

Mixed Consistency Model: Meeting Data Sharing Needs of Heterogeneous Users*

Zhiyuan Zhan
College of Computing
Georgia Institute of Technology
USA
zzhan@cc.gatech.edu

Mustaque Ahamad
College of Computing
Georgia Institute of Technology
USA
mustaq@cc.gatech.edu

Michel Raynal
IRISA
University of Rennes
France
raynal@irisa.fr

Abstract

Heterogeneous users usually have different requirements as far as consistency of shared data is concerned. This paper proposes and investigates a mixed consistency model to meet this heterogeneity challenge in large scale distributed systems that support shared objects. This model allows combining strong (Sequential) consistency and weak (Causal) consistency. The paper defines the model, motivates it and proposes a protocol implementing it.

1 Introduction

Context: heterogeneous users The rapid proliferation of mobile devices and wireless networks has made it possible to access and share information across different platforms with varying degree of computing power and network resources. For example, various cars can share traffic information through on-board communication devices. This scenario requires that the information be shared, disseminated and updated at a potentially large number of heterogeneous users. There are two sources of heterogeneity in such an environment: user needs and system resources. For instance, some users may not care about each new update, while others do. Also, some users can use relatively powerful systems to access the information, while others may only be able to use wireless handheld device to do so. The lack of computing power and/or network resources will prevent them from employing expensive protocols to ensure high quality of shared information at all time. Both sources suggest that the information sharing system should take heterogeneity as a primary concern if it is to meet the needs of varied users in a wide area network.

Introducing a mixed consistency model We assume that shared information is encapsulated in distributed objects.

Object replication is a common technique to increase the scalability and robustness of such systems, which also introduces the consistency problem. Many consistency models have been proposed in recent years. The tradeoff between performance and the level of consistency presents a dilemma. For example, strong consistency such as sequential consistency ensures a unique order of operations across different replicas, but its implementation is costly and does not scale very well; the ordering guarantees provided by different weak consistency models may result in conflicting views of shared critical information but they are relatively easy to implement with good performance. Therefore, only supporting a single level of consistency, which is commonly done in existing systems, either does not scale well or is insufficient for meeting user requirements. In order to maintain consistency in a heterogeneous environment, we are exploring an approach of employing *mixed consistency* for objects which addresses both user needs and system resource constraints.

- For a particular piece of information, users can choose the consistency level they desire based on the current resource level they have. Resourceful users can access information with stronger consistency, while others may be forced to access information with weaker consistency guarantees due to lack of resources.
- Strong consistency for critical information is necessary but has a relatively high maintenance cost. In many applications, weaker consistency provides a relatively cheap yet scalable way for a large number of users to share information that has less stringent correctness requirements.

With a mixed consistency model, applications can make use of both high (strong) and low (weak) levels of consistency to meet their needs based on the availability of resources at various nodes. However, mixing leads to a new problem: how to meaningfully define consistency when resource poor nodes want to access an object replica in a

*This work was supported in part by NSF grant ITR-0121643.

weak mode whereas others want to maintain strong consistency for their replicas of this same object. Operations observed by strong replicas should appear to be executed on a single copy serially, but nodes with weak replicas do not have the resources necessary to ensure such strong ordering. Since update operations produce new values for replicas and strong replicas must observe new values in a consistent order, we identify constraints that prohibit values of updates done at weak replicas to be disseminated to strong replicas in our model. At the same time, a weak replica can observe values of updates done at strong replicas as well as at other weak replicas. Such a model can meet the needs of many applications. In order to bound the difference between weak and strong replicas, weak replicas can periodically update themselves with values written by nodes that maintain strong replicas. The dissemination flow constraints allow us to precisely characterize the mixed consistency model, and they can easily be incorporated into a protocol that implements this model in a heterogeneous environment.

Although the mixed consistency model we propose is not limited to any particular consistency models, sequential consistency (SC) and causal consistency (CC) are the strong and weak consistency models we consider in this paper. They both have been extensively studied over the past years and many protocols have been proposed to implement them. We use consistency tags associated with replicas and processes to implement the constraints needed in a mixed consistency model.

- Each object replica has a consistency tag of either SC or CC. SC replicas encapsulate critical information updates that should be maintained under sequential consistency. CC replicas encapsulate other information which is only maintained under causal consistency.
- Each process has a consistency tag of either SC or CC. SC processes are resourceful processes that can afford to use more expensive protocols to access different types of information, while CC processes are resource limited processes that can only afford cheap protocols for the same information.

Paper contributions A careful study of mixed consistency leads to a rich set of possibilities. In this paper, we formally define a mixed consistency model to generalize many application scenarios, where each object can have both SC and CC replicas at the same time. We also propose a protocol to ensure both SC and CC consistency requirements. The main contributions of this paper are:

- Based on different application requirements, we propose a unified mixed consistency model allowing both SC and CC replicas to coexist at the same time.

- We design a mixed consistency protocol to ensure both SC and CC at the same time and show that it is correct and can offer performance versus consistency tradeoff in a heterogeneous environment.

The rest of the paper is organized as follows: Section 2 introduces the system model. Section 3 formally defines the mixed consistency model. Section 4 provides a mixed consistency protocol implementing the model and outlines the correctness proof. Section 5 presents performance related analysis of our protocol. Section 6 discusses the related work. Section 7 finally summarizes the paper and lists future work.

