Improving the Performance of a Distributed Computing System through Inconsistent Caches

Arbee L.P. Chen[*], Kuo-Fang Chieng[*], Tony C.T. Kuo[*], Eddy J.D. Lee[*], Shu-Chin Su[†]

Abstract

In a Server/Client based distributed computing system, data can be cached at the clients to increase read efficiency and system scalability. When the consistency of the caches can be relaxed and the inconsistency controlled, the performance of the distributed system can be furture improved. In this paper, we elaborate on this concept and present a prototype system which we have constructed to show the viability of the concept.

1 Introduction

In a distributed computing system based on a server/client architecture, data can be replicated among servers for increasing data availability and reliability. Data can also be cached at clients for increasing read efficiency and system scalability. Replicas among servers are usually kept consistent at every point of time while the consistency of caches at the clients could be relaxed by allowing some degree of value divergence from the data in the servers.

By this consistency relaxation, applications may tolerate some inconsistencies for several advantages. The advantages include allowing updates to be propagated from the servers to the clients more efficiently (e.g., when the system is lightly loaded, or by batching together updates), reducing the need to globally lock replicated data for updates so that data availability can be increased, and possibly eliminating the need to synchronize read-only transactions with update transactions so that the query response time can be reduced. Various consistency requirements encountered in various applications can be accommodated by a flexible inconsistency specification mechanism.

In this context, the notion of quasi-copies which was introduced in [1] becomes relevant. Quasi-copies are caches at the clients whose values may be somewhat diverged from those of the replicas but are guaranteed to meet a certain predicate (named *quasi-caching* predicate). Several types of inconsistency predicate may be defined: delay conditions (e.g., the cache must not be more than one hour old than the replicas), version conditions (e.g., the cache should not lag behind more than two versions of the replicas), and arithmetic conditions (e.g., the cache's value must not be off by more than 10% of that of the replicas).

In this paper, we elaborate on the concept by considering different types of read and a mechanism to control the cache inconsistency. There are presented in section 2. Section 3 presents a prototype system. We have built for the concept, whose *quasi-caching* predicates are based on version conditions. We conclude this work in section 4.

2 Allowing Cache Inconsistencies

2.1 Taxonomy of read operations

Three types of read operations are provided in the system for fulfilling different user requirements of the data currency and consistency. That is, a *strong-read* reads current and consistent data, an *urgent-read* always reads current data which may sometimes be inconsistent, and a *quasi-read* reads consistent data which may be out-of-date. They are detailed in the following.

- **quasi-read:** Clients may tolerate the deviation of the cached data within some range (named *quasi-range*). A quasi-read can be issued in this case to improve the read performance. In order to achieve the data consistency for all reads in a process(i.e., all data read are in the same version), the quasi-range is checked only at the opening of a file. That is, even the quasi-range is exceeded when the process is active, the same version of data is used for the whole process. We call this a *weak consistency* of read operations.

[*]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, ROC.

[†]Computer & Communication Research Laboratories, Industrial Technology Research Institute, Hsinchu, Taiwan, ROC.

- **urgent-read:** In some situations, to always read the current data without being blocked is required. In this case, there is no need to synchronize this read with other operations on the file in the server. Since each read in the process is required to read up-to-date data, the cache in the client may need to be refreshed several times before the file is closed. That is, the data read may come from different versions and therefore can possibly be inconsistent.

- **strong-read:** This type of reads requires reading the up-to-date data and also does not allow other transactions to modify the data when the process is active. The data read for the whole process are up-to-date and consistent.

2.2 Controlling cache inconsistency

The implementation of these three types of read operations and the control of the data inconsistency are accomplished by a mechanism with four flags: a *CALLBACK* flag to inform the clients that the data in the servers have been changed; a *QUASI-CACHE* flag to inform the clients that the predicates governing the consistency constraints between the replicas and the caches have been violated; a *WRITELOCK* flag to lock the data for updates at the servers such that the data cannot be read or modified by other transactions; and a *READLOCK* flag to lock the data such that the data can only be read by other transactions.

These four flags are combined as a four-flag vector (C,Q,W,R), C for CALLBACK, Q for QUASI-CACHE, W for WRITELOCK, and R for READLOCK. The vector can be stored at a server or at the clients. When it is stored at the clients, the clients can reference the vector locally to know the current status for read/write actions. However, to maintain the vectors at all clients poses a difficult consistency control problem.

