

# Consistency Models

## ABSTRACT

### 1. INTRODUCTION

The proliferation of Internet-scale services, accessed by millions of users scattered across the globe led to the appearance of a myriad of distributed storage systems that support those services [4, 6, 7, 2]. To achieve both fault-tolerance and low-latency in serving clients' requests, these distributed storage systems often replicate their stored data across several, and sometimes geographically distant, computing nodes.

Traditionally these storage systems provided, the so-called, strong consistency, i.e., the system behaved as if only one replica of the data existed. But due to the limitations evidenced by the CAP theorem, and the strong requirements of high availability, system designers shifted from strong consistency towards weaker forms of consistency, which provided better performance and reduced latencies at the cost of increasing the complexity of writing correct applications that make use of the storage system.

A consistency model defines the possible values that can be retrieved from the storage system with respect to all operations that occurred in the past. There is a wide spectrum of weak consistency models and programmers are now faced with the challenge of understanding the behaviour of their applications when executed on a particular weak consistency model that precludes a sequential reasoning of the program execution. Besides the non-sequential reasoning being already hard to tackle by programmers, the specifications of consistency models provided by each system are usually defined in their own language, and not necessarily a formal one.

A good example of such problem is the seminal definition of the session guarantees by Terry et al. [9]. The operational definition of the read-your-writes, present in Terry's paper, is subject to different interpretations throughout the literature [?].

This misinterpretation of ambiguous definitions leads to several problems that most of the times are materialised as bugs in distributed applications. For instance, the programmer may write its application considering a stronger interpretation of a consistency model, and thus during a real execution, an unaccounted behaviour may occur and lead to a program crash.

We briefly summarize the consequences of defining consistency models using informal languages or using different formalisations:

- ambiguous definitions lead to different interpretations of the same consistency model, and a wrong interpretation leads to programs bugs;
- hard to compare different consistency models which in turn makes very difficult for the programmer to choose the better storage system for his application;
- different definitions are developed for the same consistency model;
- when definitions depend on operational details it is difficult to use such definition to describe the semantics of your own storage system implementation, which leads to the creation of a new definition.

The above problems are amplified even more by the fact that storage systems were built by different research communities that use different terminology, and need different consistency guarantees for their own systems. For instance, the session guarantees widely known in replicated systems, are weaker than the weakest consistency model used by shared memory systems. Thus, we strongly believe that is important find a common language to describe the different consistency models without relying in implementation details, and also that the language should be easily understood by non-experts such as usual application programmers.

The objective of our work is to develop a theoretical framework to describe the several consistency models proposed in the literature, either for shared memory or replicated systems, in a way that is simple enough for programmers to understand the behaviour of their applications when using a particular storage system. Inspired by Atul Adya's work, we use a graph based representation to describe consistency models, and contrary to the usual axiomatic definitions that express what are the valid behaviours of an execution, our definitions expresses what behaviours are forbidden during an execution.

To define the consistency models from so different storage systems we need to abstract completely any system details and rely solely in a uniform interface between the clients and the storage system. This is a similar idea as described in [8] where consistency models are defined in terms of what can be observed.

With the development of this work we hope to clarify the subtleties between different consistency models, to define a precise hierarchy of consistency models, and help programmer to understand the outcomes of their programs when executed under a particular consistency model. As an example, we will later show some equivalences that we found between consistency models that, in our knowledge, were not known until now. Moreover, our system agnostic specification of consistency models can also be used in several practical use cases such as in the verification of real execution traces for consistency violations, or to detect execution anomalies generated by applications using program analysis as in [5].

## 2. THE CHAOTIC WORLD OF CONSISTENCY MODELS

- Present a list of most existing consistency models.
- Describe why shared-memory systems are not so different from replicated systems in the perspective of a consistency model.
- Talk briefly of the different approaches to describe consistency models (most are axiomatic approaches).
- Identify the consistency models that are equivalent (we show this equivalence in using our framework later in the paper).
- Organize all consistency models in a table.
- Talk about why serializability is not interesting for consistency models.
- Make the bridge between what Atul made with isolation levels and what we are going to do with consistency models.