2 System Model

We consider a replicated object system composed of distributed processes, each of which having a local memory where copies of objects are stored. We assume that one process can only have at most one replica of a particular object in its local memory. Replicas can be removed from the local memory when they are no longer needed or there is a memory shortage. New object replicas can be created as needed. However, we will not discuss the creation and deletion of replicas in this paper. We will focus our discussion on a system where the locations of object replicas are already determined. In such a system, a process has access to its local copies of objects. Processes can also communicate with each other through reliable pair wise inter-process communication channels.

Let P denote the process set, where each $p_i \in P$ represents an individual process (assuming $1 \leq i \leq N$). We define that P consists of two disjoint subsets: P_{SC} and P_{CC} . Processes in P_{SC} (P_{CC}) are called SC (CC) processes.

Let O denote the object replica set, where each $o_{ij} \in O$ stands for a replica of object o_j stored in process p_i 's local memory (assuming $1 \leq j \leq M$). Clearly for a particular j ($1 \leq j \leq M$), all the $o_{xj} \in O$ ($1 \leq x \leq N$) are initialized from the same object o_j . Similarly, we define that O also consists of two disjoint subsets: O_{SC} and O_{CC} . Replicas in O_{SC} (O_{CC}) are called SC (CC) replicas.

3 The Mixed Consistency Model

A process p_i accesses an object o_j by invoking the methods on an object replica o_{ij} in its local memory. We categorize the methods into “read” (r) and “write” (w) operations. The “read” method does not change the state of the object copy, while the “write” method does. We adopt the notation used in paper [2] to define the consistency model of our system, i.e., operation $r_i(o_j)v$ stands for p_i reading object o_j and returning its current value (or state) v , and operation $w_i(o_j)v$ stands for p_i writing to object o_j a new value v . We

also use $r(o_j)v$ and $w(o_j)v$ to denote the operations when who issues the operation is clear or unimportant.

3.1 Access Constraints

Mixed consistency aims to meet both application needs and resource limitations that are common in heterogeneous environments. Thus, when a resource poor node chooses to make a weak replica of an object and updates it, it cannot be expected to have enough resources to update the strong replicas of the object at other nodes in a manner that is required by the SC model. On the other hand, SC requires that updates observed by strong replicas must be ordered in a uniform manner. These conflicting goals must be reconciled in the mixed consistency model such that strong replicas attain the desired type of ordering for operations observed by them, while allowing weak replicas to achieve the sharing and efficiency required by them. We achieve these goals by developing a set of access constraints that allow weak replicas to access the values of updates done at strong as well as weak replicas. Strong replicas can only observe values of those updates that are done at the nodes that maintain strong replicas. This is reasonable because (1) nodes with strong replicas have the necessary resources to enforce the strong ordering, and (2) updates done to weak replicas at resource poor nodes, which do not want to incur the cost of strongly ordering the updates, are not observed by the strong replicas to prevent violations of ordering requirements. This is also consistent with the requirements of applications where nodes with weak replicas either do not update objects that have other strong replicas or values of their updates are only shared with other nodes that have weak replicas. The constraints that define which updates can be observed by which type of replicas can be captured by the following two rules:

- **Rule 1:** A SC process can read and write a SC replica and can only write a CC replica;
- **Rule 2:** A CC process can read and write a CC replica and can only read a SC replica.

Table 1 defines both rules in the mixed consistency model. Each row in Table 1 gives the legal access rights of the particular process group to different object groups.

	O_{SC}	O_{CC}
P_{SC}	RW	W
P_{CC}	R	RW

Table 1. Access Constraints Table

3.2 Well-formed Serialization and History

Well-formed Serialization: Let T be an arbitrary set of operations. We say T is *well-formed* if and only if no operation in T violates Table 1. We call S a *serialization* of

T if S is a linear sequence containing exactly the operations of T such that each read operation from an object returns the value written by the most recent write to that object. From now on, when we say a *serialization*, it means a *serialization* of a *well-formed* operation set by default. In other words, a *serialization* always respects **Rule 1** and **Rule 2**.

For example, let's consider a system setting of two processes (p_1, p_2) and three objects (o_1, o_2 and o_3). Both p_1 and p_2 have a local copy of all three objects. p_1, o_{11}, o_{22} are marked as SC, while $p_2, o_{21}, o_{12}, o_{13}, o_{23}$ are marked as CC. We have three operation sets defined as:

$$T_1 = \{w_1(o_3)1, r_1(o_1)1, r_1(o_2)1, r_1(o_3)1, w_2(o_1)2, w_2(o_2)2, w_2(o_3)2, r_2(o_2)2\}$$

$$T_2 = \{w_1(o_1)1, w_1(o_2)1, w_2(o_3)2, r_2(o_1)1, r_2(o_1)2, r_2(o_3)2\}$$

$$T_3 = \{w_1(o_1)1, r_2(o_1)1, w_1(o_2)2, r_2(o_3)2, w_2(o_3)2, r_1(o_1)1\}$$

We can see that T_1 is not *well-formed* because $r_1(o_2)1, r_1(o_3)1, w_2(o_2)2$ violates Table 1. T_2 is *well-formed* but it does not have a *serialization* because $r_2(o_1)2$ returns a value 2 that has never been written to o_1 (assuming each object has an initial value 0). T_3 is *well-formed* and

$$S = \{w_2(o_3)2, w_1(o_1)1, w_1(o_2)2, r_2(o_1)1, r_2(o_3)2, r_1(o_1)1\}$$

can be one *serialization* of T_3 .

Projection: Let T be an arbitrary set of operations. We define a projection of T on an arbitrary process group S , denoted as T_S , to be a subset of T , which only contains the operations performed by processes in S . Similarly, we also define a projection of T on an arbitrary object replica group J , denoted as T^J , to be a subset of T , which only contains the operations performed to replicas in J . It is easy to see that $(T_S)^J = (T^J)_S = T_S^J$.

