Operation Ordering in Distributed Systems

Irene Zhang

March 10, 2017

Abstract

Distributed systems can be hard for application programmers to both reason about and use. Because they are *replicated* and *parallel*, programmers cannot simply model them as state machines that execute operations sequentially on a single copy of the system state. Instead, programmers rely on *semantic memory models*, which constrain the system's execution of concurrent operations across replicas of the system state, to help them understand the state of the system after an execution.

In this paper, we cover semantic memory models for many different types of systems, including uniprocessor and multi-processor systems, distributed systems and database systems. We present a *visibility* property that helps readers reason about concurrent and replicated execution histories. Using this visibility property, we define semantic memory models for coherence, consistency and isolation.

1 Introduction

Many important systems today are distributed, including mobile/cloud applications, large-scale analytics platforms and distributed storage systems. Distributed systems are both *parallel* and *replicated*. They consist of distributed nodes concurrently executing operations on local copies of system state.

Parallel and replicated systems are challenging for application programmers to use. Programmers can model simple systems as state machines: these systems apply operations sequentially in the order in which they are received to a single copy of the system state. With this model, the programmer can *control* the order in which the system executes operations and *reason* about the final system state after a single execution. However, parallel systems execute memory operations concurrently and replicated systems execute operations on copies of the system state, making it impossible for programmers to predict the final system state after an execution.

To make parallel and replicated systems easier for application programmers to use, researchers have developed *semantic memory models* for these systems. These models *constrain* the order in which memory operations are applied to the system state, enabling application programmers to again predict and reason about the final state of the system.

In designing a system's semantic memory model, system builders must make a trade-off between usability, implementation complexity and performance. Ideally, systems would provide a strict model that matches the simple state machine model. However, in order to achieve this ideal, strict models require more coordination across replicated objects and concurrent threads, reducing the parallelism and performance of the system.

This trade-off between usability and performance is especially important in distributed systems where coordination is expensive and sometimes impossible. The cost of communicating between nodes is high compared to the cost of executing operations, especially across wide-area networks. As a result, requiring cross-node coordination significantly increases the latency of operations and reduces the system's throughput and scalability. Nodes and networks can both fail, making distributed coordination with some nodes impossible at times. Then, designers must make a difficult choice between violating system guarantees or leaving the system unavailable.

Researchers in the distributed systems, architecture, parallel computing and database communities have been exploring different semantic models and their design trade-offs for decades. In this paper, we review the research literature on semantic models for operation ordering and relate them to their use in distributed systems.

Section 2 gives an overview of the system models used for previous work, and Section 3 summarizes the past work from different research communities. Section 4 introduces terminology that we will use throughout the paper to explore the previous work in more detail. Section 5 introduces a *visibility* property, which we use to reason about parallel and replicated executions and to define semantic memory models. The remainder of the paper describes in detail past memory models. Section 6 covers models developed for single-threaded, unreplicated systems. Section 7 discusses models for replicated, single-threaded systems, while Section 8 describes models for parallel, unreplicated systems. Section 9 covers models for distributed systems that are replicated and parallel. Finally, in Section 10 we present models with support for multi-operation transactions. In Section 11, we summarize the semantic memory models reviewed in the paper and give some insights for future work.

2 System Model(s)

Researchers have developed many semantic memory models for both parallel and replicated systems. There is both overlap and conflict in these past models, leading to confusing terminology and making it difficult to compare memory models from different research communities. A key reason for these differing models is that they largely assume different *system models*. In this section, we give an overview of the system models for past work.

2.1 System State and Operations

We assume a general-purpose system, whose state is composed of a set of *data objects*. These objects can be memory pages, files, data tuples or objects, depending on whether the system is a shared memory system, a file system, a distributed database or a key-value store. The system supports *read and write operations* to data objects (or operations that can be converted to a set of read and write operations). One or more applications interact with the system by invoking operations through an application library.

Given this general model, past system models differ in two respects:

1. **Multi-object vs. single object atomicity.** Some semantic memory models are designed for systems with multiple atomic data objects, while other constrain histories on single atomic data objects. Consistency and isolation models are some examples of the former, while coherence models and linearizability are examples of the latter.

2. **Multi-operation vs. single operation atomic operations.** Some semantic memory models assume atomic groupings of multiple operations, while other were designed around atomic read and write operations. Weak consistency, release consistency and isolation are some models developed for multiple operation ordering, while coherence and consistency models generally constrain single atomic operations.

Semantic memory models designed for multi-operation atomicity assume a system model with support for applications to group operations. These control operations may come in the form of synchronization mechanisms, like *barriers*, or concurrency control mechanisms, like *locks* and *transactions*.

2.2 System Architecture

Past system models have a range of different system architectures. In general, we can assume that the system consists of one or more threads executing operations on one or more copies of the system state. Threads coordinate access to shared data or synchronize replicated data by communicating through the network. The network can be a high-speed processor interconnect, a local-area datacenter network or the wide-area Internet, depending on whether the system is a multi-processor, a distributed system or a wide-area distributed system.

Figure 1 summarizes the system architectures that past semantic memory models have assumed. We list the important features below:

1. **Multi-threaded vs. single threaded architectures.** Semantic memory models developed for parallel architectures focus on

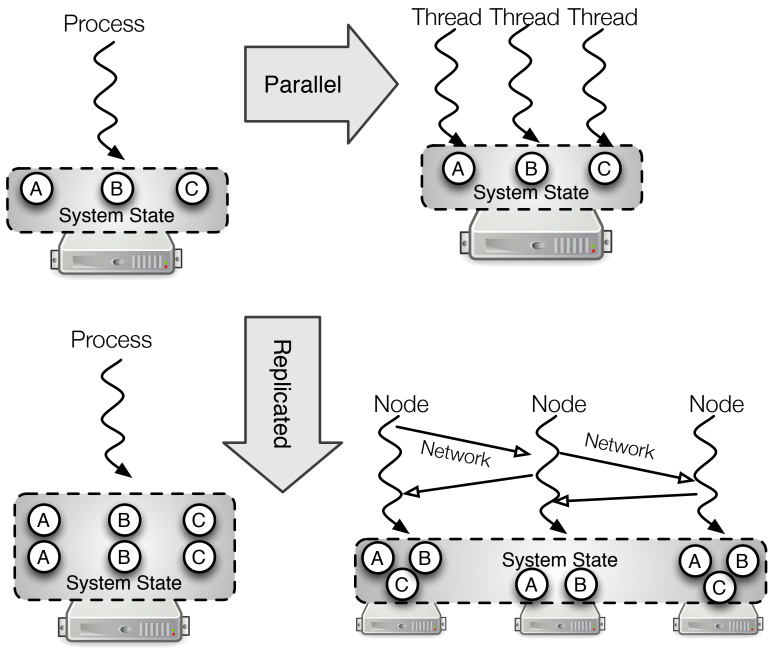

Figure 1: *Architecture of various system models.* Semantic memory models have been developed for a wide-range of system models. Each system model makes different assumptions about the architecture of the system. The simplest system model, shown on the top left, assumes a single thread executing on a single copy of the system state. In the lower left, we show a *replicated* system model, which assumes a single thread executing on multiple copies of the data objects in the system. A *parallel* system, shown on the upper right, assumes multiple threads executing on a single shared copy of the system state. In the lower right, we show a *distributed* system, which assumes multiple processes executing on multiple nodes, each with local copies of data objects. Rather than communicating through shared memory, distributed threads coordinate through the network, which can be a local-area, datacenter network or the wide-area Internet.

constraining the execution of concurrent operations. Program ordering is an example of single-threaded models, while consistency models are designed for multi-threaded systems. It is important to note that distributed systems often assume a single thread per copy of the system state, making them different from shared memory multi-processor systems, although both types of systems use consistency for semantic memory models.

2. **Multi-copy vs single copy architectures.** Semantic memory models designed for replicated systems constrain the execution of operations across multiple copies of the system state. Coherence is an example of single-threaded and single-object models and eventual consistency is an example of replicated models.

3 Overview of Past Semantic Memory Models

In order for programmers to ensure the correct behavior of their application, they must understand how the system will execute read and write operations and apply them to system state. Semantic memory models give application programmers a way to reason about the behavior of systems that might be parallel, replicated or both. Given the overview of various system models in the last section, we now categorize the semantic memory models that have been developed to help application programmers work with each type of system. Table 1 gives a list of previous semantic memory models and the system model features that they assume.

Program ordering models constrain the execution of single atomic operations across multiple objects in the system. These models were developed as processors became more complex, and features like pipelining, multiple issue and write buffer bypassing meant that programmers could no longer count on memory operations to execute in the same order as written in their program. While these models do not deal with replication or concurrency, more complex semantic memory models include or build on these simpler models.

Coherence models constrain the execution of operations to copies of a single data object. With the addition of processor caches, programmers could no longer count on memory operations to execute in the same order on every copy of the system state. These systems were designed for uniprocessor systems, so they do not cover concurrent execution of operations.

Consistency models constrain the execution of concurrent operations across multiple objects in the system. These systems were developed to help programmers reason about the execution of memory operations in multi-processor systems with shared memory. Without a semantic memory model, programmers would have to cope with arbitrary interleavings of operations executing in parallel on a multi-processor.

