message has been generated.

We call requests within a session as a *request stream*. We assume that the interarrival times between requests in a request stream follow a lognormal distribution (Paxson and Floyd 1995). As shown in Figure 8, a request stream is generated based on the request-frequency rates of storage objects and the correlations between storage objects. Hence, the request frequency rates of storage objects and the correlations between storage objects need to be generated before a meaningful request stream can be generated.

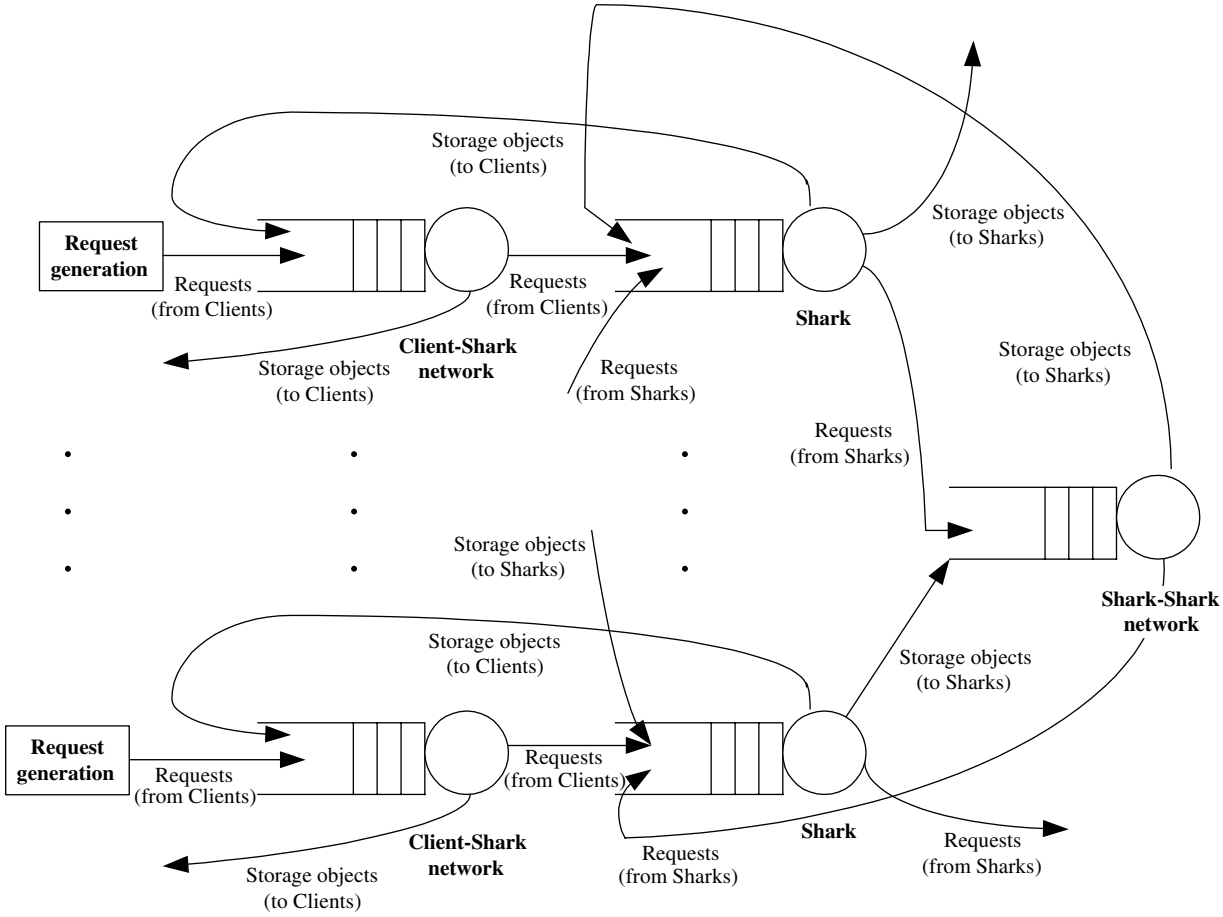

Figure 7 The NSC Simulator

The request-frequency rates of storage objects approximately follow Zipf's law, in which the request frequency rate of the kth most frequently requested storage object is proportional to $1/k$ (Breslau et al. 1999). We denote the total number of storage objects in NetShark as n_{NS} and the set of all storage objects in NetShark as SO_{NS}, where $SO_{NS} = \{so_i\}$ for $i = 1, 2, \ldots, n_{NS}$. For a storage object in NetShark so_i, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$, we denote $rank(so_i)$ as the rank of its request frequency rate among all storage objects in NetShark. Let $rank(so_i) = k$ if so_i is the kth most frequently requested storage object in NetShark. According to Breslau et al. (1999), the request frequency rate of so_i, $fre(so_i)$, is