On the other hand, if the vector is stored at a server, the maintenance of the vector becomes easier. However, the client has to access the server to check the status of the vector for its read/write actions. This increases the burden of the server and decreases the scalability of the system.

We therefore divide the vector (C,Q,W,R) into two parts (C,Q) and (W,R). (C,Q) associates with a cache in the client, and is located at the client. (W,R) associates with a file stored in the server, and is located at the server. The meanings of the four flags are listed in the following before we discuss different cases of their combinations:

- C is set to 1 when the cache in the client is out-of-date. That is, the corresponding replicas have been modified.

- Q is set to 1 when the predicate governing the consistency constraint between the replicas and the cache has been violated.

- W is set to 1 when the replicas have been locked for modification.

- R is set to 1 when the replicas have been locked for read.

The semantics of the possible cases represented by the flags are discussed as follows.

- (C,Q)=(0,0) means that the data in the cache is up-to-date, and the quasi-range is (of course) not exceeded. Quasi-read and urgent-read can be performed directly at the cache while strong-read and write need to request a lock on the replica at the server. The performance improvement for quasi-read and urgent-read can easily be seen.

  (W,R)=(0,0) permits either strong-read or write issued from a client.

  (W,R)=(0,1) disallows write operations until R is reset.

  (W,R)=(1,0) disallows strong-read and write operations.

  (W,R)=(1,1) represents an impossible situation where a data item is both read-locked and write-locked.

- (C,Q)=(0,1) represents an impossible situation because C=0 means the cache is up-to-date, which implies the quasi-range cannot be exceeded.

- (C,Q)=(1,0) means that the data in the cache is out-of-date but the quasi-range is not exceeded. In this case, quasi-read is permitted while urgent-read will require a request to refresh the cache first.

- (C,Q)=(1,1) means that the data in the cache is out-of-date and the quasi-range is exceeded. Any operation referencing the cache should request the server to refresh the cache first.

This mechanism checks the flags to know the status of the caches for various data operations. It also issues actions to refresh the cache for an operation which cannot be satisfied with the current data.

3 A Prototype System

3.1 System overview

This prototype works on BSD UNIX operating environment with one computer serving as a server and others as clients connected by computer networks. The communication mechanism uses the Berkeley socket-based IPC (InterProcess Communication) [2]. The system architecture of the prototype is shown in Figure 1.

The operations between a client and a server are shown in Figure 2. There are three processes in this figure — ServerP, ClientP, and a user process. The user program is linked with cm_library (cache management library) to form the user process. ServerP and ClientP are always active at the server and client, respectively; however, the user process is active only when the user program is executed.

ServerP receives the requests from the user process. The requests could be *getnewcache, flush, set_readlock, set_writelock, release_readlock*, and *change_quasirange* (to be explained in section 3.3.2). ClientP receives the messages from the ServerP. The messages could be *set_callback* (to set the CALLBACK flag) and *set_quasicache* (to set the QUASI-CACHE flag).

The cm_library is built on top of the C-library to provide the cache management functions. The provided functions are *cm_open, cm_close, cm_read, cm_write*, and *to_change_quasirange* (will be discussed in detail in section 3.3.1).

Cachetable, cachelist1, and cachelist2 are three tables used to maintain the cache consistency. Cachetable is referenced by ClientP and the user process while cachelist1 and cachelist2 are referenced by ServerP.

3.2 Data structure

The tables — *cachetable, cachelist1 and cachelist2* — are extensions of the four-flag vector described in section 2.1. *Cachetable* and *cachelist1* control the data inconsistency between the server and clients while *cachetable2* controls the quasi-caching predicates.