Many papers in the literature use the terms coherence and consistency interchangeably [17] or define consistency as a type of coherence [34]. This confusion is due to coherence models lacking constraints for concurrency and consistency models (sometimes) lacking constraints for replication. For the purposes of this paper, we define coherence to be a constraint on the history of operations to copies of single data objects and consistency to be a constraint on the history of concurrent operations to all objects in the system.

Linearizability [25] is an exception to this definition, as it is a memory model for concurrent accesses to single data objects. As we will discuss later, linearizability has a unique property that allows it to extend its single data object guarantees across all objects in the system.

Finally, *isolation* constrains the history of concurrent operations, grouped into *transactions*, on multiple objects in the system. Some papers in the literature use consistency and isolation interchangeably [24, 2, 36, 14] because both constrain the history of concurrent objects. Isolation could be considered a special form of consistency, which relies on application programmer-defined grouping of operations (i.e., transactions). In this paper, we define consistency to be a constraint over single atomic read and write operations, and isolation to be a guarantee over atomic transactions.

4 Terminology & Notation

In the following sections, we will explore past semantic memory models in some detail. Because

Table 1: Semantic memory models defined for different system properties.

	Multi-object	Multi-copy	Multi-thread	Multi-op.
Program Ordering	✓			
Coherence		✓		
Shared-memory Consistency	✓		✓	
Linearizability			✓	
Distributed Consistency	✓	✓	✓	
Isolation	✓	✓	✓	✓

these models were developed by researchers in different areas there is sometimes conflicting terminology. In order to enable us to compare models from different research areas, we first introduce a common terminology and set of annotations in this section before diving into the details of different semantic memory models.

4.1 Definitions

We begin with a precise definition of a system's semantic memory model. We can define execution histories in the following way:

Definition 1. ***History.*** *A system's* history *is the sequence of memory operations executed by the system.*

System histories can also be thought of as *execution traces*. We now define a system's semantic memory model in terms of histories:

Definition 2. ***Semantic Memory Model.*** *A system's* semantic memory model *is a constraint on the set of legal execution histories.*

The legal histories are then the *admissible executions* [10] of read and write operations allowed by the semantic memory model. Given a semantic memory model, the system must guarantee that it only produces legal histories.

These constraints can take many forms, which we will discuss in the next section. For example, a simple constraint could be all reads to a data object x return the latest write to x. More complex semantic memory models are often defined as a set of histories that are equivalent to a reference history, where equivalence is defined as follows:

Definition 3. ***Equivalence of Histories.*** *Two histories are* equivalent *if the resulting system state from both executions is identical.*

The reference history often matches the sequence of operations invoked by the application or is derived from it in some way. For example, sequential consistency constrains the system to execution histories that are equivalent to ones that could be executed by a uniprocessor system.

Semantic memory models help application programmers understand the possible final states of the system because they limit the number of executions that the programmer has to reason about. For semantic memory models that use equivalence to a reference history, the application programmer needs only to reason about the reference history. For example, sequential consistency allows the programmer to only reason about executions that can be produced by a uniprocessor system.

4.2 Notation

To represent histories, we use notation derived from Bernstein [11] for memory operations. $r[x]$ and $w[x]$ represent read and write operations to data object x. Given $r[x]$ and $w[x]$ from the application, the system turns them into operations executed by a system thread, process or node. Thus, we must additionally denote the *effect* of read and write operations on the system.

We borrow from static single assignment and assign version IDs to versions of a data object. These IDs do not have any semantic meaning other than identifying written versions of a data object. We label the first write $w[x]$ as $w[x^1]$ and the next as $w[x^2]$ and so on. The next read $r[x]$ that returns the value written by the first write will be annotated as $r[x^1]$. We assume that every data object x is initialized with $x^0 = \varnothing$.

Thus, the following history, H, represents a sequential execution of operations on a unreplicated, single-threaded system:

$$H = w[x^1]w[y^1]r[x^1]w[y^2]r[y^2]$$

This history shows three write operations to data objects, x and y and two reads. The first read in the history, $r[x^1]$ reflects the first version of x, written by $w[x^1]$, while the second read, $r[y^2]$, reflects the second version of y, written by $w[y^2]$. We denote ordering in histories using $\prec$, such as $w[x^1] \prec r[x^1]$.

For parallel systems, for a thread A, we will denote the history of reads and writes executed by a thread of execution, H_A using $w_A[x]$ and $r_A[x]$. For example, take the following parallel history:

$$H_A = w_A[x^1]w_A[y^1]r_A[y^2]$$
$$H_B = r_B[x^1]w_B[y^2]$$

This history shows the same read and write operations executed in parallel. Note that the two threads still share a single copy of each data object x and y. This property of the system that executed this history is reflected in $r_B[x^1]$ reading in thread B the write $r_A[x^1]$ in thread A. Likewise, $r_A[y^2]$ returns the value written by $w_B[y^2]$.

We continue this notation for replicated systems. We denote two copies of x as x_A and x_B. Take the following sequential history on two copies of x and y:

$$H = w[x^1_A]w[y^1_A]r[x^1_B]w[y^2_A]r[y^2_B]$$

Again, this history is similar to our previous two example, but it is executed by a single-threaded system with two copies of x and y. Note that the system uses a coherence protocol to keep the copies synchronized, which is not reflected in the histories. This property can be seen in $r[x^1_B]$ returning a value written to the other copy, $w[x^1_A]$ and $r[y^2_B]$ returning the latest write to copy A, $w[y^2_A]$. Thus, a coherence *model* would constrain what synchronization the coherence protocol must do to ensure that the system produces legal histories.

For distributed systems with replication and parallelism, we cannot assume a total ordering over the written versions of data objects. For example, two threads could write in parallel to local copies of x and y:

$$H_A = w_A[x^1_A]w_A[y^1_A]r_A[y^2_A]$$
$$H_B = r_B[x^1_B]w_B[y^2_B]$$

In this history, it may be uncertain whether $w_A[y^1_A] \prec w_B[y^2_B]$ or $w_A[y^2_A] \prec w_B[y^1_B]$. As a result, for distributed systems, we assume an oracle that provides a total ordering for the versions of each data object. Some replicas may reflect updates in different orders than they were written. For example, a distributed system may produce the following history:

$$H_A = w_A[x^1_A]w_A[y^1_A]r_A[y^2_A]$$
$$H_B = r_B[x^1_B]w_B[y^2_B]r_B[y^2_B]r_B[y^1_B]$$

In this history, the two writes to y were applied in different orders at nodes A and B. While node A applied y^1 and then y^2, node B applied y^2 and then y^1. Note that the labels assigned by our oracle, y^1 and y^2, do not have semantic meaning. Since the write operations executed concurrently at two different nodes, either ordering of the writes could have happened. We simply rely on the oracle to uniquely identify the different values written to y.

5 Visibility

It can be difficult to reason about histories in parallel and replicated systems. Unlike a simple state machine, the order that operations execute rarely leads to an intuitive understanding of the final system state. In this section, we discuss why this makes thinking about semantic memory models difficult and propose a visibility property and constraint that can help with defining semantic memory models.

5.1 Motivating Example

As an example, take the following sequential history of operations to two data objects x and y:

$$H_1 = w[x^1]w[y^1]r[y^1]w[x^2]r[x^1]$$

This history is possible in a simple, uniprocessor system with write buffering: the first write $w[x^1_B]$ finished passing through the buffer before the read $r[x^1_A]$ was issued but not the second write $w[x^2_B]$. As a result, although the processor executed the operations in the order shown in the history, there is actually a much more intuitive, *equivalent* sequential history:

$$H_2 = w[x^1]w[y^1]r[y^1]r[x^1]w[x^2]$$

These histories are equivalent because the last $r[x_A^1]$ in H_1 did not reflect the result of the second write $w[x_B^2]$. As a result, it "appeared" to have executed earlier. Likewise, because $r[y^1]$ and $r[x^1]$ do not conflict, they could have run in either order. In fact, a uniprocessor may have issued them in parallel for more efficient memory access. Thus, the following history is also equivalent to H_1:

$$H_3 = w[x^1]w[y^1]r[x^1]r[y^1]w[x^2]$$

As apparent from this example, even in this uniprocessor system, the real-time order of execution does not necessarily reflect the equivalent histories. In a distributed system, it is often impossible to discern the real-time order of operations without coordination. Instead, we argue that it is much more useful to reason about equivalent histories as a way to understand the final state of the system.

5.2 The Visibility Property

We define a *visibility* property that better captures "ordering" in system histories. Intuitively, an operation α is *visible* to an operation β if β executes on a copy of system state where α has been applied either by: (1) executing α before β on the same copy of the system state or (2) executing a memory update for α to the node's local copy of system state before executing β. For example, $w[x^1]$ is visible to $r[x^1]$ regardless of what order the system actually executed the operation and where the operations executed.

Visibility implies that there exists an equivalent history where $\alpha \prec \beta$, regardless of the actual execution history. As a result, visibility is a much more useful way to reason about ordering in systems than actual execution ordering.