$$fre(so_i) = \frac{c}{rank(so_i)}, \quad \text{where } c = \frac{1}{\sum_{j=1}^{n_{NS}}(1/j)}. \tag{4}$$

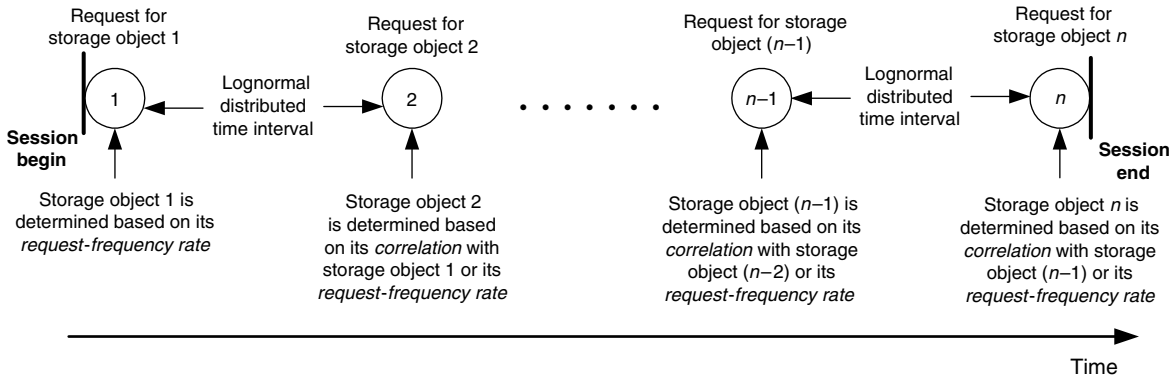

Figure 8 Request-Stream Generation

In (4), c can be approximated as

$$c \approx \frac{1}{\ln(n_{NS}) + C + 1/(2n_{NS})}. \quad (5)$$

Here, C is the Euler constant and $C \approx 0.577$. Using (4) and (5), the request-frequency rates of storage objects can be generated by the NSC simulator.

To simulate the situation in which some storage objects are often requested together within sessions, the NSC simulator generates correlations between storage objects. For a storage object in NetShark so_i, we name a storage object requested right after so_i within the same session as the next requested storage object of so_i. There are two types of the next requested storage object of so_i:

a correlated storage object of so_i: an object requested frequently right after so_i;

a randomly requested storage object of so_i: an object requested infrequently right after so_i.

For a storage object so_i, we denote $fanout(so_i)$, namely the fan-out number of so_i, as the number of the correlated storage objects of so_i; $c_j(so_i)$ denotes a correlated storage object of so_i and $\{c_j(so_i)\}$ denotes the set of all correlated storage objects of so_i, where $c_j(so_i) \in SO_{NS}$ for $j = 1, 2, \ldots, fanout(so_i)$. $P(c_j(so_i) \mid so_i)$ denotes the correlation between $c_j(so_i)$ and so_i, which is the conditional probability of requesting $c_j(so_i)$ right after requesting so_i within the same session. We denote $random(so_i)$, namely the random factor of so_i, as the conditional probability of requesting a randomly requested storage object of so_i right after requesting so_i, where,

$$random(so_i) + \sum_{j=1}^{fanout(so_i)} P(c_j(so_i) \mid so_i) = 1. \quad (6)$$

Example 5. As shown in Figure 9, storage object so_4 has two correlated storage objects, so_7 and so_9, i.e., $fanout(so_4) = 2$, $c_1(so_4) = so_7$ and $c_2(so_4) = so_9$. The correlation between so_7 and so_4, $P(so_7 \mid so_4)$, is 0.35 and the correlation between so_9 and so_4, $P(so_9 \mid so_4)$, is 0.55. The conditional probability of requesting a randomly requested storage object of so_4 right after requesting so_4, $random(so_4)$, is 0.10.