## 3. A FRAMEWORK TO SPECIFY CONSISTENCY MODELS

### 3.1 System model and assumptions

We consider a storage system that exposes two operations to its clients: a read operation and write operation. Operations are executed in the context of a session, where a session corresponds to a sequence of read/write operations issued by a single client. A read operation, denoted as  $r_s(x)v$ , executes in the context of session  $s$  and returns the value  $v$  of a key  $x$ , and a write operation, denoted as  $w_s(x, v)$ , executes in the context of session  $s$  and assigns the value  $v$  to the key  $x$ . From the perspective of a session, operations are executed synchronously, i.e., the session can only execute a new operation after receiving the result of the previous issued operation.

Nothing is known about the way these operations are implemented by the storage system.

We assume that each write operation  $w_s(x, v)$  always assigns a unique value  $v$ . Thus, when a read operation  $r_s(x)v$  returns a value  $v$  we can identify univocally which write operation assigned such value.

We will use the first letters of alphabet  $a, b, \dots$  to denote session identifiers throughout the document, and the set that contains all sessions is denoted as  $\mathcal{S}$ .

We assume that objects in the data store are similar to read/write registers with the usual sequential specification where read operations should always return the value written by the last write operation.

### 3.2 Consistency Histories

Each client session execution is described by a *session history* that is composed by a sequence of read and write operations issued within the same session. We denote a session history as  $\langle o_1, \dots, o_n \rangle_s$ , where  $o_i$  may be a read or write operation, and  $s$  is a session identifier. The operations in a session  $S$  are totally ordered according to the relation  $\prec$ :

$$\forall o_i, o_j \in S : i < j \Rightarrow o_i \prec o_j$$

A consistency history, which describes a particular execution of a set of clients, is defined as a set of session histories  $CH = \{S_1, \dots, S_k\}$ .

### 3.3 Consistency Graphs

A consistency graph is a graph-based representation of a consistency history, where the nodes correspond to the operations present in the session histories, and the edges correspond to order relations between operations.

A consistency graph represents an explanation of the order between operations present in a consistency history.

There are two kinds of dependencies in a consistency graph. The first kind, which we call *explicit* dependencies, are directly extracted from the consistency history. The second kind, which we call *implicit* dependencies, are inferred from the *explicit* dependencies of a consistency graph.

The *explicit* dependencies represent the causal order between operations. For instance, if a read operation  $r$  returns the value written by a write operation  $w$  then  $w$  must be ordered before  $r$  and therefore there is an explicit dependency from  $w$  to  $r$ . We call this explicit dependency as *write-read dependency*. Another kind of explicit dependency is called the *session dependency*, which connects two consequent operations belonging to the same session.

The *explicit* dependencies are constructed according to the following rules:

- session dependency  $o \xrightarrow{s_a} o'$ : if  $o$  and  $o'$  are operations from the same session history identified by  $a$  and  $o \prec o' \wedge \nexists o'' : o \prec o'' \prec o'$ .
- write-read dependency  $o_w \xrightarrow{w_r} o_r$ : if read operation  $o_r = r_s(x)v$  returned the value  $v$  written by a write operation  $o_w = w_{s'}(x, v)$ .

The sequential specification of a read/write register, as used in our model, states that read operations should always return

the value written by the last write operation. To verify that read operations in a consistency history satisfy the sequential specification of objects, we would have to know in which order all write operations for the same object were executed. Since we treat the storage system as a black box we cannot possibly know the order of write operations, but we can rely on the observations made by the several sessions to find restrictions to a possible ordering of the write operations. These order restrictions will be materialised in the form of *implicit* dependencies over write operations called write-write dependencies.

We can say that a write operation  $w$  must happen before another write operation  $w'$  if there is an explicit path<sup>1</sup> in the consistency graph connecting  $w$  to a read operation that reads the value written by operation  $w'$ . In other words, if there is a read operation  $r$  in the causal future of the write operation  $w$  that reads a value written by a different write operation  $w'$  then operation  $w$  must have happened before operation  $w'$ , otherwise the read operation would have read the value written by  $w$ .