Now, we give a more precise definition of *visibility*:

Definition 4. ***Visibility.*** *We define a read $r[x]$ to be visible to an operation α where α cannot affect the value returned from $r[x]$. We define a write $w[x]$ to be visible to an operation β where the effect of β changes if the written value of $w[x]$ changes.*

We use $\beta \lhd \alpha$ to denote visibility. We can define an operation, α's *visibility set* as the set of operations, β, where $\beta \lhd \alpha$.

Visibility is related to dependency graphs. An operation's visibility set can also be thought of as the set of operations that it depends on. Thus, given a dependency graph G and an operation α, every operation in G that is reachable from α is visible to α.

Visibility is also related to Gharachorloo's [21] "executes with respect to" relationship: an operation α performed by processor P_i is considered *performed with respect to* processor P_k at a point in time when α is visible to all subsequent operations executed by P_k.

We additionally define a *global visibility* property that encapsulates atomicity for concurrent and replicated operations:

Definition 5. ***Global Visibility.*** *We define an operation α to be* globally visible *to an operation β, if, for* all *operations γ where β is visible to γ, α is visible to γ.*

Global visibility enforces the same ordering guarantee for two operations across both threads and replicas, ensuring that the operations appear atomic to each other. More importantly, global visibility ensures that there exists an equivalent single-threaded, unreplicated history where $\alpha \prec \beta$. We use $\alpha \blacktriangleleft \beta$ to denote global visibility.

Visibility is useful for constraining the set of legal histories allowed by a semantic memory model. We can specify the legal histories allowed by a semantic memory model in terms of the constraints placed on the visibility of different operations. For example, a model that requires that reads reflect the most recent write could be described as requiring all writes to be visible to reads. In the remainder of this paper, we will define a wide range of memory models based on *visibility constraints*.

In a distributed system, visibility constraints directly give us insight into the performance and availability of different semantic memory models because *visibility implies coordination* . Without coordination, visibility is not possible between operations executing on different nodes to shared data objects. As a result, semantic models that place more visibility constraints on histories require more expensive distributed coordination than more relaxed semantic models with fewer constraints. Further, models that have strict visibility constraints cannot provide *availability* when coordination is impossible.

5.3 Strong vs. Weak Models and Visibility Anomalies

Using visibility, we can categorize past semantic memory models into *strong* or *weak* models. Strong semantic memory models use our ideal state machine as the reference history, while weak semantic memory models reveal inconsistencies that stray from the ideal model. In order to mirror the state machine model, strong semantic memory models limit the histories that the system can produce to *sequential histories.* Informally, sequential histories are ones where there exists a single serial ordering of all operations. Using visibility, we can more precisely define sequential histories:

Definition 6. ***Sequential Histories.*** *The set of* sequential histories *is all histories where, for all operations α and β in H, either $\alpha \blacktriangleleft \beta$ or $\beta \blacktriangleleft \alpha$.*

Sequential histories provide the "appearance" of atomicity for each operation across replicas and a sequential execution of operations across threads or processes. However, they do not have to actually execute sequentially, while, at the same time, some sequential executions are not sequential histories. Instead, sequential histories are *equivalent* to a history that could be produced by a simple state machine As a result, they simplify reasoning about parallelism and replication.

Some semantic memory models (i.e., linearizability, strict serializability) go a step further and give a more "real-time" guarantee for the sequential ordering of operations.[1]. They require that the order in which the system orders operations in the sequential ordering reflect that which an *external observer* might expect. Sequential histories do not necessarily have this feature as they only requires a *single* serial ordering of operations but not a particular serial ordering. We define these "real-time" histories as *linearizable* histories, according to the definition given by Herlihy [25]:

Definition 7. ***Linearizable.*** *A sequential history is further* linearizable *if, for every operation α that returns before β begins, $\alpha \blacktriangleleft \beta$.*

[1] This is sometimes termed *external consistency* vs. an internally consistent, non-"real-time" guarantee but we will refrain from this terminology because the same terms are used to define something else in database systems and isolation models

In contrast, weaker semantic memory models allow the system to produce histories that are not sequential. We can categorize weak semantic memory models based on the kind of *visibility anomalies* that they reveal to the application programmer; however, we must also take into account the system model that the weak memory model assumes because some anomalies are only possible under certain system models. In the next section, we explore both strong and weak semantic memory models for different system models, using visibility to precisely define each system model and semantic memory model.

6 Program Ordering Models

In the following sections, we explore each category of semantic memory models in detail. We begin each section with a short, precise description of the assumptions of our system model.

We begin with memory models for non-concurrent, non-replicated systems, which are generally described as *program ordering* models. In these systems, visibility implies global visibility because there is no replication or parallelism. While they are simple, these models will be important for using as reference histories when constraining the histories of replicated, concurrent systems.

6.1 System Model

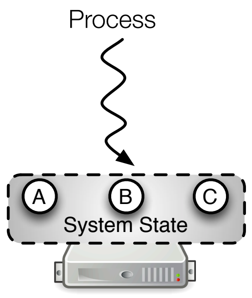

Figure 2: *Single-threaded, Unreplicated System Architecture.*

We assume a single-threaded, unreplicated system, similar to a state machine. Figure 2 shows the architecture. The system executes read and write operations sequentially on a set of data objects. We define the system model using the following visibility rules:

1. There is a *single* ordering of operations, H, that reflects the order in which the system executed operations to a set of data objects $\{x, y, \ldots\}$.
2. If operations $\alpha[x] \prec \beta[y]$ (to either the same or different objects x and y) in H, then $\alpha[x] \lhd \beta[y]$.

Our system model ensures this visibility rule for all histories that a single-threaded, unreplicated system could *possibly* produce. Any semantic memory model must be considered as a set of additional constraints over this set of possible histories.

6.2 Sequential Program Ordering

The simplest semantic memory model for a single-threaded, unreplicated system matches a state machine. We define *program order* to be the order in which the application invokes operations on the system. The *sequential program ordering* model allows programmers to assume that all memory operations will execute one at a time in the order in program order. We define the model as a set of constraints as follows:

Definition 8. ***Strict program order*** *requires that for any operation α that appears before operation β in the application program, α must be globally visible to β.*

As Adve [1] notes, this requirement does not limit the processor to only executing one memory operation at a time. The system is still free to re-order unrelated operations (e.g., operations to different memory locations, operations without control flow, etc.). The system only needs to produce a history that is equivalent to the *reference history*, which in this case is a history of the memory operations in program order. This program ordering model is often used to describe and constrain more complex models, like consistency models.

6.3 Relaxed Program Ordering

There are many ways that the sequential program ordering model can be relaxed. These relaxations can improve performance by allowing processor and compiler optimizations that reorder memory operations. As a result, these models generally provide operations to allow applications to control reordering, like *memory barriers*:

Definition 9. ***Memory barriers*** *require that for any operations α that appears before a barrier σ in program order, then for all operations β, where $\sigma \lhd \beta$, then α must be globally visible to β.*

7 Coherence Models

Next, we look at replicated memory models for systems without parallel execution. These *coherence* models constrain histories of operations to *copies* of single data objects but do not constrain histories of operations over multiple objects. After defining our system model, we will begin with the strongest coherence models and move to weaker models.

7.1 System Model

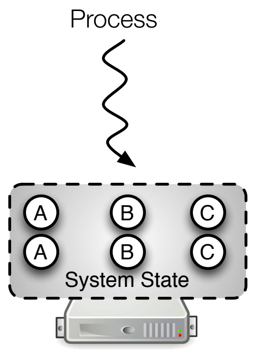

Figure 3: *Single-threaded, Replicated System Architecture.*

We assume a single-threaded, replicated system, such as a uniprocessor with caching, as shown in Figure 3. The system history consists of sequential read and write operations to copies of a single data object. We define the system model using the following rules for read and write operations:

1. There is a *single* ordering of operations, H, that reflects the order in which the system

executed operations to *copies* of a single data object $\{x_A, x_B, \ldots\}$.

2. If operations to a single copy of x, $\alpha[x_A] \prec \beta[x_A] \in H$, then $\alpha[x_A] \lhd \beta[x_A]$.

3. If $\alpha[x_A] \prec \beta[x_B] \in H$ *and* the system synchronized x_A and x_B after $\alpha[x_A]$ and before $\beta[x_B]$, then $\alpha[x_A] \lhd \beta[x_B]$

These visibility rules state the assumptions of our single-threaded, replicated system model. The first rule states that the system has a single-thread of execution. The second rule states that, for any two operations to the same replica x_A, the earlier operation is visible to the later one. The last rule states that, for any two operations to different replicas, an earlier operation on one replica can only be visible to a later operation on another replica if the system synchronized the two replicas in between the operations. Again, these rules simply state the assumptions about the system model, but they constrain the set of *possible* histories that a system with this model could produce.

Within these visibility rules, the semantic memory model constrains when the system *must* synchronize replicas to meet the requirements of the memory model. For example, a memory model will dictate, if $w[x_A^i]$ is followed by $r[x_B^j]$ in H without intervening writes, whether the system must *ensure* that $w[x_A^i]$ is be visible to $r[x_B^j]$.