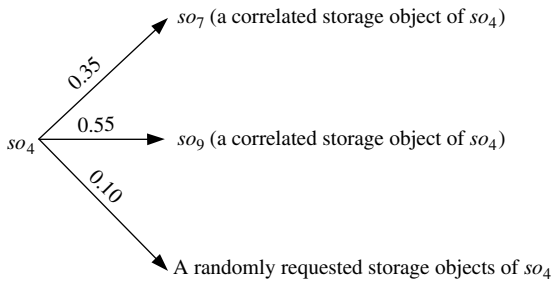

Figure 9 Correlations Between Storage Objects

Input: $fanout(so_i)$: the fan-out number of so_i
$random(so_i)$: the random factor of so_i
Output: $\{c_j(so_i)\}$: the set of all correlated storage objects of so_i, where $c_j(so_i) \in SO_{NS}$ for $j = 1, 2, \ldots, fanout(so_i)$
$P(c_j(so_i) \mid so_i)$: the correlation between $c_j(so_i)$ and so_i

```
corr = 1 − random(so_i); /*probability of requesting correlated
                          storage objects of so_i*/
{c_j(so_i)} = φ;
While (fanout(so_i) ≥ 1)
  x = rand(0, 1); /*x ~ uniform distribution over (0, 1)*/
  /*pick a storage object based on its request frequency rate*/
  select k, where x ∈ [F(k − 1), F(k));
  If so_k ∈ {c_j(so_i)} /*so_k has been selected before*/
    continue;
  End if
  {c_j(so_i)} = {c_j(so_i)} ∪ {so_k};
  If (fanout(so_i) > 1)
    P(so_k | so_i) = rand(0, corr);
    corr = corr − P(so_k | so_i);
  else
    P(so_k | so_i) = corr; /*last correlated storage object of so_i*/
  End if
  fanout(so_i) − −;
End while
```

Figure 10 Generate Correlated Storage Objects of a Storage Object

Figure 10 gives an algorithm to generate all correlated storage objects of so_i, $\{c_j(so_i)\}$, and their correlations with so_i, $P(c_j(so_i) \mid so_i)$, given the fan-out number of so_i, $fanout(so_i)$, and the random factor of so_i, $random(so_i)$. In the algorithm, $F(k)$ is denoted as the kth cumulative request frequency rate, where $k = 0, 1, 2, \ldots, n_{NS}$:

$$F(k) = \begin{cases} \sum_{m \le k} fre(so_m) & k = 1, 2, \ldots, n_{NS}, so_m \in SO_{NS} \\ & \text{for } m = 1, 2, \ldots, n_{NS} \\ 0 & k = 0. \end{cases} \quad (7)$$

In the NSC simulator, $fanout(so_i)$ is simulated using a Poisson random variable with parameter λ, and $random(so_i)$ is simulated using a uniform random variable over the interval $(0, r)$, where $0 < r < 1$. λ and r can be manipulated to change correlations between storage objects. Running the algorithm for every storage object in NetShark, correlations between storage objects in NetShark can be generated.

Figure 11 gives the request-stream-generation algorithm. In the algorithm, $F(t, so_i)$ denotes the tth cumulative correlation of so_i, where $t = 0, 1, 2, \ldots, fanout(so_i)$:

$$F(t, so_i) = \begin{cases} \sum_{j \le t} P(c_j(so_i) \mid so_i) & \\ & t = 1, 2, \ldots, fanout(so_i) \\ 0 & t = 0. \end{cases} \quad (8)$$

Example 6. For correlations given in Example 5, we have

$$F(1, so_4) = P(c_1(so_4) \mid so_4) = P(so_7 \mid so_4) = 0.55;$$

Event: a session-begin message has been received
Input: $fre(so_i)$: request frequency rates of storage objects, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$,
$P(c_j(so_i) \mid so_i)$: correlations between storage objects, where $so_i \in SO_{NS}$ for $i = 1, 2, \ldots, n_{NS}$ and $c_j(so_i) \in SO_{NS}$ for $j = 1, 2, \ldots, fanout(so_i)$.
Output: a request stream