Causal Order: Let A be a complete set of all operations. We define the *program order* \rightarrow on A to be a complete set of $\langle op_1, op_2 \rangle$ such that both op_1 and op_2 are performed by the same process p and op_1 precedes op_2 according to p 's local clock. In this case, we write $op_1 \rightarrow op_2$.

Without loss of generality, we assume that all writes to the same object are uniquely valued. The *writes-into order* \mapsto on A is defined as such that $op_1 \mapsto op_2$ holds if and only if there are o_j and v such that $op_1 = w(o_j)v$ and $op_2 = r(o_j)v$.

A *causal order* \Rightarrow induced by both \rightarrow and \mapsto in our model is a partial order that is the transitive closure of the program order and the writes-into order defined on A . To be specific: $op_1 \Rightarrow op_2$ holds if and only if one of the following cases holds:

- $op_1 \rightarrow op_2$ (program order); or

p_1 (SC):	w(x)1	r(x)1	w(y)1	r(x)2		
p_2 (SC):	w(x)2		r(x)2	w(y)2	r(x)2	
p_3 (CC):	w(y)3	r(y)4		r(x)1	r(x)2	r(y)2
p_4 (CC):	w(y)4	r(y)3	r(x)1		r(x)2	r(y)2

Figure 1. Mixed Consistency Example with $O_{SC} = \{x_1, x_2, x_3, x_4\}$ and $O_{CC} = \{y_1, y_2, y_3, y_4\}$

- $op_1 \mapsto op_2$ (writes-into order); or
- there is another op_3 such that $op_1 \Rightarrow op_3 \Rightarrow op_2$

History: We define the *global history* (or *history*) of an operation set A , denoted as H , to be a collection of A 's operations and the *program order* among those operations, i.e. $H = \langle A, \rightarrow \rangle$.

3.3 Mixed Consistency

We say that a history H is *mixed-consistent* if it satisfies all the following requirements:

1. **SC requirement:** there exists a serialization of A^{SC} such that it respects the program order;
2. **CC requirement:** for each CC process p_i , there exists a serialization of $A_{\{p_i\}} \cup W$ such that it respects the causal order (W denotes the set of all writes).

In order to illustrate the mixed consistency model, let's consider one example. Figure 1 gives the system setting and the execution history H (operations in the same column denote concurrent operations), where the notion x_i (or y_i) denotes the replica of object x (or y) that process p_i has.

The history H in Figure 1 is mixed-consistent because it meets all the requirements:

1. **SC requirement:** we know that

$$A^{SC} = \{w_1(x)1, r_1(x)1, r_1(x)2, w_2(x)2, r_2(x)2, r_2(x)2, r_3(x)1, r_3(x)2, r_4(x)1, r_4(x)2\}$$

Clearly A^{SC} is well-formed. And

$$S = \{w_1(x)1, r_1(x)1, r_3(x)1, r_4(x)1, w_2(x)2, r_1(x)2, r_2(x)2, r_2(x)2, r_3(x)2, r_4(x)2\}$$

is a serialization that respects the program order of all p_i ($1 \leq i \leq 4$).

2. **CC requirement:** we have

$$\begin{aligned} A_{\{p_3\}} &= \{w_3(y)3, r_3(y)4, r_3(x)1, r_3(x)2, r_3(y)2\} \\ A_{\{p_4\}} &= \{w_4(y)4, r_4(y)3, r_4(x)1, r_4(x)2, r_4(y)2\} \\ W &= \{w_1(x)1, w_2(x)2, w_1(y)1, w_2(y)2, w_3(y)3, w_4(y)4\} \end{aligned}$$

For process p_3 :

$$S_3 = \{w_3(y)3, w_4(y)4, r_3(y)4, w_1(x)1, r_3(x)1, w_2(x)2, r_3(x)2, w_1(y)1, w_2(y)2, r_3(y)2\}$$

is the serialization that respects the causal order.

For process p_4 :

$$S_4 = \{w_4(y)4, w_3(y)3, r_4(y)3, w_1(x)1, r_4(x)1, w_2(x)2, r_4(x)2, w_1(y)1, w_2(y)2, r_4(y)2\}$$

is the serialization that respects the causal order.

4 A Mixed Consistency Protocol

To design brand new protocols to ensure either SC or CC is not our purpose. Instead, we show in this paper that we can combine existing SC and CC protocols together to meet mixed consistency requirements. We choose two well-studied protocols in the literature to achieve this goal.

SC: Home-based Protocol We choose home-based protocol [6] to ensure SC requirements. A designated home node is associated with every object that has a SC replica to coordinate access to such replicas. A node with SC replica must acquire a token prior to access. The home node keeps track of what nodes can read or write the object's replicas and grants READ/WRITE_TOKEN to the nodes that want to read/write this object. The READ_TOKEN is shared by multiple readers, while the WRITE_TOKEN does not coexist with the READ_TOKEN and can only be issued to one writer at a time. Tokens are issued based on a First-Come-First-Serve order observed by the home node. Currently, the location of the home node is randomly assigned in our protocol, although results in [7, 19, 21] suggest that random assignment may have a negative impact on the performance of home-based protocols. We can employ a "smart" home node assignment when an application profile is available.

CC: Causal Memory Protocol We choose vector clock based causal memory protocol [2] to ensure CC requirements are met. When a write operation is performed, a new value along with the local clock will be disseminated to other replicas. The receiver applies the new value when all the "causally preceding" values have arrived and been applied, which is determined based on the receiver's local vector clock and the clock value that comes with the new object value. In this protocol, the dissemination process can be done in background. Therefore, write operations do not block and return immediately.