7.2 Strict Coherence Model

Li [32] and Bennett [8] describe memory as *strictly coherent* if the value returned by a read is always the same as the value written by the most recent write operation to the same object. Coherence models were developed for uniprocessor systems, so the most recent write can be defined as latest write in H (i.e., the last write executed by the processor). Strict coherence can be more precisely defined as:

Definition 10. ***Strict coherence*** *requires that for any two operations $\alpha[x_A]$ and $\beta[x_B]$, if $\alpha[x_A] \prec \beta[x_B] \in H$, then $\alpha[x_A]$ must be globally visible to $\beta[x_B]$ in all legal histories.*

This model constrains the replicated uniprocessor system to a reference history that is identical to the processor execution ordering. For this reason, in a strict coherence model, the application essentially cannot detect that there are multiple copies of data objects. Many of the early distributed shared memory systems provided strict coherence, including hardware-based systems like, Memnet [16], as well as many software-based systems, like Ivy [32], Mermaid [46] and Mirage [20].

Because coherence only constrains histories on a single data object, the strict coherence model does not require systems to immediately update all copies of data objects. If $\alpha[x_A]$ and $\beta[y_B]$ are to different data objects, then the system can update the copies of after executing $\beta[y]$ and still have an equivalent history. However, strict coherence models require a more coordination across replicas and allow less parallelism than weaker models. This limitation becomes a problem in multi-processor systems, so researchers have developed and implemented many weaker alternatives with better performance.

7.3 Weak Coherence Models

Bennett [8] defines a memory model for multi-processor systems to be *loosely coherent* if the value returned by a read is the same value written by a write operation to the same object that *could* have preceded the read in some legal schedule of the executing threads. This model allows better performance, while still obeying the memory consistency model of the multi-processor system.

Definition 11. ***Loose coherence*** *requires that for any two operations $\alpha[x]$ and $\beta[x]$, if $\alpha \prec \beta$ in a legal history based on the consistency model, then $\alpha \blacktriangleleft \beta$ in all legal histories.*

This coherence model takes into account the fact that, in multi-processor systems, it is often difficult to figure out which was the most "recent" write operation. But it still respects the legal schedule of executing threads as constrained by the consistency model. This model provides better performance than strict coherence for systems with weak consistency models; for example, Bennett's Munin [8] system, which uses release consistency.

Loose coherence is an important example of the interaction between semantic memory models when they are layered together. In a multi-processor system with caching, a coherence model is not sufficient because the application programmer must also have a consistency model to constrain the execution of concurrent operations on the multi-processor. As Adve describes [1], a memory consistency model places a upper and lower bounds on the coherence model. Thus, if the consistency model is relaxed enough then a strict coherence model is not needed.

A final even weaker coherence model is proposed by Adve [1]. They define memory as *cache coherent* if the values returned to reads are in the same order as written and reads eventually return all writes.

Definition 12. ***Cache coherence*** *requires that, for any two writes,* $w[x^i] \prec w[x^j] \in H$, $w[x^i] \blacktriangleleft w[x^j]$ *and all* $r[x^i] \blacktriangleleft r[x^j]$ *and eventually all* $r[x]$ *must return* x^j*.*

This model only guarantees a serial ordering of reads and writes to each data object, where the writes and reads separately execute in program order, but can be interleaved in any way. The eventual guarantee is also very weak because it does not put an upper bound on when writes will be globally propagated.

8 Shared Memory Consistency Models

Consistency models were developed for systems that support concurrent accesses. There exist a wide range of consistency models proposed by database, architecture and systems researchers. We divide these models into ones developed for multi-processors, which we discuss in this section, and ones for distributed systems, which we cover in the next section. The key difference between the two is that multi-processor consistency models assume concurrently executing *threads* that access *shared* memory, while distributed system consistency models assume processes, each with local copies of system state, and no shared memory.

8.1 System Model

We assume a multi-threaded, unreplicated system, as shown in Figure 4.

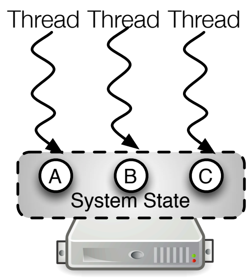

Figure 4: *Single-threaded, Unreplicated System Architecture.*

The system history consists of multiple threads of sequential read and write operations to a shared set of data objects. We define the system model using the following rules for read and write operations:

1. There is a set of sequential orderings of operations, $\{H_A, H_B, \ldots\}$, that reflects the order in which the system executed operations in each thread $\{A, B, \ldots\}$ to a shared set of data objects, $\{x, y, \ldots\}$.
2. If $\alpha_A[x] \prec \beta_A[y] \in H_A$, then $\alpha_A[x] \lhd \beta_A[y]$ (where x and y may be the same or different data objects).
3. For two operations $\alpha_A[x]$ and $\beta_B[y]$ in different threads A and B, if the system updated shared memory after $\alpha_A[x]$ and before $\beta_B[x]$, then $\alpha_A[x] \lhd \beta_B[y]$ (where x and y may be the same or different data objects).

Again, these rules state the visibility assumptions of our multi-threaded, shared memory system model. The first rule states that each thread in the system has a sequential ordering of operations to a set of multiple data objects. The next rule states that earlier operations are visible to later operations executed in the same thread. And, the last rule states that operations from one thread can only be visible to operations from another thread if the system coordinates across threads through shared memory.

Within these visibility rules, consistency models constrain when the system must coordinate across threads. For example, a memory model will dictate when operations from one thread must block in order to ensure the thread sees operations from another thread.

8.2 Strong Consistency Models

Strong consistency models only allow sequential histories. There are two primary models that could be considered strong consistency: sequential consistency and linearizability. There are two key differences between the two modes. First, linearizability constrains the system to *linearizable*, not *sequential* histories. Second, sequential consistency is an *global* constraint, which means that it constrains the history of all operations in the system, while linearizability is a *local* constrain, which means that it constrains the history of all operations to each atomic object in the system.

8.2.1 Sequential Consistency

Lamport [29] first formally defined sequential consistency as two requirements: (1) the result of the execution is equivalent to executing all of the operations across threads in a sequential order and (2) the operations of any individual thread are executed in strict program order (as we defined in Section 6). We can also define sequential consistency using constraints on histories as follows:

Definition 13. ***Sequential consistency*** *requires that, (1) for all threads, A, H_A must be in strict program order, and, (2) if $\alpha_A \prec \beta_A \in H_A$, then $\alpha_A \blacktriangleleft \beta_A$, and, (3) for any α_A and β_B in different threads, either $\alpha_A \blacktriangleleft \beta_B$ or $\beta_B \blacktriangleleft \alpha_A$.*

Many systems provide sequential consistency because it is easy to reason about and is close to the ideal state machine model. Intuitively, the reference history for sequential consistency is any interleaving of the threads in program order. However, it does not make any guarantees across operations or threads. As a result, it is hard to reason about *what* sequential ordering two operations in different threads may end up in. For example, given two threads with histories $H_A = w[x_A]r[x_A]$ and $H_B = w[x_B]r[x_B]$, any ordering of these operations (i.e., $w[x_A]w[x_B]r[x_B]r[x_A]$, etc.) is legal under sequential consistency, regardless of which operations finish first.

8.2.2 Linearizability

Linearizability provides a sequential ordering of concurrent operations that better matches real-time guarantees. Herlihy [25] defined linearizability as a constraint on histories of atomic operations, which each consist of some number of read and write memory operations, to shared data objects. A history is *linearizable* if it preserves sequential program ordering in each thread and every operation is ordered after every operation that returned before it started and before every operation that starts after it returns. We can define linearizability using constraints as follows:

Definition 14. ***Linearizability*** *requires that the history be sequentially consistent and, for any operation $\alpha[x]$ that returns before $\beta[x]$ begins, $\alpha[x] \blacktriangleleft \beta[x]$.*

Unlike sequential consistency, linearizability does not constrain operations across objects. However, assuming that $r[x]$ must return before being written to $w[y]$, linearizability still enforces an ordering across x and y because $r[x]$ will be ordered before $w[x]$. Due to this property, linearizability is *local*: if every object in a system is linearizable, then the whole system is linearizable.

8.3 Causal Consistency Models

The next strongest group of consistency models are *causal consistency models.* Unlike strong consistency models, which require that all operations see all previous operations, causal models only require that operation see all previous operation on which it has a causal dependency.

The ways of finding causal dependencies between operations differ. Some systems rely on application programmers or a compiler to generate the causal dependencies between operations. Traditionally, causal consistency has been defined using the causal dependency relation, $\rightsquigarrow$, which relies on program ordering and read-write relationships. The definition from Lloyd [33], which is based on an older definition from Ahamad [4] and an even older one from Herlihy [25], states that potentially $\alpha \rightsquigarrow \beta$ if and only if one of the following holds:

- $\alpha \prec \beta$ in program ordering
- all $r[x^i]$ for any $w[x^i]$
- $\alpha \rightsquigarrow \gamma$ and $\gamma \rightsquigarrow \beta$

Causal dependencies are transitive and form acyclic graphs. Causal consistency uses these dependencies to impose constraints on legal histories as follows:

Definition 15. ***Causal consistency*** *requires that, (1) for all threads, A, H_A must be in program order, and (2) for any operations in two threads A and B, if a causal dependency exists between $\alpha_A \rightsquigarrow \beta_B$, then α_A must be globally visible to β_B.*

The first rule ensures strict program ordering within each thread, while the second rule ensures causal dependencies are preserved across threads. Causal consistency does not impose a single serial ordering on concurrent operations, so conflicts can occur, which the application must resolve.