/*generate the first requested storage object based on its request frequency rate*/
$x = rand(0, 1)$; /*$x \sim$ uniform distribution over $(0, 1)$*/
select k, where $x \in [F(k-1), F(k))$;
place the request for so_k in the request stream;
current $= so_k$; /*currently requested storage object*/
While (session not end) /*not receiving a session-end message*/
  $x = rand(0, 1)$;
  If $(x < (1 - random(\text{current})))$
  /*generate a correlated storage object of the currently requested storage object*/
    select t, where $x \in [F(t-1, \text{current}), F(t, \text{current}))$;
    place the request for $c_t(current)$ in the request stream;
    current $= c_t(current)$;
  else
  /*generate a randomly requested storage object based on its request frequency rate*/
    $x = rand(0, 1)$;
    select k, where $x \in [F(k-1), F(k))$;
    place the request for so_k in the request stream;
    current $= so_k$;
  End if
End while

Figure 11 The Request-Stream-Generation Algorithm

$$F(2, so_4) = P(c_1(so_4) \mid so_4) + P(c_2(so_4) \mid so_4)$$
$$= P(so_7 \mid so_4) + P(so_9 \mid so_4) = 0.9.$$

The request-stream-generation algorithm is triggered whenever there is a session-begin message. The first requested storage object in a request stream is generated based on its request frequency rate. The rest of the requested storage objects are generated using a loop that stops when a session-end message has been received. Within the loop,

if a randomly generated number (i.e., uniformly distributed) is less than one minus the random factor (i.e., $random(\cdot)$) of the currently requested storage object, a storage object is generated based on its correlation with the currently requested storage object;

otherwise, a storage object is generated based on its request frequency rate.

5.2. The Server Queuing Models

To simulate the servers, the sizes of storage objects need to be generated first. According to Paxson and Floyd (1995), the sizes of storage objects follow a Pareto distribution, which is a heavy-tailed distribution with distribution function

$$F(x) = 1 - \left(\frac{\alpha}{x}\right)^{\beta} \quad x \geq \alpha. \tag{9}$$

Here, α is the location parameter, which is the minimum size of all storage objects, and β is the shape parameter with the value ranging from 0.9 to 1.4 (Paxson and Floyd 1995). Using (9), the sizes of storage objects can be generated.

5.2.1. Client-Shark/Shark-Shark Network Servers. In the NSC simulator, Client-Shark network servers and Shark-Shark network servers are simulated using G/G/1 queues. As shown in Figure 12, both Client-Shark network servers and Shark-Shark network servers transfer requests and storage objects. The distribution of request interarrival times at a Client-Shark network server is determined by such factors as the distribution of session interarrival times and the distribution of request interarrival times at its connected request generation model. The distribution of request interarrival times at a Shark-Shark network server is determined by such factors as hit rates of its connected Shark servers. Request transmission time of Client-Shark and Shark-Shark network servers can be estimated using (1). The distributions of storage-object interarrival times at Client-Shark and Shark-Shark network servers are determined by factors such as waiting times and service times of their connected Shark servers. The storage-object transmission time of Client-Shark and Shark-Shark network servers can be estimated using (1).

5.2.2. Shark Server. As shown in Figure 13, a Shark server receives requests from clients or other Shark servers and retrieve storage objects to fulfill these requests. Requests that cannot be fulfilled and