We can define the construction rule of implicit write-write dependencies as:

- write-write dependency  $o_w \xrightarrow{ww} o'_w$ : there is a path of *explicit* dependencies from the write operation  $o_w = w_s(x, v)$  to some other read operation  $o_r = r_{s'}(x)v'$  that reads the value written by the write operation  $o'_w = w_{s''}(x, v')$ .

When defining some consistency models, specially more weaker<sup>2</sup> ones, the write-write dependencies may be inferred from more restrictive conditions. Whenever such restrict form is necessary, we will specify the write-write dependency construction rule in the context of the consistency model being defined.

Besides the restrictions to the ordering between write operations, we can also find restrictions to the ordering between read and write operations. We define an implicit read-write dependency similar to the *anti-dependency* defined in [1]. This dependency between a read operation  $r$  and a write operation  $w$  restricts  $r$  to have happened before  $w$  if we try to find a total order between all operations of a causal history.

Implicit dependencies between read and write dependencies are inferred from the dependencies between write operations. We can say that a read operation  $r$  must happen before a write operation  $w$  if the write operation  $w'$  that wrote the value read by operation  $r$  is ordered before  $w$  by a write-write dependency. In other words, if we know that a write operation  $w'$  must have happened before a write operation  $w$  then any read operation that reads the value written by  $w'$  must have also happened before the write operation  $w$ , otherwise the read operation would have read the value written by  $w$ .

We can define the construction of implicit read-write dependencies as:

- read-write dependency  $o_r \xrightarrow{rw} o_w$ : let  $o'_w = w_{s''}(x, v)$  be the write operation that writes the value read by  $o_r =$

<sup>1</sup>path composed solely by explicit dependencies.

<sup>2</sup>The *weaker* relation will be formally defined later in the paper.

$r_s(x)v$ , there is a write-write dependency from  $o'_w$  to  $o_w = w_{s'}(x, v')$ .

### 3.4 Consistency Anomalies

Don't know if we need this section. The idea is just to say that an anomaly corresponds to a cycle in the consistency graph.

## 4. CONSISTENCY MODELS

A consistency model definition consists of a set of precluded graph cycles plus a construction rule for write-write dependencies. We can use a consistency model definition to verify if a given consistency history violates or not the consistency model. To that end, we construct the consistency graph using the construction rule of the consistency model, and check if the resulting consistency graph contains any of the cycles precluded by the consistency model. If the graph contains a cycle then it violates the consistency model.

### 4.1 Read-Your-Writes

The read-your-writes (RYW) consistency model was first proposed in [9]. Often designated as one of the *session guarantees*, it guarantees that a read operation issued in a session will observe a state that reflects at least all the writes issued by the same session.

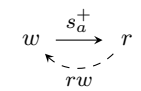
The RYW model was defined with an operational specification in [9]. Later, Chockler et al. defined the same model in [3] using an axiomatic specification that captures the client perspective.

As we described in the previous section, we define a consistency model as a set of precluded cycles and a construction rule for the write-write dependencies. We define the RYW consistency model as:

**DEFINITION 4.1 (RYW MODEL).** *Given a session  $a \in S$ , the RYW model orders the write operations according to the write-write rule (W1):*

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \\ \exists_{r(x)} : w(x) \xrightarrow{s_a^+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

and precludes the following anomaly (A1):



Please note that, in the above definition, session  $a$  is existentially quantified and is the same for both the write-write rule and the session path in the cycle.

The RYW model does not give any guarantees on the order of write operations, nor the order of read operations, issued in the same session. The only guarantee that is given is in the order between write operations and subsequent read operations issued in the same session. Rule W1 captures this ordering guarantee.

The anomaly A1 captures the occurrence of a violation in a history where is guaranteed the ordering between write and read operations (forced by rule R1). Intuitively, A1 presents a situation where the read operation  $r$  has read a value that was

written by a write operation  $w'$  that is known to have happened before the write operation  $w$ , although  $r$  was issued after  $w$ .