8.4 Weak Consistency Models

The first weak consistency models were defined for shared memory multi-processors. In recent years, they have become an important feature of high-performance replicated storage. We cover these roughly in order from strongest to weakest.

Before we cover weak consistency models, we catalog the set of requirements that must be satisfied for sequential consistency, which weak consistency models relax. Reframing the relaxations summarized by Adve [1], we have the following visibility constraints. We add a P0 for strict program ordering within threads; however, no weak models violate that property because it is relatively cheap to provide.

P0 If, for some thread A, $\alpha_A \prec \beta_A$ in program ordering, then $\alpha_A \prec \beta_A \in H_A$.

P1 If, for some thread A, $w_A[x] \prec r_A[y] \in H_A$, then $w_A[x]$ must be globally visible to $r_A[y]$.

P2 If, for some thread A, $w_A[x] \prec w_A[y] \in H_A$, then $w_A[x]$ must be globally visible to $w_A[y]$. order, then $w[x]$ must be globally visible to $w[y]$.

P3 If, for some thread A, $r_A[x] \prec r_A[y] \in H_A$, then $r_A[x]$ must be globally visible to $r_A[y]$.

P4 If, for some thread A, $r_A[x] \prec w_A[y] \in H_A$, then $r_A[x]$ must be globally visible to $w_A[y]$.

P5 If, for some thread A, $w_A[x] \lhd r_A[x]$, then $w_A[x]$ must be globally visible to $r_A[x]$.

P6 If, for any two threads A and B, $w_A[x] \lhd r_B[x]$, then $w_A[x]$ must be globally visible to $r_B[x]$.

P1-P4 simply state the program order requirements in sequential consistency with respect to operations on different objects. These constraints all pertain to different data objects x and y because if $x = y$, then the constraints on their visibility would be covered by the coherence model. P5-P6 state constraints on how atomic writes appear to other threads. P5 states that once a write becomes visible in a thread, it must be visible in all threads. P6 allows threads to see their own writes early, but once another thread has seen the write, all threads must see it. Now, we give definitions of common weak consistency models in processor architectures using these properties.

Definition 16. ***Total Store Ordering*** *requires P0, P2-P4 and P6.*

Total store ordering relaxes the program ordering requirement between writes followed by a read in the same thread, either to different objects or the same object. This relaxation can lead to the following inconsistent behavior. Take the following history of threads A and B:

$$H_A = w_A[x^1]w_A[x^2]r_A[y^1] \tag{1}$$

$$H_B = w_B[y^1]w_B[y^2]r_B[x^1] \tag{2}$$

There exists no serializable history for these threads; however, total store ordering was not violated. In particular, eliminating requirement $P1$ means that thread A was able to read y^1 after writing x^2 and thread B was able to read x^1 after writing y^2. The only way to achieve a serializable ordering with this history is to relax the program order and order $r_B[x^1]$ before $w_B[y^2]$.

Definition 17. ***Processor Consistency*** *requires P0, P2-P4.*

Processor consistency further relaxes total store ordering for the atomicity of writes across processors. Given the following history of threads A, B and C,

$$H_A = w_A[x^1]w_A[x^2] \tag{3}$$

$$H_B = w_B[y^1]r_B[x^2]w_B[y^2] \tag{4}$$

$$H_C = r_C[y^2]r_C[x^1] \tag{5}$$

We are not able to find an equivalent total store order history for this history because it violates P6; however it does satisfy processor consistency. In order to achieve a serializable ordering, we could have to relax the order in which C sees $w_A[x^2]$, to enable us to order it before $w_B[y^2]$ and $r_C[y^2]$, which are both visible to $r_C[x^1]$.

Definition 18. ***Partial Store Ordering*** *requires P0, P3-P4 and P6.*

Partial store order further relaxes the program ordering requirement between writes to different objects. This relaxation leads to inconsistent behavior as follows:

$$H_A = w_A[x^1]w_A[x^2]w_A[y^1]w_A[y^2] \tag{6}$$

$$H_B = r_B[y^2]r_B[x^1] \tag{7}$$

In this case, B executes $r_B[y^2]$, seeing the later executed write to y in thread A but does not see the later update to x, which was executed earlier in thread A. Now that we have defined some weak consistency models and their properties, we will next look at weak consistency models with support for application programmers to control their weak properties.

8.4.1 Synchronization-based weak consistency models

All of the previous weak consistency models support atomic read-update-write operations to allow programmers sequential behavior for a single pair of read and write operations. Some also support other synchronization mechanisms like barriers and fences to allow programmers to explicitly ensure visibility in certain cases.

Weak ordering [1], also called *weak consistency* [21], allows programmers to label certain operations as *synchronization* operations. These operations to allow programmers to explicitly constrain the legal histories in the system in the following way:

Definition 19. ***Weak Ordering*** *requires that, for all* synchronization operations σ_A *and* ω_B *invoked in two threads A and B: (1)* σ_A *must be globally visible to* ω_B *or* ω_B *is globally visible to* σ_A*, and, (2) if an operation* $\alpha_A \prec \sigma_A \in H_A$*, then* $\alpha_A \blacktriangleleft \sigma_A$*, and (3) if* $\sigma_A \lhd \beta_B$*, then* $\alpha_A \blacktriangleleft \beta_B$*.*

The first condition requires that all synchronization operations are sequentially ordered. The next condition requires that normal operations are not re-ordered with respect to synchronization operations in a single thread and the last condition ensures that normal operations are not re-ordered with respect to synchronization operations across threads. Weak consistency depends heavily on the application programmer or compiler to correctly label synchronization operations to ensure any consistency guarantees that the application may require.

Release consistency is an extension of weak consistency designed for locking-based synchronization. It further categorizes synchronization operations into *acquires* and *releases.* Again, release consistency depends on the application or compiler correctly labeling the operations in the program to ensure consistency.

Definition 20. ***Release Consistency*** *requires that, (1) given an acquire operation in thread A,* σ_A*, for all operations* $\sigma_A \prec \alpha_A \in H_A$*,* $\sigma \blacktriangleleft \alpha_A$*, and for all operations* β_A*, before a release operation* ϕ_A*, such that* $\beta_A \prec \phi_A \in H_A$*,* $\beta_A \blacktriangleleft \phi_A$*, and (3) the history of acquire and release operations must be either sequential or processor consistent.*

Release consistency comes in two variants, one where synchronization operations must be sequential and another where they must be processor consistent. Release consistency also places strict requirements on the application programmer to correctly label operations and protect them with acquire and release synchronization operations. Assuming a correctly-labeled program, release consistency can provide much better performance than sequential consistency but still produce a sequential history. This property is due to understanding that operations between acquire and release operations are *data-race-free* and eliminating the need to coordinate them.

9 Replicated Consistency Models

Next, we tackle consistency models for distributed systems *without* shared memory. Their system model assumes concurrent processes executing operations on *local* copies of system state and communicating to *synchronize* those copies. Because these systems are both replicated and parallel, they can produce histories that only parallel or only replicated system cannot. In particular, parallel processes on different nodes can apply the same operations in different orders to their local replica, which cannot happen in a shared memory system, leading to *divergent* histories at each node.

When histories at different nodes diverge, *conflicts* can arise. We define a conflict as follows:

Definition 21. ***Conflict.*** *We define two operations α and β to be* conflicting *if $\alpha \lhd \beta$ and $\beta \lhd \alpha$ on different threads executing on local copies of the system state.*

Conflicting operations have to be *resolved* by the application or the system; otherwise, the histories at each node will continue to diverge. Conflicts can be resolved using different strategies from simple ones, like "last-writer-wins" [44] to more complex application-specific strategies like merging versions of a document. Conflict resolution can lead to anomalous behavior because it usually requires some nodes to undo or eliminate operations that they have already executed, which can cause them to "appear" as if they are moving back in time relative to other replicas.

In comparison, shared memory, parallel systems cannot have conflicts because they always have a last-writer-wins policy for the shared copy of the system state, where every write to shared memory overwrites previous updates.

9.1 System Model

We assume a multi-threaded, replicated distributed system, where each node concurrently executes operations on a local copy of the system state. Figure 5 shows the assumed architecture.

The system history consists of multiple histories of read and write operations to a replicated set of data objects distributed across nodes. For simplicity, we assume that all operations at a node A are to its local copies of the data objects. We define the following visibility rules for the distributed system model:

1. There is a set of sequential orderings of operations, $\{H_A, H_B, \ldots\}$, that reflects the order in which the system executed operations at each node $\{A, B, \ldots\}$ to local copies of shared data objects, $\{\{x_A, y_A, \ldots\}, \{x_B, y_B, \ldots\}, \ldots\}$.