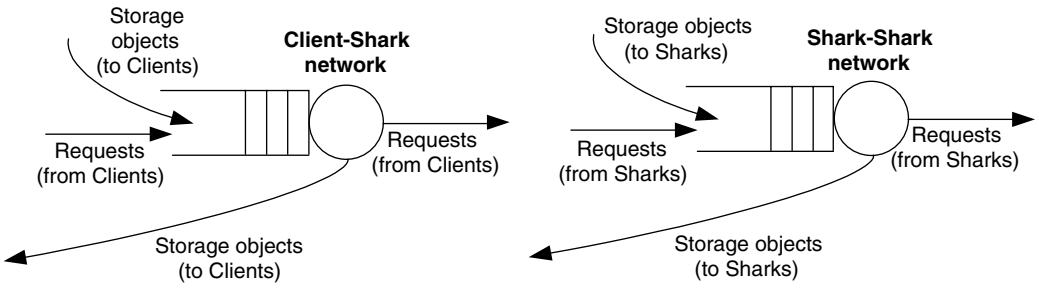

Figure 12 Client-Shark/Shark-Shark Network Servers

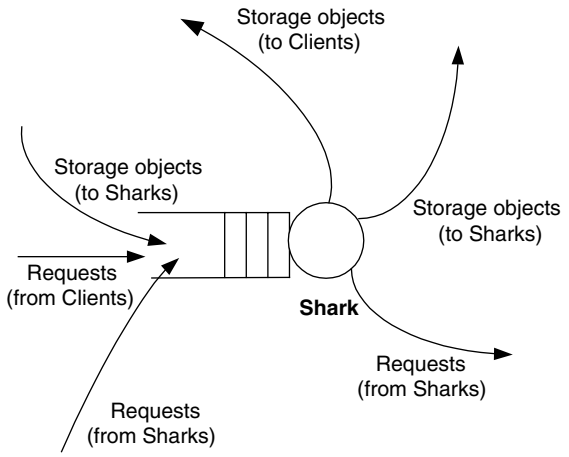

Figure 13 A Shark Server

requests for prefetched storage objects are sent to other Shark servers. A Shark server also receives cached and prefetched storage objects from other Shark servers and stores them. In the NSC simulator, a Shark server is simulated using a G/G/1 queue. The distributions of request and storage object interarrival times at a Shark server are determined by such factors as request and storage object interarrival times at its connected Client-Shark/Shark-Shark network servers and request and storage object transmission times of these network servers. The service time of a Shark server consists of two parts: the computational time of the online caching algorithm running at the Shark server, which is considered as a constant in the NSC simulator, and the time to retrieve/store a storage object, *so*, from/to a Shark server, $IO(so)$, defined as

$$IO(so) = \frac{size(so)}{dio}. \tag{10}$$

Here *dio* is the disk I/O speed of Sharks.

6. Simulation Results

Using the NSC simulator described in §5, we compared three caching approaches: the data-mining-based prefetching approach described in this paper, caching on demand, and a widely used prefetching approach—the popularity-based prefetching approach (Dias et al. 1996, Markatos and Chronaki 1998). Before presenting the simulation results, we list values of major simulation parameters in Table 6. The means of session interarrival times and the means of request interarrival times were calculated from an FTP log. By running network-routing software—VisualRoute—we found that the number of hops between clients and servers within the same region ranged from three (e.g., clients and servers are in an organization's Intranet) to nine and the number of hops between clients and servers in different regions ranged from 14 to 22 or even more (e.g., clients and servers are in different countries). In NetShark, clients and their Home Sharks are in the same region while Sharks are distributed across different regions. Hence, we set the number of hops between clients and their Home Sharks to six (average of three and nine) and the number of hops between Sharks to 18 (average of 14 and 22). We assume that clients and Sharks are connected using Intranet/Internet and Sharks are connected using ATM PSDN.

Table 6 Values of Major Simulation Parameters

Parameter	Value
The simulation time	12 hours
The number of regions	6
The number of Sharks	6
The number of storage objects	20,000
The location parameter of the Pareto distribution for storage object size (α)	30 K bytes
The shape parameter of the Pareto distribution for storage object size (β)	1.06
Mean of session interarrival times	80.9844 seconds
Mean of request interarrival times	5.9894 seconds
The random factors of storage objects (*random*(·))	uniformly distributed over (0, 0.3)
Mean of the fan-out numbers of storage objects (*fanout*(·))	100
The request size	50 bytes
The bandwidth of Client-Shark network ($B(nw_{CS})$)	500 K bps
The bandwidth of Shark-Shark network ($B(nw_{SS})$)	155 M bps
Average hop delay of Client-Shark/ Shark-Shark network (*HD*(·))	0.015 second
The number of hops on Client-Shark network ($HN(nw_{CS})$)	6
The number of hops on Shark-Shark network ($HN(nw_{SS})$)	18
The distance of Client-Shark network ($D(nw_{CS})$)	20 km
The distance of Shark-Shark network ($D(nw_{SS})$)	1000 km
The signal velocity of Client-Shark network ($v(nw_{CS})$)	231,000 km/second
The signal velocity of Shark-Shark network ($v(nw_{SS})$)	205,000 km/second
The disk I/O speed of Sharks (*dio*)	43.1 M bytes/second
Maximum cache size	10% of the total storage object sizes