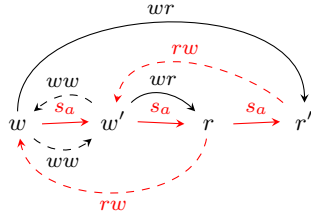
We can informally state the RYW consistency model as:

**DEFINITION 4.2 (INFORMAL RYW).** *RYW guarantees that, when considering only operations of the same client, is not possible to read a value that was written before the last read operation that was issued.*

#### 4.1.1 Violation Example

The following history shows a situation that is impossible to occur under the RYW consistency model.

$$\{\langle w(x, 1), w'(x, 2), r(x)2, r'(x)1 \rangle_a\}$$



Under the RYW model, the second read operation  $r'$  could not read the value written by operation  $w$  as it occurred before the read operation  $r$  that already established that the value written by  $w'$  is the more recent value.

## 4.2 Monotonic Reads

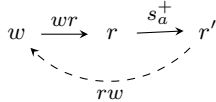
Monotonic reads (MR) consistency is a session guarantee [9] that ensures that any two read operations in the same session return an increasingly up-to-date state of the storage system over time. As in the previous consistency model, the MR model was first defined with an operational specification in [9], and later with an axiomatic specification in [3].

The MR consistency model is defined in our framework as:

**DEFINITION 4.3 (MR MODEL).** *Given a session  $a \in \mathcal{S}$ , the MR model orders the write operations according to the write-write rule (W2):*

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad \exists_{r(x), r'(x)} : w(x) \xrightarrow{wr} r(x) \\ \wedge r(x) \xrightarrow{s_a^+} r'(x) \wedge w'(x) \xrightarrow{wr} r'(x)$$

and precludes the following anomaly (A2):



In the MR model the order of write operations is given by the order of read operations of a single client. This ordering is captured by the rule W2.

The anomaly A2 captures the occurrence of a violation in a history where is guaranteed the ordering between read operations (forced by rule W2). Intuitively, A2 presents a situation where the read operation  $r'$  has read a value that was written

by a write operation  $w'$  that is known to have happened before the write operation  $w$  that was read by the previously read operation  $r$ .

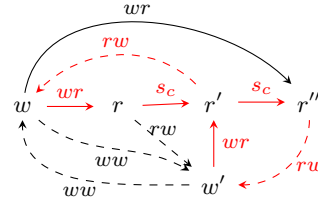
We can informally state the MR consistency model as:

**DEFINITION 4.4 (INFORMAL MR).** *MR guarantees that we always either read the last value read or a value that was never read before.*

#### 4.2.1 Violation Example

The following history shows a situation that is impossible to occur under the MR consistency model.

$$\{\langle w(x, 1) \rangle_a, \langle w'(x, 2) \rangle_b, \langle r(x)1, r'(x)2, r''(x)1 \rangle_c\}$$



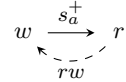
Under the MR model, the third read operation  $r''$  could not read the value written by operation  $w$  as it was not the last value read and it was already read by a past read operation from the same client.

## 4.3 Local Monotonic-Reads

**DEFINITION 4.5 (LMR MODEL).** *Given a session  $a \in \mathcal{S}$ , the LMR model orders the write operations according to the write-write rule (W2):*

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad \exists_{r(x), r'(x)} : w(x) \xrightarrow{wr} r(x) \\ \wedge r(x) \xrightarrow{s_a^+} r'(x) \wedge w'(x) \xrightarrow{wr} r'(x)$$

and precludes the following anomaly (A1):



## 4.4 Writes-Follows-Reads

Write-follows-read (WFR) consistency is a session guarantee that ensures that the effects of a write operation issued after a read operation in the same session, should be applied in a storage state that reflects the observation of the read operation.

In [9] is presented an operational definition, and in [3] is presented an axiomatic definition although with a different name. In [3], the WFR consistency model is called *session causality*.

Still in [3], is defined a new consistency model called read-before-writes, which cannot even be expressed in the model presented in our paper because, the consistency model is defined under the assumptions that session operations are asynchronous, i.e., the client can execute an operation without waiting for the response of the previous operations.

The WFR consistency model as presented in [9] and in [3], as session causality, is very different from the consistency models presented until now, as the main target of the guarantees

given by the consistency model is not the clients but rather the system. The original definition [9] is completely omisse on the guarantees provided to clients, and as a consequence several interpretations may emerge on what should be the behavior of clients within this consistency model.