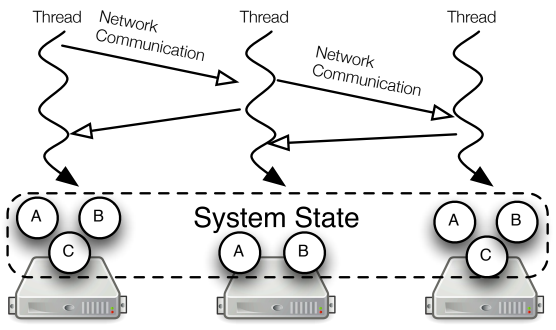

Figure 5: *Distributed System Architecture.*

2. If $\alpha_A[x_A] \prec \beta_A[y_A] \in H_A$, then $\alpha_A[x_A] \lhd \beta_A[y_A]$.
3. For two operations at different nodes, $\alpha_A[x_A]$ and $\beta_B[y_B]$, if the system coordinates across nodes A and B and synchronizes copies of x and y after $\alpha_A[x_A]$ and before $\beta_B[y_B]$, then $\alpha_A[x_A] \lhd \beta_B[y_B]$.

These rules state the visibility properties of our distributed system model. The first rule states that there is a single thread of execution at each node that executes on local copies of shared system state. Rule 2 states that any local operations are visible to later operations at the same node. Rule 3 states that, in order for an operation at another node to be visible, the two nodes must have coordinated to make that possible.

Distributed consistency models constrain histories that distributed systems can produce. Rule 3 implies that every time a semantic memory model requires that operations from one node be visible at other nodes, there must be distributed coordination.

9.2 Strong Replicated Consistency Models

Strong consistency models for replication mask the replication in the system from application programmers. The *one-copy serializability* [5] model is the replicated equivalent of sequential consistency. It combines a sequential consistency and strict coherence. Because our global visibility property covers both threads and replicas, we define one-copy serializability in exactly the same way as sequential consistency:

Definition 22. ***One-copy serializability*** *requires that, (1) for all nodes A, H_A is in program order, and, (2) if $\alpha_A \prec \beta_A \in H_A$, then $\alpha_A \blacktriangleleft \beta_A$, and, (3) for any α_A and β_B in different threads, either $\alpha_A \blacktriangleleft \beta_B$ or $\beta_B \blacktriangleleft \alpha_A$.*

Primary-backup systems [5] provide one-copy serializability as long as all operations – reads and writes – are processed by the primary. In that case, the backups are strictly coherent and the primary enforces sequential consistency.

Linearizability can also serve as a more real-time memory model for replicated and parallel systems. We can use the same definition of linearizability as before:

Definition 23. ***1-copy Linearizability*** *requires that the history be 1-copy serializable and, for any operation $\alpha[x]$ that returns before $\beta[x]$ begins, $\alpha[x] \blacktriangleleft \beta[x]$.*

Many popular quorum-based state machine replication protocols, like Viewstamped Replication [35] and Multi-Paxos [30], provide linearizability because the group only processes one operation at a time. In this case, the application is the atomic object and concurrent operations are always executed in the order in which the replica group receives them.

9.3 Causal Consistency Models

Weaker models for consistency in multi-threaded and replicated systems require the application to perform some form of conflict resolution. Lloyd [33] defines *causal+ consistency*, a variant of causal consistency, that requires *convergent*, or deterministic, conflict handling. More specifically, in addition to causal consistency, causal+ consistency requires that every thread resolves conflicts in the same way without coordination.

Definition 24. ***Causal+ consistency*** *requires that if a causal dependency exists between $\alpha \rightsquigarrow \beta$, then $\alpha \blacktriangleleft \beta$, and, for any concurrent operations, γ and σ, then there is a deterministic function to decide whether γ should be globally visible to σ or σ should be globally visible to γ (i.e., "last-writer-wins" [44]).*

9.4 Eventual Consistency Models

Strong consistency models for replication require coordination between replicas for every operation. This coordination is expensive if the network is slow, and it can keep the system from processing operations when some replicas cannot communicate. As a result, weaker consistency models were developed for distributed systems for better performance and availability.

Unlike strong consistency models, *eventual consistency* models can be implemented in an optimistic or lazy way. As long as the system ensures that all nodes eventually apply all write operations, which is often called the *anti-entropy* [42] property, nodes do not have to coordinate during normal execution of memory operations. Intuitively, eventual consistency models provide a guarantee that, if the system stops processing operations, it will eventually converge to a sequentially consistent state. Until convergence, application can see inconsistencies between concurrently executing operations or across replicas; however these conflicts do not persists as in weak consistency models.

Eventual consistency models are distinguished based on what inconsistencies an application may see at a given node. Different systems implementing the same model may give applications different mechanisms for dealing with these inconsistencies as well. In this section, we review some different guarantees and conflict resolution mechanisms.

9.4.1 Session Ordering

Session ordering guarantees were presented by Terry [42] as a way to provide some weak consistency guarantees to a single client within the environment of an application-defined *session.* Session could be considered similar to long-running transactions; however, they do not provide isolation or atomicity guarantees.

Terry [42] presents four types of session guarantees: Read Your Writes, Monotonic Reads, Writes Follow Reads and Monotonic Writes. We will define these guarantees using our visibility property. We define *session order* as program order within a session.

Read Your Writes ensures that applications do not miss any updates that they previously invoked in the same session. Eventual consistency is weak enough that this may happen in some cases without the guarantee. We define Read Your Writes as follows:

Definition 25. ***Read Your Writes*** *requires that if $w_A[x_A^i] \prec r_B[y_B^j]$ in session order, then $w_A[x_A^i]$ must be globally visible to $r_B[y_B^j]$ (where x and j can be the same or different objects).*

This rule states that, within a session, later reads should always reflect earlier writes, even if they do not go to the same node. In practice, this model is often enforced with *stickiness*, where clients always send operations to the same node within a session. Then, the system only has to enforce that if $w_A[x_A^i] \prec r_A[x_A^j]$ in session order, then $w_A[x_A^i] \lhd r_A[x_A^j]$. We extend Read Your Writes to enforce the visibility property globally. Since eventual consistency systems generally do not propagate reads, this should be a natural extension.

The Monotonic Reads property keeps applications from seeing updates that later seem to "disappear." We define it as follows:

Definition 26. ***Monotonic Reads*** *requires that if $r_A[x_A^i] \prec r_B[y_B^j]$ in session order (where x and y could be the same data object or not), then for all writes, $w_A \lhd r_A[x_A^i]$, then w_A must be globally visible to $r_B[y_B^j]$.*

This states that any subsequent reads in session order must reflect at least any writes reflected in earlier reads, ensuring the set of writes reflected in the reads in any session is always monotonically increasing.

The Writes Follow Reads property guarantees that all writes are ordered and reads are ordered with respect to the writes. It is similar to the cache coherence model introduced in Section 7 but applies to multiple objects.

Definition 27. ***Writes Follow Reads*** *requires that if $r_A[x_A^i] \prec w_B[y_B^j]$ in session order, then, for any write $w_A \lhd r_A[x_A^i]$, w_A must be globally visible to $w_B[y_B^j]$ (where x and y can be the same or different objects).*

Unlike, the previous two properties, this property applies to global ordering rather than simply ordering within the session. This ensures that if the later write depended on the earlier write, they will always be applied in the same order.

Similarly, Monotonic Writes ensures that the write ordering within a session is preserved globally.

Definition 28. ***Monotonic Writes*** *requires that if $w_A[x_A^i] \prec w_B[y_B^j]$ in session order, then $w_A[x_A^i] \blacktriangleleft w_B[y_B^j]$ (where x and y can be the same or different objects).*

These four properties were chosen because they provide useful guarantees on top of eventual consistency for application programmers and require minimal modification to eventual consistency systems to implement.

9.4.2 Lazy Replication

Lazy Replication [28] provides an alternative model for imposing constraints on eventual consistency models. It introduces three categories of operations: causal, forced and immediate. We define each below.

Definition 29. ***Causal operations*** *require that for two causal operations α and β, if $\alpha \rightsquigarrow \beta$, then $\alpha \blacktriangleleft \beta$.*

Causal operations are the weakest category as they do not have any global ordering; thus, they can be executed in different orders on different replicas.

Definition 30. ***Forced operations*** *require that for two forced operations α and β, either $\alpha \blacktriangleleft \beta$ or $\beta \blacktriangleleft \alpha$.*

Note that forced operations are sequential, so if all operations were forced, the system would be sequentially consistent.

Definition 31. ***Immediate operations*** *require that for an immediate operation α and any other operation β, either $\alpha \blacktriangleleft \beta$ or $\beta \blacktriangleleft \alpha$.*

Immediate operations are sequential with respect to all other operations. In addition, Ladin [28] notes that they are externally consistent in that they reflect real-time ordering, similar to linearizability [25].