To compare the three caching approaches on the same benchmark, we used the same replacement approach, LRU, when comparing them. Each experiment consists of 10 simulation runs. Tables 7 and 8 report mean hit rates (Measurement 2) and mean client-perceived latencies (Measurement 1) of the three caching approaches as the cache size increased from 22% of the maximum cache size to 100% of the maximum cache size with an 11% gap. Compared with caching on demand, the popularity-based prefetching approach improves mean hit rate by an average of 20.3%, and the data-mining-based prefetching approach improves mean hit rate by an average of 32.2%. Compared with the popularity-based prefetching approach, the data-mining-based

Table 7 Hit Rate Comparison Among the Three Caching Approaches

	Caching on demand	Popularity-based prefetching		Data-mining-based prefetching		
Percentage of the maximum cache size (%)	Mean hit rate	Mean hit rate	Mean difference (1)**	Mean hit rate	Mean difference (2)**	Mean difference (3)**
22	0.3812	0.5118	0.1306	0.5286	0.1474	0.0168
33	0.4301	0.5381	0.1080	0.5835	0.1534	0.0454
44	0.4807	0.5706	0.0899	0.6263	0.1456	0.0557
55	0.4876	0.5706	0.0830	0.6349	0.1473	0.0643
66	0.4881	0.5706	0.0825	0.6373	0.1492	0.0667
77	0.4885	0.5706	0.0821	0.6379	0.1494	0.0673
88	0.4885	0.5706	0.0821	0.6390	0.1505	0.0684
100	0.4885	0.5706	0.0821	0.6390	0.1505	0.0684

Notes. Mean difference (1): mean hit rate difference between popularity-based approach and caching on demand

Mean difference (2): mean hit rate difference between data-mining-based approach and caching on demand

Mean difference (3): mean hit rate difference between data-mining-based approach and popularity-based approach

**all mean differences statistically significant at 0.001 level using paired-*t* test

prefetching approach improves mean hit rate by an average of 10.0%. All the improvements are statistically significant at 0.001 level. Compared with caching on demand, the popularity-based prefetching approach reduces mean perceived-latency by an average of 5.3%, and the data-mining-based prefetching approach reduces mean perceived-latency by an average of 8.1%. Compared with the popularity-based prefetching approach, the data-mining-based prefetching approach reduces mean perceived latency by an average of 3.0%. All the perceived-latency savings are statistically significant at 0.001 level.

Table 8 Client Perceived-Latency Comparison Among the Three Caching Approaches

	Caching on demand	Popularity-based prefetching		Data-mining-based prefetching		
Percentage of the maximum cache size (%)	Mean perceived latency (second)	Mean perceived latency (second)	Mean difference (1)**	Mean perceived latency (second)	Mean difference (2)**	Mean difference (3)**
22	4.2786	3.9888	−0.2898	3.9340	−0.3446	−0.0548
33	4.2134	3.9775	−0.2359	3.9104	−0.3030	−0.0671
44	4.1833	3.9581	−0.2252	3.8538	−0.3295	−0.1043
55	4.1626	3.9581	−0.2045	3.8244	−0.3382	−0.1337
66	4.1607	3.9581	−0.2026	3.8190	−0.3417	−0.1391
77	4.1588	3.9581	−0.2007	3.8136	−0.3452	−0.1445
88	4.1588	3.9581	−0.2007	3.8101	−0.3487	−0.1480
100	4.1588	3.9581	−0.2007	3.8101	−0.3487	−0.1480

Notes. Mean difference (1): mean perceived-latency difference between popularity-based approach and caching on demand

Mean difference (2): mean perceived-latency difference between data-mining-based approach and caching on demand

Mean difference (3): mean perceived-latency difference between data-mining-based approach and popularity-based approach

**all mean differences statistically significant at 0.001 level using paired-*t* test

Table 9 Shark-Shark Response Time Comparison Between the Data-Mining-Based Approach and the Popularity-Based Approach