We believe that consistency models should always describe the possible behaviors of clients, and not the behaviors of systems. In the next section we discuss the possible pitfalls of the original WFR definition.

#### 4.4.1 Terry's WFR Discussion

The write-follows-read definition presented in [9] is stated below:

**WFR-guarantee:** If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1, t1, R1) is also in DB(S2) and WriteOrder(W1, W2).

The above definition guarantees that in any server  $S_k$  that received both write operations W1 and W2, the order WriteOrder(W1, W2) is respected. But, it is impossible to guarantee that a client always observe W1 and W2 by the order defined by WriteOrder(W1, W2).

Consider the following example. A client C1 performs a read operation on S1, which read the value updated by a write operation W1, and then performs a write operation W2 in server S2, as stated in the above definition. Now a second client C2, performs a read R2 on server S2, which reads the value updated by W2, and then performs another read R3 on server R1, before the arrival of update W2 to server S1. The second read operation R3 will observe the value of the update W1, and from the perspective of client C2, the observed order of writes is W2 before W1.

The observation of client C2 in the above example contradicts the following statement presented in [9]:

*[...] Not only does the session observe that the Writes it performs occur after any Writes it had previously seen, but also all other clients will see the same ordering of these Writes regardless of whether they request session guarantees.*

Given the above contradiction, there is space for a lot of different interpretations of what guarantees should be provided by WFR. In the following sections we present two different interpretations, each providing different guarantees to clients.

#### 4.4.2 Local Write-Follows-Read

Our first interpretation of WFR is called *local write-follows-read* (LWFR), which only provides guarantees for a single session, following the philosophy of the previous consistency models presented so far.

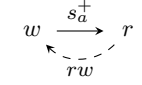
The LWFR consistency model is defined in our framework as:

**DEFINITION 4.6 (LWFR MODEL).** *Given a session  $a \in S$ , the LWFR model orders the write operations according to*

*the write-write rule (W3):*

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \\ \exists_{r(x)} : w(x) \xrightarrow{wr} r(x) \wedge r(x) \xrightarrow{s_a^+} w'(x)$$

*and precludes the following anomaly (A1):*



The LWFR establishes an order between the writes already observed and the writes to be performed. This ordering guarantees that a single client will never observe a value that was already observed and was written before a write operation to the same key by the same client.

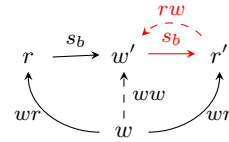
We can informally state the LWFR consistency model as:

**DEFINITION 4.7 (INFORMAL RYW).** *LWFR guarantees that, is not possible for a client to read a value already observed that was updated by the same client afterwards.*

#### 4.4.3 Violation Example

The following history shows a situation that is impossible to occur under the LWFR consistency model.

$$\{\langle w(x, 1) \rangle_a \langle r(x) 1, w'(x, 2), r'(x) 1 \rangle_b\}$$



Under the LWFR model, the second read operation  $r'$  could not read the value written by operation  $w$  as it was already observed by operation  $r$  and then overridden by the value written in operation  $w'$ .

#### 4.4.4 Write-Follows-Read

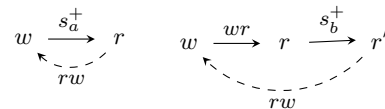
Our definition of WFR consistency model is constructed by strengthening the LWFR consistency model with an additional anomaly to be precluded. Thus, our WFR consistency model is strictly stronger than the LWFR anomaly.

The WFR model guarantees that any client will observe the write order imposed by rule R3, instead of only the client that establishes the order as in the LWFR model.

**DEFINITION 4.8 (WFR MODEL).** *Given sessions  $a, b \in S$ , the WFR model orders the write operations according to the write-write rule (W3):*

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \\ \exists_{r(x)} : w(x) \xrightarrow{wr} r(x) \wedge r(x) \xrightarrow{s_a^+} w'(x)$$

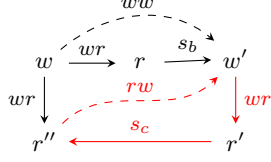
*and precludes the following anomalies (A1, A2):*



#### 4.4.4.1 Violation Example.

The following history shows a situation that is impossible to occur under the WFR consistency model.

$$\{\langle w(x, 1) \rangle_a \langle r(x) 1, w'(x, 2) \rangle_b, \langle r'(x) 2, r''(x) 1 \rangle_c\}$$



Under the WFR model, the second read operation  $r''$  of session  $c$  could not read the value written by operation  $w$  as it was ordered before write  $w'$  due to session's  $b$  actions.