INITIALIZATION	
NAME	FUNCTIONALITY
init()	Initialize local data structures such as timestamps and queues.
APPLICATION INTERFACE (invoked by upper layer)	
read(x)	Return the value v of object x.
write(x,v)	Write new value v to object x.
LOCAL FUNCTIONS (invoked by local application interfaces)	
write_miss(x,v)	Request a new WRITE_TOKEN from the home node and complete the local write.
read_miss(x)	Request a new READ_TOKEN from the home node and pull new value from the latest writer if necessary.
SYNCHRONOUS COMMUNICATION PRIMITIVES (invoked by remote nodes, block before return)	
value_request(y)	Return the latest value of y (and associated timestamps, if necessary) to the caller node.
write_token_request(y)	Return the new WRITE_TOKEN to the caller node after revoking all other tokens.
read_token_request(y)	Return the new READ_TOKEN to the caller node after revoking other's WRITE_TOKEN, if any.
write_token_revoke(y)	Delete local WRITE_TOKEN.
read_token_revoke(y)	Delete local READ_TOKEN.
ASYNCHRONOUS COMMUNICATION PRIMITIVES (keep running forever, monitoring queues)	
send_daemon()	Keep running forever; send msgs to selected destinations from output queue oqueue.
receive_daemon()	Keep running forever; receive msgs from the input queue iqueue, and apply new values if necessary.

Figure 2. Categorized interfaces of mixed consistency protocol

4.1 Challenges for Mixed Consistency

We give our mixed consistency protocol in Figure 6. The protocol interfaces are summarized in Figure 2, which will be explained later in the paper to help understand the protocol. Before that, we want to address several problems of integrating the SC and CC protocols together in order to implement the mixed consistency model.

Possible Causal Order Violation: In our model, a process can have both SC and CC replicas. If we simply run the two protocols to maintain replicas based on their type, we could have a potential CC violation. For instance, suppose there are two processes, namely p_1 (SC) and p_2 (CC). Each has a SC copy of o_1 and a CC copy of o_2 . p_1 writes o_2 first and then writes o_1 . p_2 reads o_1 and then o_2 . The home-based protocol can guarantee that p_2 reads what p_1 writes to o_1 . But when p_2 reads o_2 , it might get the “old” value because p_1 ’s new value for o_2 is possibly still on the way because of the background dissemination of the causal memory protocol. We overcome this obstacle by delaying CC reads (setting the replica as NOT READABLE) if a potential CC violation is possible. In Figure 6, function *read_miss()* evaluates the vector clock returned by process p_k and sets any CC object as NOT READABLE if a causal order violation is possible as described above. New CC values will eventually arrive (please refer to Section 4.4) so the delayed CC reads will eventually return with the correct value. In Figure 6, function *receive_daemon()* resets the object as READABLE when it receives a new CC value.

Possible Vector Clock Error: When a SC process writes a SC object, the new value will be disseminated to the CC replicas held by other processes. The receiver needs a correct vector clock in the message to order the update to the CC replica (i.e. when to apply the new value). Therefore, all SC replicas should maintain a “correct” vector clock associated with them, which in turn requires that when a CC pro-

cess writes a CC object, although the value does not propagate to SC replicas (otherwise violates **Rule 1**), the vector clock does. Our design enforces the dissemination of vector clocks as required. In Figure 6, function *send_daemon()* sends the message to all processes (both SC and CC) when the local writer p_i is a CC process. Please note that this does not violate our access constraints because the function *receive_daemon()* at the receiver’s side only applies the value to the local copy when the receiver is a CC process. If the receiver is a SC process, only the vector clock is used to keep the local vector clock correct. The CC value embedded in the message is discarded.

4.2 Protocol Interfaces

The protocol interface is shown in Figure 2, where we categorize the functions into initialization, application interface, local functions, synchronous and asynchronous communication primitives. We use the term “synchronous communication” to refer to blocking, RPC style communication. It is performed through Remote Method Invocation (RMI). We use “asynchronous communication” to refer to non-blocking, send/receive daemon style communication. Our protocol does not place any specific requirements on how to perform asynchronous dissemination. Various multicast or rumor-spreading based techniques can be used as the communication layer support for our protocol. We are exploring an adaptive communication layer under our protocol, featuring better performance and bandwidth utilization in a heterogeneous environment.

In our protocol, *read(x)* and *write(x,v)* are the interfaces exposed to the applications. These two functions first evaluate if the constraints in Table 1 are violated. If yes, an exception will be thrown. For a SC replica write shown in Figure 3 and 4, *write(x,v)* triggers a *write_miss(x,v)* call if a WRITE_TOKEN is missing,

which in turn asks the home node for a write token by calling `write_token_request(x)`. Old or conflicting tokens are revoked by `write/read_token_revoke(x)`. A SC replica read, shown in Figure 5, can trigger a `read_miss(x)` if a READ_TOKEN is missing. When the home node issues the READ_TOKEN, it also tells the reader where the latest copy is (the latest writer) and the reader calls `value_request(x)` to fetch the copy. Vector clocks are also returned along with the latest copy by the latest writer in order to ensure the cross SC/CC causal relationship is correct.

For a CC replica write, a message is constructed and inserted into the outgoing queue (`oqueue`). Function `send_daemon()` will eventually send out the message and the destination side `receive_daemon()` will eventually receive it. A CC replica read is returned immediately if the replica is READABLE. As restricted by the constraints of Table 1, writes to a CC replica will not be propagated to SC replicas. However, we do propagate the meta data (vector clock, to be specific) from CC replicas to SC replicas when a write to CC replica happens. This ensures that the vector clocks on SC replicas can correctly capture the causal relationship even though they do not share the CC values.