9.4.3 Conflict Resolution

In the previous section, we covered ordering constraints for eventual consistency systems. These ordering constraints are quite weak, so conflicts can arise. Next, we cover techniques from eventual consistency models for *conflict resolution.* While the ordering guarantees presented previously limit the types of divergences, the model's model for resolving inconsistencies dictates the final sequential ordering of operations and the final state after replicas converge. We give a brief overview of the conflict resolution techniques.

These techniques can be split into syntactic and semantic, as described by Saito [39]. Syntactic conflict resolution is more automated and only depends on histories. Semantic resolution is *application-specific*, either depending on the application to categorize operations or to do the conflict resolution explicitly as in Bayou [43]. We focus on syntactic resolution in this section since it depends solely on visibility constraints and not application functionality.

The simplest conflict resolution technique is the "last-writer-wins" technique, also known as the Thomas Write Rule [44]. In this resolution technique, replicas order writes locally – for example, using local timestamps – and then conflict resolution simply preserves the write with the highest timestamp. However, this can lead to *lost updates*, where concurrently executing writes are lost due to conflicts. Other options include "first-writer-wins", and more complex systems may use multiple timestamps or vector clocks to more carefully resolve conflicts.

Burckhardt [12] proposes a more complex *revision consistency* model for tracking and resolving divergences. This model uses atomic data types, similar to linearizability. They model histories using a revision graph, and, using the graph and a semantic understanding of the data types, they can automatically resolve conflicts without relying on the application.

10 Isolation Models

Isolation models were designed for *transactions* in database management systems. Transactions are application-defined groupings of operations to shared data objects. Systems with support for transactions generally ensure that these groupings of operations are *atomic* (i.e., all operations execute or none), *consistent* (i.e., invariants are preserved after all operations run), *isolated* (i.e., all operations execute as if they run in a separate thread) and *durable* (i.e., all operations are not lost) after commit.

Isolation models serve the same purpose as consistency models for concurrently executing transactions, rather than concurrently executing threads. Gray [24] first defined the isolation levels (called degrees of consistency) in terms of locking implementations. These levels were codified into a standard by ANSI [6] and later refined by Berenson [9] and Adya [3]. In this section, we cover these previously defined isolation levels and others.

10.1 System Model

For now, we will model the system as a single thread executing read and write operations from concurrent transactions to a shared set of data objects (i.e., a uniprocessor system or a sequentially consistent multi-processor system). In the following sections, we will discuss multi-versioned and replicated database system models.

We assume that every operation in the system executes within the scope of a *transaction*:

Definition 32. ***Transaction.*** *We define a transaction as a sequence of read and write operations terminated with either a commit or abort operation.*

We label every transaction with a globally-unique *transaction ID*. Similar to our data object IDs, these IDs have no semantic meaning. Many database systems assign transactions IDs. However, in some distributed systems, globally unique IDs may not be possible; in those cases, we assume an oracle that assigns IDs.

We label every transaction as T_a, where a is the transaction ID. Every read and write operation in T_a is labeled $r_a[x]$ and $w_a[x]$. Every transaction ends with a commit c_a or an abort a_a. We define a transaction's *read set*, to be the set of all data objects involved in a read operation in the transaction and the transaction's *write set* as the set of all data objects with a write operation in the transaction. We assume that the system does not allow *blind writes*, so that the write set is always a subset of the read set.

We define the system model using the following rules:

1. There is a sequential ordering of operations H, that reflects an interleaving of the system's executed transactions, $\{T_1, T_2, \ldots\}$, to a shared set of data objects, $\{x, y, \ldots\}$.

2. If $\alpha_a \prec \beta_a \in T_a$, then $\alpha_a \prec \beta_a \in H$.

3. If $\alpha \prec \beta \in H$, then $\alpha \lhd \beta$.

The first rule states that the system executes transactions with some interleaving of their operations. The second rule assumes that the system executes every transaction in program order. The final rule states that earlier operations executed by the system are visible to later operations, which fits with our unreplicated, single-threaded system model. In fact, this system model is largely identical to our simple state machine model in Section 6. A system's isolation model can be defined as a set of constraints for concurrently executing transactions in this system model.

10.2 Strong Isolation Models

Strong isolation models, or *levels* as they are usually called, maintain the illusion that the system executes each transaction sequentially. There are two isolation levels that could be considered strong isolation: *strict serializability* and *serializability*. Both maintain a *sequential* ordering of transactions – termed *serializability* [18] – with no interleaving of operations between concurrently executing transactions, also called *strict isolation*. Strict serializability additionally ensures *linearizable* histories of transactions, similar to linearizability [25] but for multi-operation transactions instead of operations on atomic objects.

We define strict serializability as follows:

Definition 33. ***Strict serializability*** *requires that, for all pairs of transactions T_a and T_b, (1) either all operations in T_a are globally visible to all operations in T_b or all operations in T_b are globally visible to all operations in T_a, and, (2) if T_a commits before T_b begins, then all operations in T_a are globally visible to all operations in T_b (and vice versa if T_b commits before T_a begins).*

Strict serializability's two constraints can be summarized as follows:

1. There must exist a sequential ordering of transactions, which we will summarize as $(\forall\alpha_A \in T_a | \alpha_a \blacktriangleleft \forall\beta_b \in T_b) \vee (\forall\beta_b \in T_b | \beta_b \blacktriangleleft \forall\alpha \in T_a)$.
2. If a transaction commits before another begins, then the later transaction must see the effects of the earlier transaction, which we summarize as, $\exists\beta_b \in T_b | c_a \prec \beta_b \in H \rightsquigarrow \forall\alpha_a \in T_a | \alpha_a \blacktriangleleft \forall\beta_b \in T_b$.

Strict serializability ensures that the final state of the system is equivalent to one that could be achieved by a single uniprocessor sequentially processing transactions one at a time, in the order in which they are received.

Serializability weakens the real-time ordering constraint:

Definition 34. ***Serializability*** *requires that, for all pairs of transactions T_a and T_b, either all operations in T_a are globally visible to all operations in T_b or all operations in T_b are globally visible to all operations in T_a.*

It only requires a serial ordering of transactions; however, any serial ordering of transactions is acceptable. In particular, serializability allows *snapshot read* transactions, where a read-only transaction executes on a slightly stale snapshot copy of the system state.

Neither strict serializability nor serializability require the system to only run a single transaction at a time. Transactions that do not conflict, where both transactions have the same data object x in their read sets and at least one transaction has x in its write set, can still execute concurrently. The system only has to ensure that the final state of the system is *equivalent* to one where the system ran all transactions sequentially.

10.3 Weak Isolation Phenomena

Next, we discuss weaker levels of isolation that do not fully enforce the illusion of sequentially executing transactions. Many papers in the literature [18, 6, 9, 3] describe weaker isolation levels in terms of disallowed phenomena, so we begin by giving an overview of these phenomena. Weak isolation phenomena can generally be thought of as violations of the visibility constraints for serializability.

Definition 35. ***Weak Isolation Phenomena*** *are visibility relationships between two transactions, T_a and T_b, where some operation $\alpha_a \in T_a$ is visible to operations in T_b* and *another operation $\beta_b \in T_b$ is visible to operations in T_a.*

Researchers have characterized these phenomena based on the type of dependency between the operations that violate the visibility constraints.

We begin by summarizing the phenomena given by Berenson [9] and Adya [3] below. In general, the Berenson definitions disallow more histories because they assume a *pessimistic* system using locking, where the database does not abort transactions. Adya incorporates *optimistic* schemes where the database may roll-back a transaction if it detects a violation of the isolation level. As a result, the Berenson phenomena covers all histories where the phenomena *could* occur, while Adya [3] covers all histories where the phenomena *will* occur in committed transactions.

10.3.1 Dirty Write

As defined by Berenson [9], a *dirty write*, labeled P0, is where transaction T_a modifies a data object, then another transaction T_b modifies the same data object before either transaction commits or aborts. They summarize this as:

$$H_{P0} = w_a[x^1] \dots w_b[x^2] \dots ((c_a \vee a_a) \wedge (c_b \vee a_a))$$

Dirty writes can be avoided by holding *long-duration exclusive write locks*. We define *long-duration locks* to be locks that are held from the operation until commit (or abort), while *short-duration locks* are only held for the duration of the operation. We assume exclusive write locks and shared read locks.

Adya [3] gives a more permissive definition for the phenomena as *interleaved writes*, labeled G0, which we summarize here using visibility:

Definition 36. ***G0 phenomena*** *occurs if, for two transactions, T_a and T_b, there is some write* $w_a[x] \lhd w_b[x]$ and *some write* $w_b[y] \lhd w_a[y]$ *(where x and y can be the same or different data objects).*

This definition allows for optimistic schemes where both T_a and T_b commit with no interleaving of their writes. For example, the history, $w_a[x^1]w_b[x^2]c_a c_b$, can be serialized as $w_a[x^1]c_a w_b[x^2]c_b$ as long as both transactions commit. This history would not be possible in a locking-based system because $w_b[x^2]$ could not execute until c_a releases the long-duration write lock. Without using locks, an optimistic system would allow this history because there is no interleaving. Optimistic schemes can detected interleaved writes using a dependency graph, timestamps or version ids and abort one of the two transactions.