cachetable : this table exists at each client. Each tuple in the table corresponds to one file cached in this client. This table contains five attributes. Its structure is shown as follows :

filename	CALLBACK	QUASI-CACHE	mode	active

- **filename :** if a filename exists in this table, this file is cached at the client.
- **CALLBACK :** this flag is set when the corresponding cache is out-of-date. It is referenced by cm_open operation whose open mode (will be discussed later) can be urgent-read, strong-read, or read-write. It can also be referenced by cm_read operation with open mode urgent-read.
- **QUASI-CACHE :** this flag is set when the corresponding predicate is violated. It is referenced by cm_read operation with open mode quasi-read.
- **mode :** when a file is opened, the open mode is set and saved in this flag (i.e., the open operation's format is cm_open(filename, open mode)). The open mode can be quasi-read, urgent-read, strong-read, or read-write. If a file whose open mode is quasi-read (urgent-read, strong-read), then the cm_read operation issued on this file is quasi-read (urgent-read, strong-read). This flag is referenced by read, write and close operations.
- **active :** when the file has been opened, this flag is set. We allow a user to open many files at the same time, but disallow a file to be opened by multiple users at the same time to prevent conflicting open modes. When a user wants to open a file, the active flag will be checked. If the flag is set then this operation is rejected; otherwise, the file is opened. Another function of this flag is that it prevents this file from being flushed out during cache replacement. When we close the file, this flag is reset.

cachelist1 : The structure of cachelist1 table is shown as follows :

filename	READLOCK	WRITELOCK	version

If a file exists at the server, there should be a tuple in this table. READLOCK and WRITELOCK flags are set by cm_open operation with the open mode strong-read and read-write, respectively. Version is used to record the number of updates on the file. It is increased by 1 each time when the corresponding file is updated.

cachelist2 : the cachelist2 table controls the quasi-caching predicate. We show the structure of it as follows :

filename	client	allow	version

- **filename :** the name of a file cached at the client.
- **client :** the address of a client which caches the file. This attribute is used when the server wants to communicate with the client.
- **allow :** it denotes the allowable quasi-range.
- **version :** the version number when the file was cached at the client. When a file is updated at the server, the version number in cachelist1 table is increased. Therefore, when

  $$(cachelist1.version - cachelist2.version) > cachelist2.allow$$

  the predicate is violated.

3.3 Operations

3.3.1 Clients

The following functions are used by the clients:

- **cm_open(filename, open mode) :** The open mode could be read-write, strong-read, urgent-read, and quasi-read.

  read-write : it sends set-writelock message to the server; if the server returns a failure message (e.g., the WRITELOCK or READLOCK flag has been set) then this function returns a failure message to the user. Otherwise, it checks CALLBACK flag to see if the cache can be used to perform the update. If CALLBACK is set then it sends getnewcache (to be explained later) to the server.

  strong-read : it sends set-readlock message to the server; if the server returns a failure message (e.g.,the WRITELOCK flag has been set) then this function returns a failure message to the user. Otherwise, it checks CALLBACK flag to see if the cache can be used for the read. If CALLBACK flag is set then it sends getnewcache message to the server.

  urgent-read : it checks CALLBACK flag first; if this flag is set then it sends getnewcache message to the server; CALLBACK flag must be checked for each of the reads in the process.

  quasi-read : it checks QUASI-CACHE flag first; if this flag is set then it sends getnewcache message to the server. Notice that in order to maintain a weak consistency, QUASI-CACHE flag need not be checked for the following read operations.

- **cm_close(filename) :** it checks the open mode first; if it is urgent-read or quasi-read then it need not inform the server. Otherwise,

  read-write : it sends flush message to the server and transfers the modified data from the client to the server.

  strong-read : it sends release-readlock message to the server.

- **cm_read(filename) :** it checks the open mode first; if it is urgent-read then checks the CALLBACK flag; otherwise, reads the data directly. If the CALLBACK flag is set then it sends the getnewcache message to the server. Other types of read read the cache directly.

- **cm_write(filename) :** it checks whether the file is opened for read-write mode; if it isn't then this request is rejected else the data are written directly.

- **to_change_quasirange(filename, client-address, quasi-range) :** we allow users to change the quasirange specified in cachelist2. That is, after the execution of to_change_quasirange, cachelist2.allow will have a new value "quasi-range".

3.3.2 Servers

The following functions support the requests from the client :

- **set_writelock(filename) :** the server checks WRITELOCK and READLOCK flags; if one of the two flags is set then it returns a failure message. If the two flags are both reset, the server determines which set_writelock request to accept when there are more than one request arriving at the same time. The server returns a success message to the client and sets the WRITELOCK flag accordingly.