4.3 Correctness of the Protocol

The protocol given in Figure 6 is correct because we can show that both SC and CC requirements are met in this protocol (due to page limitation, we only give a sketch of the proof in this paper).

(1) *If there are only SC (CC) replicas in the system, the protocol behaves the same as the home-based protocol [6] (causal memory protocol [2]) does. Both SC and CC requirements are met.*

(2) *If there are mixed SC and CC replicas in the system, the SC requirements are not violated because the definition of access constraints isolates the writes to CC replicas from SC replicas. These writes, which are not sequentially ordered, are not visible to SC replicas. Now we are trying to argue that CC requirements are not violated either. Let's suppose CC requirements are violated. There are two possibilities: either the causal order among CC objects is broken, or the causal order among SC and CC objects is broken. The correctness of the causal memory protocol guarantees that the first case does not happen. Our solutions proposed in Section 4.1 prevent the second case from happening. So we can be sure that causal order is correctly maintained in our protocol. Therefore, both SC and CC requirements are met.*

4.4 Communication and Host Failures

In this paper, we mainly focus on introducing the mixed consistency model and the protocol described in Figure 6

assumes a failure free environment for simplicity reasons. However, our protocol can be extended to handle communication and node failures. In a fault-tolerant version of the mixed consistency protocol that we are developing, we assume omission failure of communication channels and fail-stop failures of nodes where the processes execute. The omission failure is tolerated by the re-transmission support of the communication layer which our protocol is built upon. When a node fails, all the replicas maintained by that node fail too. The CC protocol used in our MC implementation can be easily extended to tolerate fail-stop failures, while the home-based SC protocol cannot. When object x 's home node D_x fails, all read and write operations on x block and x becomes inaccessible. However, our mixed consistency model provides a new way to address fail-stop failure handling through replica downgrading, which allows computation to continue at a lower consistency level when failures happen.

The downgrading process happens when the system detects that D_x has failed. The system will then downgrade all the SC replicas of x maintained by non-faulty nodes to CC copies. During the downgrading process, all the write operations on all replicas will be frozen in order to maintain correct causal relationship. The downgrading procedure can be summarized in three steps: 1) When a process detects that D_x has failed, it sends out a DOWNGRADING_START message to all the nodes in the system, initializing the start of downgrading (Duplicated DOWNGRADING_START messages will be ignored). 2) The actual downgrading process starts when the system has applied all outstanding writes. During the downgrading process, all new write requests will be delayed until the process is finished. Each node that has a SC copy of x will downgrade it to CC. 3) After a node finishes its downgrading, a LOCAL_DOWNGRADING_FINISH message will be sent to all other nodes. When a node receives all the LOCAL_DOWNGRADING_FINISH messages, the computation will continue with only CC copies of object x in the system. Since an application may want stronger consistency, we are also exploring protocols that can upgrade replicas after a node recovers from failures.

5 Analysis

A simple system is used to analyze the network traffic and timeliness of the mixed consistency protocol. It consists of $n + m$ processes (n SC processes and m CC processes) sharing one object x . We assume that the average one-way message transmission time is t . The dissemination is done by a tree-based protocol, with a fan-out factor of f . For example, the average time for a process to disseminate a new value to m processes is $(\log_f m)t$. We estimate the network traffic by the number of messages generated for different

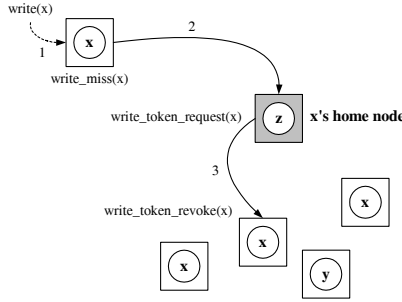


Figure 3. A sample illustration of write(x, v) (x is SC) with write token revocation.

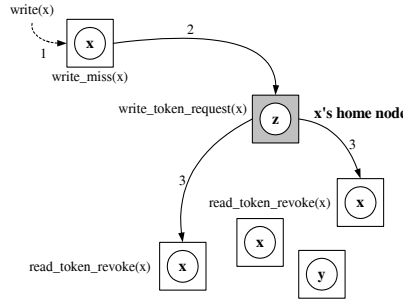


Figure 4. A sample illustration of write(x, v) (x is SC) with read token revocation.

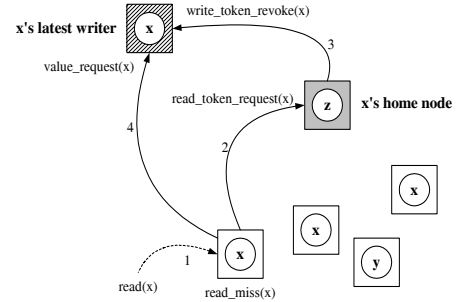


Figure 5. A sample illustration of read(x) (x is SC) with write token revocation.

operations. The timeliness of read operation is estimated by the time it takes to return a value, the timeliness of write operation is estimated by the time it takes for the rest of the system to become aware of its new value. We compare the mixed consistency protocol against SC consistency and CC consistency implementations in Figure 7.

SC consistency (home-based protocol implementation)

Let's assume x 's home node is p_0 .