### 4.5 Monotonic Writes

The monotonic-writes (MW) consistency model, is a session guarantee, originally defined in [9]. As in the previous consistency model, the original operational definition in [9] and the axiomatic definition in [3] of the MW model, does not give any guarantees to what clients must observe. The MW definitions in [9] and in [3] only guarantee that the system will respect the ordering of write operations performed by the same client.

Also as in the previous section, we present two interpretations of the MW consistency model. The first, gives ordering guarantees only to the client that performs the write operations. The second gives ordering guarantees to other clients as well.

#### 4.5.1 Local Monotonic Writes

DEFINITION 4.9 (LMW MODEL). Given a session  $a \in S$ , the LMW model orders the write operations according to the write-write rule (W4):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{s_a^+} w'(x)$$

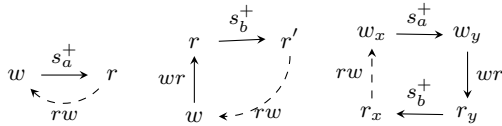
and precludes the following anomaly (A1):

#### 4.5.2 Monotonic Writes

DEFINITION 4.10 (MW MODEL). Given sessions  $a, b \in S$ , the MW model orders the write operations according to the write-write rule (W4):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{s_a^+} w'(x)$$

and precludes the following anomalies (A1, A2, A3):



### 4.6 Slow Consistency

DEFINITION 4.11 (SLOWC MODEL). A system guarantees the slow consistency model if guarantees  $RYW + LMR + MR + LWFR + LMW$ .

### 4.7 PRAM

DEFINITION 4.12 (PRAM MODEL). A system guarantees the PRAM model if guarantees  $SlowC + MW$ .

### 4.8 Cache Consistency

DEFINITION 4.13 (CACHEC MODEL). A system guarantees the Cache consistency model if guarantees  $SlowC + WFR$ , and additionally, the CacheC model orders the write operations according to the write-write rule ( $W1+W2+W3+W4$ ):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if}$$

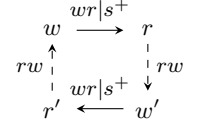
$$\vee \exists_{r(x)} : w(x) \xrightarrow{s^+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

$$\vee \exists_{r(x), r'(x)} : w(x) \xrightarrow{wr} r(x) \wedge r(x) \xrightarrow{s^+} r'(x) \wedge w'(x) \xrightarrow{wr} r'(x)$$

$$\vee \exists_{r(x)} : w(x) \xrightarrow{wr} r(x) \wedge r(x) \xrightarrow{s^+} w'(x)$$

$$\vee w(x) \xrightarrow{s^+} w'(x)$$

and precludes the following anomaly (A4):



### 4.9 Causal Consistency

DEFINITION 4.14 (CAUSALC MODEL). A system guarantees the Causal consistency model if guarantees  $PRAM+WFR$ , and additionally, the CausalC model orders the write operations according to the write-write rule ( $W6+W7$ ):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{+} w'(x)$$

$$\vee \exists_{r(x)} : w(x) \xrightarrow{+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

and precludes the following anomaly (A5):

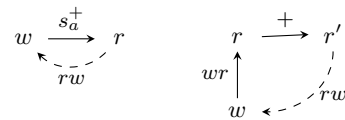
### 4.10 Partial-Store-Order

DEFINITION 4.15 (PSO MODEL). A system guarantees the Partial-Store-Order (PSO) consistency model if guarantees  $CacheC + PRAM$ , and additionally, the PSO model orders the write operations according to the write-write rule ( $W6+W7$ ):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{+} w'(x)$$

$$\vee \exists_{r(x)} : w(x) \xrightarrow{+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

and precludes the following anomalies (A1, A6):



## 4.11 Causal+ Consistency

DEFINITION 4.16 (CAUSAL+ MODEL). A system guarantees the Causal+ model if guarantees CausalC + PSO.