- **set_readlock(filename) :** the server checks the WRITELOCK flag; if it is set then it returns a failure message. If the WRITELOCK flag is not set then the READLOCK flag is increased by one. The READLOCK flag is implemented as a counter to represent the number of the transactions currently reading the data. The server then returns a success message to the requested client.

- **release_readlock(filename) :** it decreases the number of the transactions currently accessing the data (i.e., decreasing the value of the READLOCK flag). After that, if none of the transactions are reading the data then the value of READLOCK is zero.
- **getnewcache(filename, client address) :** it transfers data from the server to the client to refresh the cache. The CALLBACK and QUASICACHE flags at the client are reset accordingly. If the file is newly cached by the client, then a new tuple is inserted to the cachelist2 table; otherwise, the value of the version attribute in the corresponding tuple of cachelist2 table is updated by the value of the cached file's new version.
- **flush(filename) :** it updates the file according to the modified one received from the client, sets the CALLBACK flags for all the other clients which cache the data, tests quasi-range for all the other clients which cache the data, and sets the QUASICACHE flag at the clients if needed. Moreover, the WRITELOCK flag is reset.
- **change_quasirange(filename, client address, quasi-range):** When the server receives the to_change_quasirange request from the client, it updates the allow attribute value in the corresponding tuple of cachelist2 table. In addition, it must check whether the quasi-range is violated due to this change.

4 Future Work

We have described a mechanism to control inconsistent caches and an implementation of the mechanism. The following issues are our future work items on this system :

- **provide a robust system :**
  We have discussed the advantages of the inconsistent-caches in a distributed system. To gain these advantages, however, the system must be able to control the cache inconsistency in the presence of failures. There are many algorithms proposed to preserve database consistency when failures occur [3, 4, 5, 6, 7, 11, 13, 14]. We will investigate if they can be adapted to our environment.
- **provide a flexible quasi-range :** Quasi-range can be defined based on delay conditions, version conditions, or arithmetic conditions. We have used version conditions as the quasi-caching predicate. Other conditions can also be considered. Furthermore, we should provide users a flexibility to choose from different conditions for their own application requirements.
- **performance analysis :**
  Currently, we are doing system performance analysis based on the prototype system and a simulation model. The effect of various cache units (lock units), cache replacement strategies, percentages of quasi-read, quasi-ranges, and other issues are under investigation.

References

[1] Alonso, R., Barbara, D. and Garcia-Molina, H., "Data Caching Issues in an Information Retrieval System", *ACM Trans. Database Syst.,* Sept. 1990.

[2] W. Richard Stevens, "UNIX Network programming", *Prentice-Hall,* 1990, pp. 258 — 341.

[3] Blaustein, B. T. and Kaufman, C. W., "Updating Replicated Data During Communication Failures", Proceedings of the 11th International Conference on Very Large Data Bases, 1985.

[4] Davidson, S. B., "Optimism and Consistency in Partitioned Distributed Database Systems", *ACM Trans. Database Syst.,* Sept. 1984.

[5] Garcia-Molina, H. and Barbara, D., "How to Assign Votes in a Distributed System", *Journal of ACM,* Oct. 1985.

[6] Jajodia, S. and Mutchler, D., "Dynamic Voting," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1987.

[7] Jajodia, S. and Mutchler, D., "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database", *ACM Trans. Database Syst.,* Jun. 1990.

[8] Paris, J. F., "Voting with Witnesses: A Consistency Scheme for Replicated Files", Proceedings of the IEEE International Conference on Distributed Computing, 1986.

[9] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access", *IEEE Computer,* May 1990.

[10] Satyanarayanan, M., et al., "Coda: A Highly Available File System for a Distributed Workstation Environment", *IEEE Trans. on Computers,* Apr. 1990.

[11] Wright, D. D., "On Merging Partitioned Databases", *ACM SIGMOD Rec.,* 1983.

[12] Levy, E. and Silberschatz, A., "Distributed File Systems: Concepts and Examples", *ACM Computing Surveys,* 1990.

[13] Bhargava, B. and Lian, S., "Typed Token Approach for Database Processing during Network Partitioning", Tech. Rep., Computer Science Department, Purdue University, 1991.