	Popularity-based prefetching	Data-mining-based prefetching	
Percentage of the maximum cache size (%)	Mean Shark-Shark response time (second)	Mean Shark-Shark response time (second)	Mean difference**
22	0.337266	0.334963	−0.0023
33	0.328977	0.326443	−0.00253
44	0.324755	0.321587	−0.00317
55	0.324493	0.320855	−0.00364
66	0.324493	0.319953	−0.00454
77	0.324493	0.319238	−0.00525
88	0.324493	0.319281	−0.00521
100	0.324493	0.319281	−0.00521

Notes. Mean difference: mean Shark-Shark response time difference between data-mining-based approach and popularity-based approach

**all mean differences statistically significant at 0.001 level using paired-*t* test

It is not surprising that both prefetching approaches outperform caching on demand, as shown previously in (Kroeger et al. 1997). Between the two prefetching approaches, the data-mining-based prefetching approach outperforms the popularity-based prefetching approach for the following reasons. The popularity-based prefetching approach prefetches storage objects based only on their popularities (i.e., request frequencies) while the data-mining-based prefetching approach prefetches storage objects based on both their popularities (i.e., support of intra/intersession patterns) and their correlations. Hence, the latter is more effective in prefetching storage objects that will be requested by clients. Furthermore, the data-mining-based prefetching approach considers the network-traffic-increase problem associated with prefetching approaches and tries to cache storage objects when the network traffic is not heavy. Hence, as shown in Table 9, the data-mining-based prefetching approach reduces mean Shark-Shark response time (Measurement 3), when compared with the popularity-based prefetching approach, and the time savings are statistically significant at 0.001 level.

7. Conclusion and Future Work

Network storage is a key technique in solving the local-storage-shortage problem and to realize storage outsourcing over the Internet. This paper has presented the implementation of a caching-based network storage system, NetShark, and has proposed a caching approach, a data-mining-based prefetching approach. A simulator has been developed to evaluate the proposed caching approach. Simulation results have shown that the proposed caching approach outperforms other popularly used caching approaches.

Future work is needed in the following areas. First, we plan to evaluate the proposed caching approach using real-world applications such as the knowledge management system mentioned in §2. Second, as NetShark is used continuously, data in Log-DB accumulate continuously. Hence, intra/intersession patterns learned from offline data mining need to be refreshed accordingly. Refreshing patterns too often could not only inflict unbearable costs, but also often result in repetitive patterns identical with previous mining, while refreshing patterns too seldom may result in the missing of critical patterns (Ganti et al. 2001). We plan to apply and extend the research in Fang and Sheng (2000) to design an efficient pattern-refreshing algorithm. Third, we plan to incorporate structural relationships between storage objects (e.g., storage objects in the same directory) into the proposed caching approach. Finally, an analytical model based on queuing theory needs to be developed to analyze the performance of different caching approaches mathematically.

References

Agrawal, R., R. Srikant. 1995. Mining sequential patterns. *Proc. 1995 Internat. Conf. Data Engrg.* Taipei, Taiwan, 3–14.

Barish, G., K. Obraczka. 2000. World Wide Web caching: Trends and techniques. *IEEE Comm. Magazine* **38** 178–185.

Bertsekas, D., R. G. Gallagher. 1995. *Data Networks*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ.

Breslau, L., P. Cao, L. Fan, G. Phillips, S. Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. *Proc. 1999 IEEE INFOCOMM*, New York, 126–134.

Cao, P., S. Irani. 1997. Cost-aware WWW proxy caching algorithm. *Proc. 1997 USENIX Sympos. Internet Tech. Systems*, Monterey, CA, 193–206.

Cao, P., C. Liu. 1998. Maintaining strong cache consistency in the World Wide Web. *IEEE Trans. Comput.* **47** 445–457.

Chang, C., M. Chen. 2002. Web cache replacement by integrating caching and prefetching. *Proc. ACM Internat. Conf. Inform. Knowledge Management*, McLean, VA, 632–634.

Chinen, K., S. Yamaguchi. 1997. An interactive prefetching proxy server for improvement of WWW latency. *Proc. INET'97*. Kuala Lumpur, Malaysia.

Cooley, R., B. Mobasher, J. Srivastava. 1999. Data preparation for mining World Wide Web browsing patterns. *Knowledge Inform. Systems* **1** 1–27.

Crovella, M., P. Barford. 1998. The network effects of prefetching. *Proc. 1998 IEEE INFOCOM*, San Francisco, CA, 1232–1240.