- **READ:** Let's further assume process p_i ($\neq p_0$) does not have x 's READ_TOKEN and p_j ($\neq p_0$) is last writer of x , so a read request at p_i will generate additional control messages for requesting/revoking tokens.
 - Suppose there is no other readers between p_j 's write and p_i 's read, the number of messages for p_i to return the latest value of x is 6 (1 READ_TOKEN_REQUEST from p_i to p_0 , 1 WRITE_TOKEN_REVOKE from p_0 to p_j , 1 VALUE_REQUEST from p_i to p_j , and 3 REPLY messages for each of these messages). The total time for p_i to return a new value is $6t$.
 - Suppose there are other readers that read x 's value before p_i , the number of messages will be reduced to 4 (p_0 does not need to revoke p_j 's WRITE_TOKEN). And the total time is $4t$.
- **WRITE:** Let's assume process p_i ($\neq p_0$) does not have x 's WRITE_TOKEN. There are two possibilities: a process p_j ($\neq p_0$) holds x 's WRITE_TOKEN, or there are k processes holding x 's READ_TOKEN. The total number of messages for p_i to complete a write operation is 2 and $2(k+1)$ respectively. Therefore, the total time for p_i 's write is either $2t$ or $2(k+1)t$.
- For both **READ** and **WRITE**, if p_i already has the proper token, no messages will be generated. Thus we

say that the communication time each operation takes is 0.

CC consistency (vector clock based protocol)

- **READ:** Since there is no communication involved for serving read requests, no messages will be generated. And the time to complete a read request is 0.
- **WRITE:** A new object value (together with vector clock) needs to be disseminated to all other processes, when a write request is serviced. So a total number of $n+m-1$ messages will be generated. It takes an average of $t(\log_f(n+m-1))$ for this new object value to be disseminated to all other processes, although the write operation itself immediately returns before communication completes.

Mixed consistency

- **SC READ:** Same as **READ** in SC consistency, i.e. 4 or 6 messages and $4t$ or $6t$ time.
- **SC WRITE:** Similar as **WRITE** in SC consistency, except that m additional dissemination messages are generated because the writer needs to disseminate the new value to all CC replicas as well. Therefore, the total number of messages is either $2+m$ or $2(k+1)+m$. And the total execution time of write is $2t$ or $2(k+1)t$, the dissemination time is $\log_f m$.
- For both **SC READ** and **SC WRITE**, if p_i already has the proper token, no control messages will be generated (m additional dissemination messages are still generated for **SC WRITE**, taking $t(\log_f m)$ time to be disseminated) and both operations immediately return with 0 communication time.

Assume there are N processes in the system,
i.e. p_1, \dots, p_N .
 O : array of M objects. x : object name.
i.e. $O[x]$ contains value v of object x .

```
init()
//initializing the meta data
for every CC object  $x$  do
  set  $x$  as READABLE;
for every object  $x$  do
  for  $j = 1$  to  $N$  do
     $t_x[j] = 0$ ; //initialize the timestamp
oqueue =  $\langle \rangle$ ;
iqueue =  $\langle \rangle$ ;
```

```
write(x, v)
//write new value  $v$  to object  $x$ 
if  $x \in SC$  and  $p_i \in SC$ 
  if  $p_i$  has  $x$ 's WRITE_TOKEN
     $t_x[i] = t_x[i] + 1$ ;
     $O[x] = v$ ;
    enqueue(oqueue,  $\langle i, x, v, t \rangle$ );
  else write_miss( $x, v$ );
else if  $x \in CC$ 
   $t_x[i] = t_x[i] + 1$ ;
   $O[x] = v$ ;
  enqueue(oqueue,  $\langle i, x, v, t \rangle$ );
else throw WRITE_EXCEPTION;
```

```
read(x)
//read object  $x$ 's value
if  $x \in CC$  and  $p_i \in CC$ 
  if  $x$  is NOT READABLE
    wait until  $x$  is READABLE;
  return  $O[x]$ ;
else if  $x \in SC$ 
  if  $p_i$  has  $x$ 's READ_TOKEN
    return  $O[x]$ ;
  else return read_miss( $x$ );
else throw READ_EXCEPTION;
```

```
write_miss(x, v)
//wait for a WRITE_TOKEN from  $x$ 's home node  $p_j$ 
calls  $p_j$ .write_token_request( $x$ );
set ' $p_i$  has  $x$ 's WRITE_TOKEN'';
 $t_x[i] = t_x[i] + 1$ ;
 $O[x] = v$ ;
enqueue(oqueue,  $\langle i, x, v, t \rangle$ );
```

```
read_miss(x)
//wait for a READ_TOKEN from  $x$ 's home node  $p_j$ 
call  $p_k = p_j$ .read_token_request( $x$ );
//assume  $p_k$  has the latest value of  $x$ 
set ' $p_i$  has  $x$ 's READ_TOKEN'';
if  $p_k$  is null
  return  $O[x]$ ;
else
  call  $\langle O[x], s \rangle = p_k$ .value_request( $x$ );
  for any CC object  $z$  do
    if  $(\exists j \neq i: s_z[j] > t_z[j])$ 
      set  $z$  as NOT READABLE;
  if  $(s_x[i] > t_x[i])$ 
     $t_x[i] = s_x[i]$ ;
  return  $O[x]$ ;
```

```
value_request(y)
//assume it is called by process  $p_j$ 
return  $\langle O[y], t \rangle$ ; //return the entire timestamp array
```

```
write_token_request(y)
//assume it is called by process  $p_j$ 
if  $y$  has a WRITE_TOKEN issued to  $p_m$ 
  call  $p_m$ .write_token_revoke( $y$ );
  return; //grant  $p_j$  with  $y$ 's WRITE_TOKEN
if  $y$  have any READ_TOKENs issued
  for any process  $p_n$  being issued
    call  $p_n$ .read_token_revoke( $y$ );
  return; //grant  $p_j$  with  $y$ 's WRITE_TOKEN
return;
```

```
read_token_request(y)
//assume it is called by process  $p_j$ 
if  $y$  have any READ_TOKENs issued
  //assume  $p_k$  is the last process that had  $y$ 's WRITE_TOKEN
  return  $p_k$ ; //grant  $p_j$  with  $y$ 's READ_TOKEN
if  $y$  has a WRITE_TOKEN issued to  $p_m$ 
  call  $p_m$ .write_token_revoke( $y$ );
  return  $p_m$ ; //grant  $p_j$  with  $y$ 's READ_TOKEN
return null;
```

```
write_token_revoke(y)
//assume it is called by process  $p_j$ 
wait for outstanding writes to  $y$  are finished;
delete  $y$ 's WRITE_TOKEN;
return;
```

```
read_token_revoke(y)
//assume it is called by process  $p_j$ 
wait for outstanding reads to  $y$  are finished;
delete  $y$ 's READ_TOKEN;
return;
```

```
send_daemon()
//sending daemon, keeps running forever
if oqueue  $\neq \langle \rangle$ 
  //let MSG = dequeue(oqueue)
  if  $p_i \in CC$ 
    disseminate MSG to all other processes;
  else //  $p_i \in SC$ 
    disseminate MSG to all CC processes;
```

```
receive_daemon()
//receiving daemon, keeps running forever
if iqueue  $\neq \langle \rangle$ 
  //let  $\langle j, x, v, s \rangle = \text{head}(\text{iqueue})$  be the msg from  $p_j$ 
  if  $(\forall h \neq x \text{ and } \forall k \neq j: s_h[k] \leq t_h[k]) \text{ AND } (s_x[j] = t_x[j] + 1)$ 
    dequeue(iqueue);
     $t_x[j] = s_x[j]$ ;
    if  $p_i \in CC$ 
       $O[x] = v$ ;
    if  $x$  is NOT READABLE
      set  $x$  as READABLE;
```