## 4.12 Processor Consistency

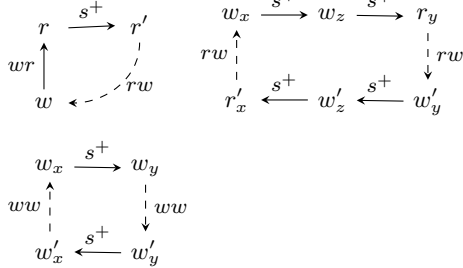
DEFINITION 4.17 (PROCESSORC MODEL). A system guarantees the Processor consistency model if guarantees CacheC + PRAM, and additionally, the ProcessorC model orders the write operations according to the write-write rule (W1+W4+W5):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad \exists_{r(x)} : w(x) \xrightarrow{s^+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

$$\vee \quad w(x) \xrightarrow{s^+} w'(x)$$

$$\vee \quad \exists_{w(y), r(y)} : w(x) \xrightarrow{s^+} w(y) \wedge w(y) \xrightarrow{wr} r(y) \wedge r(y) \xrightarrow{s^+} w'(x)$$

and precludes the following anomalies (A2, A7, A8):

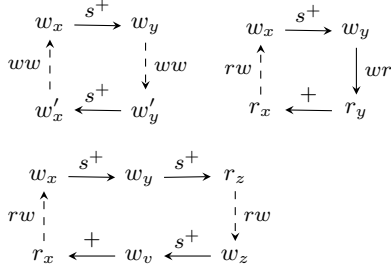


## 4.13 Total-Store-Order

DEFINITION 4.18 (TSO MODEL). A system guarantees the TSO consistency model if guarantees Causal+, and additionally, the TSO model orders the write operations according to the write-write rule (W6):

$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{+} w'(x)$$

and precludes the following anomaly (A7, A9, A10):



## 4.14 Sequential Consistency

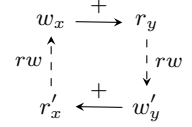
DEFINITION 4.19 (SC MODEL). A system guarantees the SC consistency model if guarantees TSO + ProcessorC, and

additionally, the SC model also orders the write operations according to the write-write rule (W6+W7):

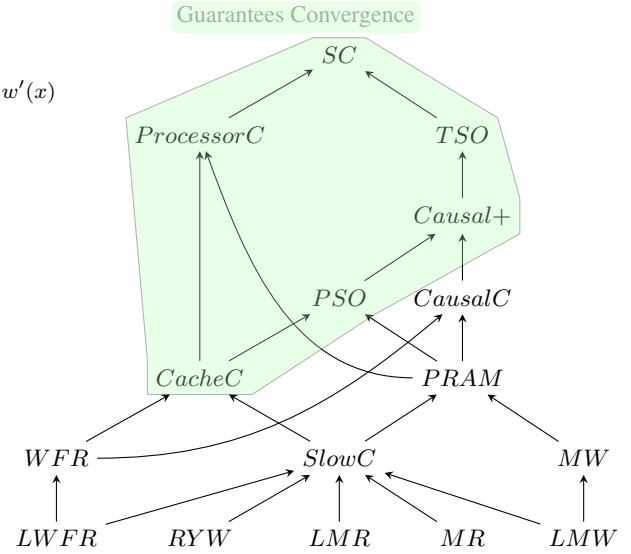
$$w(x) \xrightarrow{ww} w'(x) \quad \text{if} \quad w(x) \xrightarrow{+} w'(x)$$

$$\vee \quad \exists_{r(x)} : w(x) \xrightarrow{+} r(x) \wedge w'(x) \xrightarrow{wr} r(x)$$

and precludes the following anomaly (A11):



## 5. CONSISTENCY HIERARCHY



## 6. AVAILABILITY ANALYSIS

In the context of distributed storage systems, there is a very important concern regarding whether a consistency model can be implemented in a *highly-available* way. Availability means that the storage service should eventually respond to any client request even in the presence of network partitions.

In this section we will categorize the consistency models according to the availability property.