[14] Susan B, Davidson. and Hector Garcia-Molina. and Dale Skeen. "Consistency in Partitioned Networks", *ACM Computing Surveys,* 1985

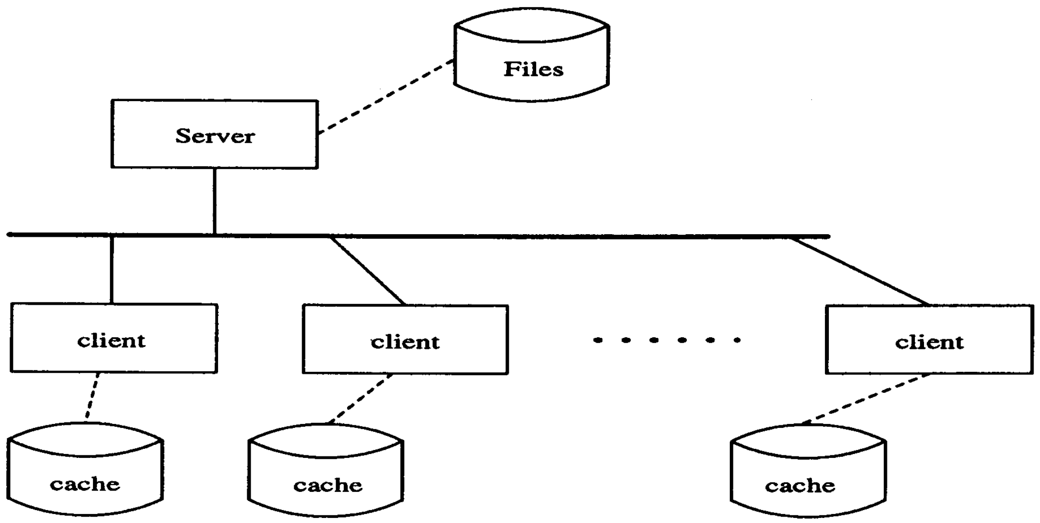

Figure 1: The server/client system architecture.

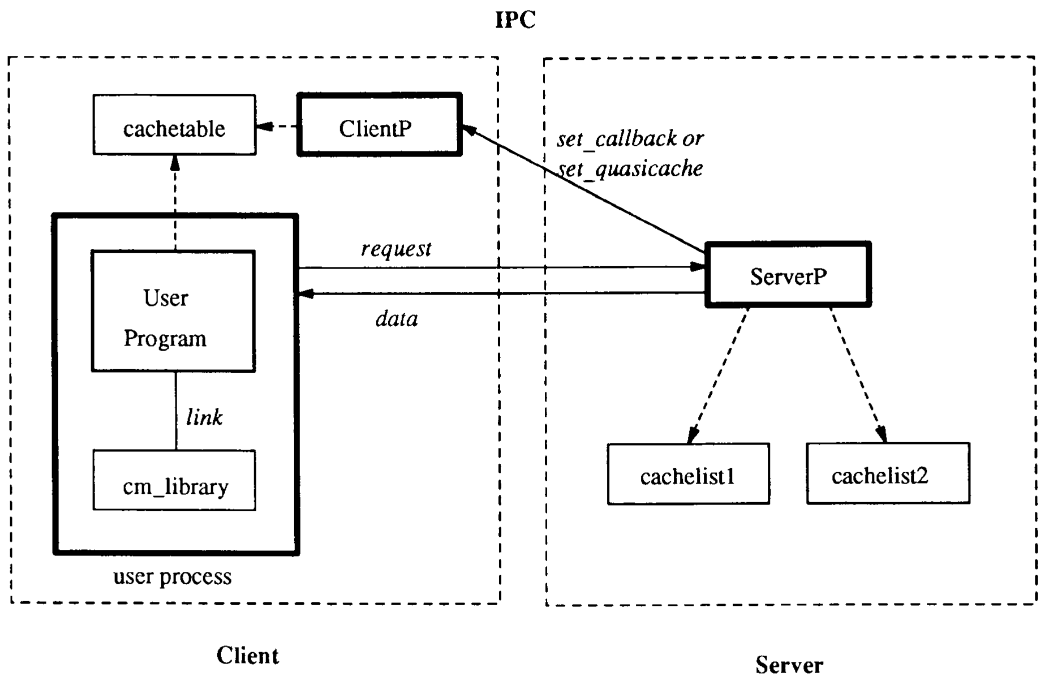

Figure 2: The operations beteeen server and client.