Figure 6. The Mixed Consistency Protocol - at process p_i

		Network Traffic	Time
READ in SC	reader has token	0	0
	reader does not have token	4 or 6	4t or 6t
SC READ in MC	reader has token	0	0
	reader does not have token	4 or 6	4t or 6t
WRITE in SC	writer has token	0	0
	writer does not have token	2 or $2(k+1)$	2t or $2(k+1)t$
SC WRITE in MC	writer has token	m	0 execution, $\log_f m$ dissemination
	writer does not have token	$2+m$ or $2(k+1)+m$	2t or $2(k+1)t$ execution, $\log_f m$ dissemination
READ in CC	-	0	0
CC READ in MC	-	0	0
WRITE in CC	-	$n+m-1$	0 execution, $\log_f(n+m-1)$ dissemination
CC WRITE in MC	-	$n+m-1$	0 execution, $\log_f(n+m-1)$ dissemination

Figure 7. Analysis of mixed consistency protocol

- *CC READ*: Same as *READ* in *CC* consistency.
- *CC WRITE*: Similar as *WRITE* in *CC* consistency, $m - 1$ dissemination messages are generated. In order to prevent “possible vector clock error”, the writer needs to send its vector clock to all *SC* processes. So n control messages are generated. A total number of $n + m - 1$ messages will be generated. The time is also the same as *WRITE* in *CC* consistency.

To summarize, the network traffic and timeliness of *SC*, *CC* and *MC* models are shown in Figure 7. It is clear to see that the performance of mixed consistency protocol is almost the same as *SC* and *CC*, depending on what tag the reader/writer has. The overall performance of an *MC* system is determined by how many *SC/CC* replicas are there in the system.

6 Related Work

Agrawal et. al. first introduced the term *mixed consistency* in [1] to refer to a parallel programming model for distributed shared memory systems, which combines causal consistency and *PRAM* [13] consistency. Four kinds of explicit synchronization operations: read locks, write locks, barriers and await operations are provided in their model. In this paper, we use the same term to refer to a new consistency model combining sequential consistency and causal consistency. The access constraints we propose represent a more general approach: *PRAM* consistency can be integrated into our mixed consistency model with little effort.

Attiya and Friedman introduced the concept of *hybrid consistency* in [4]. In this model, all read and write operations on shared objects are categorized as either strong or weak. All processes agree on a sequential order for all strong operations, and on the program order for any two operations issued by the same process in which at least one of them is strong. It does not guarantee any particular order of any two weak operations between two strong operations. In our model, an operation is weak or strong depending on whether it is executed with a strong or weak replica. Thus, both strong and weak operations can not be executed on a replica by the same process. We define access constraints to develop the mixed consistency model where we do not assume that strong operations can be used to establish order among weak operations when necessary.

Fernandez, Jimenez, and Cholvi proposed a simple algorithm to interconnect two or more causal consistency systems into one causal consistency system through *gates* in [8]. They further defined a formal framework to describe the interconnection of distributed shared memory systems in [9] and showed that only fast memory models can be interconnected, where read and write operations return immediately after only local computations (i.e. no global syn-

chronization, which is required to implement *SC* systems). In our mixed consistency model, we take a different access constraints based approach to combine *SC* and *CC* together. We do not try to interconnect *SC* and *CC* through *gates*, where all the communications between two systems must flow through. Therefore, our result does not contradict theirs.