As a first step, we show that any consistency model that only precludes anomalies containing only a single *rw*-dependency can be implemented in an *available* way.

Imagine an implementation where for each write operation, besides storing the value, we also store the entire causal history of the operation. Dually, when a client reads a value from the storage, it also receives the entire causal history of such value.

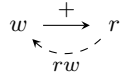
Such implementation is not very realistic as the size of messages would have to be proportional to the number of operations in the entire system, but nevertheless is still a possible implementation.

When an execution's consistency graph contains a cycle with a single *rw*-dependency, it means that one read operation has not read the most recent value of a key according to the causal history of that key. Therefore, if an implementation wants to

avoid creating such cycle, it only has to guarantee that always returns the most recent value according to the causal history, or a new value that does not exist yet in the causal history.

It is simple to see that in the presence of network partitions where new write operations may not be propagated to all servers, servers just need to rely on the causal history of each key stored in the storage system to return a consistent value without contacting any other replica. The consequence of network partitions in these consistency models is that implementations may return stale values, according to real-time ordering of write operations, but will never return a value that violates the causal ordering.

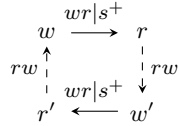
All consistency models that descend from the Causal Consistency model, including Causal Consistency, may be implemented in an available way according to the above argument. More formally, any consistency model that only precludes a cycle of the following form:



mented in an available way.

Now we analyse the the consistency models that preclude

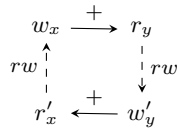
the following form of a cycle:



that all consistency models that only preclude the above cycle can also be implemented in an available way.

The above cycle represents the situation where two concurrent writes for the same key, are read in different orders by different clients. In order to prevent such situation there must be a mechanism that ensures the ordering of any two concurrent writes. We can use any deterministic conflict resolution mechanism, one instance is the mechanism used in the COPS [6] system, to deterministically impose the same ordering of all concurrent writes across all replicas. Since the mechanism is deterministic there is no need for communication between replicas and therefore the mechanism is tolerant to network partitions.

The third cycle form is the following:



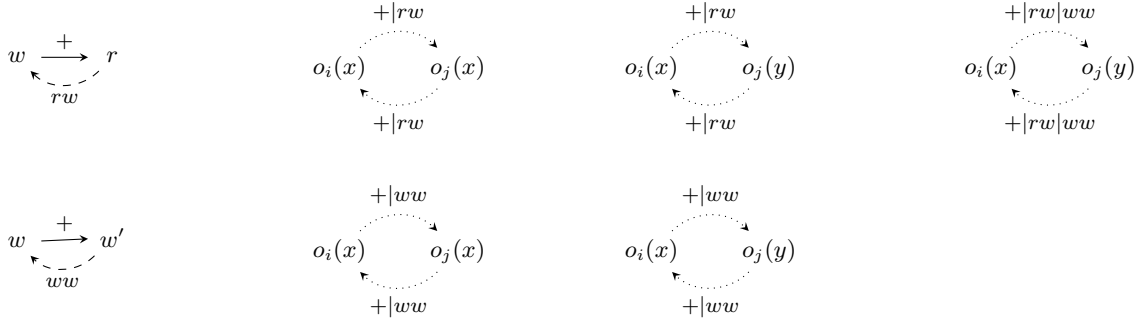
the consistency models that preclude this cycle cannot be implemented in an available way. We show this by proving that no implementation exists that prevents the above anomaly and at the same time can tolerate network partitions.

We use the same proof style as in Gilbert and Lynch [1] and prove by contradiction.

## 7. REFERENCES

- [1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Mar. 1999. Also as Technical Report MIT/LCS/TR-786.
- [2] S. Almeida, J. a. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, 2013.
- [3] G. Chockler, R. Friedman, and R. Vitenberg. Consistency conditions for a corba caching service. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 374–388, London, UK, UK, 2000. Springer-Verlag.
- [4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.
- [5] A. Fekete, D. Liarakis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.
- [6] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011.
- [7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, 2013.
- [8] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, Sept. 2004.
- [9] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–150, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.





**Figure 1: All kinds of possible cycles in consistency graph.**