Davison, B. D. 2001. NCS: Network and cache simulator—An introduction. Technical report DCS-TR-444, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Davison, B. D. 2002. Predicting Web actions from html content. *Proc. 2002 ACM Hypertext Conf.* College Park, MD, 159–168.

Dias, G. V., G. Cope, R. Wijayaratne. 1996. A smart Internet caching system. *Proc. INET'96*. Montreal, Canada.

Fan, L., Q. Jacobson, P. Cao, W. Lin. 1999. Web prefetching between low-bandwidth clients and proxies: Potential and performance. *Proc. 1999 Joint Internat. Conf. Measurement Model. Comput. Systems*. Atlanta, GA.

Fang, X., O. Sheng. 2000. A monitoring algorithm for incremental association rule mining. *Proc. 10th Workshop Inform. Tech. Systems*. Brisbane, Australia.

Feldmann, A., R. Caceres, F. Douglis, G. Glass, M. Rabinovich. 1999. Performance of Web proxy caching in heterogeneous bandwidth environments. *Proc. 1999 IEEE INFOCOMM*. New York, 107–116.

Ganti, V., J. Gehrke, R. Ramakrishnan. 2001. DEMON: Mining and monitoring evolving data. *IEEE Trans. Knowledge Data Engrg.* **13** 50–63.

Gibson, G., R. Meter. 2000. Network attached storage architecture. *Comm. ACM* **43** 37–45.

Gwertzman, J., M. Seltzer. 1995. The case for geographically push-caching. *Proc. 1995 Workshop Hot Oper. Systems*. Orcas Island, WA, 51–55.

Horng, Y., W. Lin, H. Mei. 1998. Hybrid prefetching for www proxy servers. *Proc. 1998 IEEE Internat. Conf. Parallel Distributed Systems*. Tainan, Taiwan, 541–548.

Jin, S., A. Bestavros. 2000. Popularity-aware greedy dual-size Web proxy caching algorithms. *Proc. ICDCS'2000*. Taipei, Taiwan, 254–261.

Kim, S., J. Kim, J. Hong. 2000. A statistical, batch, proxy-side Web prefetching scheme for efficient Internet bandwidth usage. *Proc. 2000 Networld + Interop Engrg. Conf.* Las Vegas, NV.

Kroeger, T. M., D. D. E. Long, J. C. Mogul. 1997. Exploring the bounds of Web latency reduction from caching and prefetching. *Proc. USENIX Sympos. Internet Techn. Systems*. Monterey, CA, 13–22.

Law, A. M., W. D. Kelton. 2000. *Simulation Modeling and Analysis*, 3rd ed. McGraw-Hill, New York.

Lee, E. K., C. A. Thekkath. 1996. Petal: Distributed virtual disks. *Proc. 8th ACM Internat. Conf. Architectural Support Programming Languages Oper. Systems*. Cambridge, MA, 84–92.

Liu Sheng, O. R. 1992. Analysis of optimal file migration policies in distributed computer systems. *Management Sci.* **38** 459–482.

Lou, W., H. Lu. 2002. Efficient prediction of Web access on a proxy server. *Proc. ACM Internat. Conf. Inform. Knowledge Management*. McLean, VA.

Markatos, E., C. E. Chronaki. 1998. A top-10 approach to prefetching on the Web. *Proc. INET'98*. Geneva, Switzerland.

Padmanabhan, V. N., J. C. Mogul. 1996. Using predictive prefetching to improve World Wide Web latency. *Proc. SIGCOMM'96 Conf.* Stanford, CA.

Paxson, V., S. Floyd. 1995. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Trans. Networking* **3** 226–244.

Steinke, S. 2001. Network delay and signal propagation. *Network Magazine*. http://www.networkmagazine.com

Tanenbaum, A. S., A. S. Woodhull. 1997. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ.

Thekkath, C. A., T. Mann, E. K. Lee. 1997. Frangipani: A scalable distributed file system. *Proc. 16th ACM Sympos. Oper. Systems Principles*. Saint-Malo, France, 224–237.

Yin, J., L. Alvisi, M. Dahlin, A. Iyengar. 2002. Engineering Web caching consistency. *ACM Trans. Internet Tech.* **2** 224–259.