Yu and Vahdat presented a *conti-based continuous consistency* model for replicated services in paper [20]. In their model, applications use three measurements of numerical error, order error and staleness as bounds to quantify their relaxed consistency requirements. The authors used relaxed consistency model with bounded inconsistency to balance the tradeoffs between performance, consistency and availability. By exploring application and system resource heterogeneity, we also suggest that replicated distributed services should be able to support more than just strong consistency. However, instead of relaxing strong consistency requirements, we believe that certain applications can benefit from having both strongly and weakly consistent replicas at the same time.

Raynal and Mizuno survey many consistency models for shared objects in [15]. They particularly emphasize linearizability, *SC*, hybrid consistency and *CC*. Raynal and Schiper [16] show that *SC* is strictly stronger than *CC* by showing that $CC + \text{“total order on all writes on all objects”} = SC$. Based on the results of these two papers, we choose *SC* and *CC* as the example of strong and weak consistency to implement the mixed consistency model in this paper.

Two orthogonal dimensions of consistency, timeliness and ordering, are explored in [3]. **Ordering** determines the value that should be applied/returned by the consistency protocols. Sequential consistency [12], [5], [16] requires all operations appear to be executed in a serial order that respects the “read-from” ordering and “process” ordering. Causal consistency [2] is a weaker form of consistency model because it does not require all replicas to agree on a unique execution history. There are various other consistency models (e.g. lazy release consistency [10] and Bayou session guarantees [18]), which explore different aspects (e.g. synchronization, transaction) of distributed computation and how extra facility can be used to enhance consistency models. **Timeliness** decides how fast the proper value should be applied/returned. Maintaining web content consistency [14] and δ -time consistency [17] suggest that the value returned should be at most δ time unit “old”. [11] presents an efficient implementation of timed consistency based on combined “push” and “pull” techniques. Our work fits into the operation ordering dimension of consistency model. The mixed consistency model tries to combine different order guarantees together to satisfy application needs in wide area heterogeneous environment.

7 Conclusion

In this paper, we propose a mixed consistency model to combine existing strong (SC) and weak (CC) consistency models together to meet the heterogeneity challenge for large scale distributed shared memory systems. Our model defines new SC and CC requirements on subsets of replicas shared by different processes. We propose access constraints as the base to implement our mixed consistency model. We make minor modifications to two existing protocols and combine them together to implement the mixed consistency model. We show that the result protocol satisfies all the requirements we define.

The access constraints based approach we proposed in this paper is not limited to combining just SC and CC. Currently we are exploring the combination of other consistency schemes under similar access constraints. We'd like to implement a fault-tolerant version of our protocol in order to better meet the application needs under real life conditions. And as an extension of our work, we are building our protocols on top of an adaptive update dissemination framework, which can provide better performance and bandwidth utilization in a heterogeneous environment.

8 Acknowledgements

We thank anonymous reviewers and our shepherd Dr. Neeraj Suri. Their comments have greatly improved this paper.

References

- [1] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Mixed consistency: a model for parallel programming (extended abstract). In *The 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.
- [2] M. Ahamad, G. Neiger, J. E. Burns, P. W. Hutto, and P. Kohli. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9:37–49, 1995.
- [3] M. Ahamad and M. Raynal. Ordering vs timeliness: Two facets of consistency? *Future Directions in Distributed Computing*, 2003.
- [4] H. Attiya and R. Friedman. A correctness condition for high-performance multiprocessors. In *Proc. of the 24th ACM Symposium on Theory of Computing (STOC)*, 1992.
- [5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 1994.
- [6] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.
- [7] A. Cox, E. de Lara, and W. Z. Y. C. Hu. A performance comparison of homeless and home-based lazy release consistency protocols for software shared memory. In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, 1999.
- [8] A. Fernandez, E. Jimenez, and V. Cholvi. On the interconnection of causal memory systems. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.
- [9] E. Jimenez, A. Fernandez, and V. Cholvi. Decoupled interconnection of distributed memory models. In *Proc. of the 7th International Conference on Principles of Distributed Systems (OPODIS 2003)*, December 2003.
- [10] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.
- [11] V. Krishnaswamy, M. Ahamad, M. Raynal, and D. Bakken. Shared state consistency for time-sensitive distributed applications. In *21th International Conference on Distributed Computing Systems (ICDCS)*, 2001.
- [12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691, 1979.
- [13] R. Lipton and J. S. Sandberg. Pram: A scalable shared memory. *Technical Report CS-TR-180-88, Princeton University, Dept. of Computer Science*, 1988.
- [14] C. Liu and P. Cao. Maintaining strong cache consistency in the world-wide web. In *International Conference on Distributed Computing Systems (ICDCS)*, 1997.
- [15] M. Raynal and M. Mizuno. How to find his way in the jungle of consistency criteria for distributed objects memories (or how to escape from minos' labyrinth). In *Proc. of the IEEE International Conference on Future Trends of Distributed Computing Systems*, September 1993.
- [16] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. In *15th Conference on Foundations of Software Technologies and Theoretical Computer Science*, pages 180–194, Bangalore, India, 1995. Springer-Verlag.
- [17] A. Singla, U. Ramachandran, and J. K. Hodgins. Temporal notions of synchronization and consistency in beehive. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 211–220, 1997.
- [18] D. B. Terry, M. M. Theimer, K. Peterson, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM SOSP*, 1995.
- [19] B.-H. Yu, Z. Huang, S. Cranefield, and M. Purvis. Homeless and home-based lazy release consistency protocols on distributed shared memory. In *27th Australasian Computer Science Conference (ACSC'04)*, 2004.
- [20] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. In *ACM Transactions on Computer Systems (TOCS)*, 2002.
- [21] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.