10.3.2 Dirty Read

Berenson describes a *dirty read* (P1) as a transaction T_a modiqfying a data object and then transaction T_b reading the data object before T_a commits, as summarized below by Berenson:

$$H_{P1} = w_a[x^1] \dots r_B[x^1] \dots ((c_a \vee a_a) \wedge (c_b \vee a_a))$$

Dirty reads can be avoided by holding *short-duration read locks*, in addition to long-duration exclusive write-locks.

Adya categorizes dirty reads in three ways: *aborted reads*, which is a read of a value that later disappears because the transaction aborts, *intermediate reads*, which is a read of a value that is later updated before the transaction commits, and *circular information flow*, which is where two transactions both write values based on reads of each others writes. We can generally describe these three phenomena using a single visibility rule based on Adya's dependency graph definition:

Definition 37. ***G1 phenomena*** *occurs if two transactions, T_a and T_b, have operations* $w_a[x] \lhd r_b[x]$ and $w_b[y] \lhd r_a[y]$ *(where x and y can be the same or different data objects).*

G1 is more permissible again than P1 because it allows reads of uncommitted values *if* the transaction later commits and there are no dependency cycles. For example, $w_a[x^1]r_b[x^1]c_a c_b$ is permissible because the history can be serialized as $w_a[x^1]c_a r_b[x^1]c_b$. However, this history would not be possible in a locking-based system because $r_b[x^1]$ would not be able to execute until it is able to acquire the short-duration read lock after c_a releases T_a's long-duration write lock.

10.3.3 Fuzzy Read

A *fuzzy or non-repeatable read* (P2) is a transaction T_a reading a data object and then another transaction T_b modifying the data object and committing before T_a commits. Berenson gives the following summary loose interpretations of the ANSI phenomena:

$$H_{P2} = r_a[x^1] \dots w_b[x^2] \dots ((c_a \vee a_a) \wedge (c_b \vee a_a))$$

Fuzzy reads can be avoided in a locking-based system with long-duration read locks in addition to long-duration write locks.

Adya defines this set of phenomena as *item anti-dependency cycles*, where a transaction reads a data object, then another transaction writes to it and commits, making it out-of-date for the first transactions. We can define these cycles as follows:

Definition 38. ***G2-item*** *occurs if two transactions, T_a and T_b, have operations where $r_a[x^1] \lhd w_b[x^2]$* and *$w_b[x^2] \lhd \alpha_a[y]$ (where x and y can be the same or different data objects).*

Both of these definitions actually encompasses several phenomena separately labeled by Berenson as: (P4)*lost update*, (A5A)*read skew* (A5B) and *write skew*. We summarize these as in [9] in Figure 6.

We can define each of these using visibility criteria as well to separate the different cases. We first give a separate definition for non-repeatable read:

Definition 39. ***Non-repeatable read*** *occurs if two transactions, T_a and T_b, have operations where $r_a[x^1] \lhd w_b[x^2]$* and *$w_b[x^2], c_b \lhd r_a[x^2]$.*

It is important that c_b must be visible to $r_a[x^2]$ in a non-repeatable read, otherwise this phenomena would be labeled a dirty read. This definition separates non-repeatable reads from dirty reads and the following phenomena:

Definition 40. ***Lost update*** *occurs if two transactions, T_a and T_b, have operations where $r_a[x^1] \lhd w_b[x^2]$* and *$w_b[x^2] \lhd w_a[x^3]$.*

This phenomena is called lost update because $w_a[x^3]$ overwrote $w_b[x^2]$ without seeing its effects. T_b does not have to commit before T_a because in either case, $w_b[x^2]$ will be overwritten.

Next, we describe more general cases of non-repeatable read and lost update with multiple objects.

Definition 41. ***Read skew*** *occurs if two transactions, T_a and T_b, have operations where $r_a[x^1] \lhd w_b[x^2]$* and *$w_b[y^1], c_b \lhd r_a[y^1]$ (where x and y are different data objects).*

Read skew is a more general form of non-repeatable reads where x and y are not the same data object; however, as Berenson notes, read skew is more common and harder to detect. Likewise, write skew can be considered a form of lost update where x and y are not the same data object:

Definition 42. ***Write skew*** *occurs if two transactions, T_a and T_b, have operations where $r_a[x^1] \lhd w_b[x^1]$* and *$r_b[y^1] \lhd w_a[y^2]$ (where x and y are different data objects).*

In all of these cases, interleavings of concurrent transactions lead to non-serializable histories.

10.3.4 Phantom

Finally, in systems with support for predicates, a *phantom* is a transaction T_a running a query with predicate P and then another transaction T_b modifying a data object that changes the query results before T_a commits. Berenson gives the following summary:

$$H_{P3} = r_a[P] \dots w_b[y \in P]((c_a \vee a_a) \wedge (c_b \vee a_a))$$

Phantoms can be avoid by taking a long-duration predicate lock in addition to long-duration read and write locks.

Adya [3] characterizes phantoms as part of an *anti-dependency cycle*, where dependencies can be due to predicates. We do not define this phenomena using visibility because predicates are not captured by our notion of visibility. Eliminating phantoms, along with the other phenomena, leads to strict serializability in a locking-based system and serializability in optimistic systems.

10.4 Weak Isolation Levels

Now, we can define weak levels of isolation by eliminating the described phenomena. The earliest models were developed for constraining *single-valued* histories, as defined by Bernstein [11], where all committed and ongoing transactions share a *single* copy of each data item. We discuss these models first and then cover *multi-version* models in the next section.

The weakest isolation level, which is defined as Degree 0 consistency by Gray [24] and PL-0 by Adya [3], allows all of the described phenomena.

$$\begin{aligned}
H_{P4} &= r_a[x^1]\ldots w_b[x^2]\ldots w_a[x^3]\ldots c_a \\
H_{A5A} &= r_a[x^1]\ldots w_b[x^2]\ldots w_b[y^1]\ldots c_b\ldots r_a[y^1]\ldots(c_a \vee a_a) \\
H_{A5B} &= r_a[x^1]\ldots r_b[y^1]\ldots w_a[y^1]\ldots w_b[x^2]\ldots(c_a \wedge c_b)
\end{aligned}$$

Figure 6: Summary of anti-dependency phenomena.

Essentially, it provides no isolation because it allows any interleaving of operations from concurrent transactions. However, read and write operations must be atomic and are presumed sequential. We can define this level as follows:

Definition 43. ***PL-0 (Degree 0)*** *requires that, for all operations in all transactions, α and β, either α must be globally visible to β or β is globally visible to α.*

Since we already assume a sequential history H in our system model, this isolation level gives no additional guarantees. The ANSI standard [6] does not explicitly have this level because almost no databases allow this weakest level of isolation.

10.4.1 Read Uncommitted

The next level is defined as Degree 1 by Gray [24], Locking Read Uncommitted by Berenson [9] and PL-1 by Adya [3]. In this level, the database does not allow dirty write, so a locking-based system would hold long-duration write locks. We can define this level as follows:

Definition 44. ***PL-1*** *requires that, for all pairs of transactions, T_a and T_b, either all writes $w_a \in T_a$ must be globally visible to all writes $w_b \in T_b$ or all w_b must globally visible to all w_a.*

This rule defines PL-1 as requiring a sequential ordering of writes from each transaction in the system. This model could be considered the most general definition of Read Uncommitted. A locking-based system would require that the earlier transaction commit before the first write in the later transaction. So, a more constrained definition would be:

Definition 45. ***Locking Read Uncommitted (Degree 1)*** *requires that, for all pairs of transactions, T_a and T_b, either all write $w_a \in T_a$ and c_a must be globally visible to all writes $w_b \in T_b$ and c_b or all w_b and c_b must be globally visible to all w_a and c_a.*

However, this definition is dependent on the system using long-duration write locks, so we will stick to Adya's portable level (PL) definitions, which work for both pessimistic and optimistic models.

10.4.2 Read Committed

The next level is defined as Degree 2 by Gray, Locking Read Committed by Berenson and PL-1 by Adya.

Definition 46. ***PL-2 (Locking Read Committed, Degree 2)*** *requires that, for all pairs of transactions, T_a and T_b, (1) either all writes $w_a \in T_a$ are globally visible to all writes $w_b \in T_b$ or all w_b are globally visible to all w_a, and (2), if $\exists r_a \succ c_b \in H$, then $\forall w_b \in T_b | w_b \blacktriangleleft r_a$, otherwise $\forall w_b \in T_b | w_b \ntriangleleft r_a$, and likewise, if $\exists r_b \succ c_a \in H$, then $\forall w_a \in T_a | w_a \lhd r_b$; otherwise, $\forall w_a \in T_b | w_a \ntriangleleft r_b$.*

The first part of the rule maintains a sequential ordering of writes across transactions with no interleaving from PL-1. The second part ensures that reads only see committed transactions; however, it does not require that reads in a single transaction see a consistent snapshot of the database or that there is a sequential ordering of reads from each transaction.

The final levels are all variants of strong isolation.

Definition 47. ***PL-2.99 (Locking Repeatable Read)*** *is serializability for all reads and writes but not predicate reads.*

Definition 48. ***PL-3 (Serializability)*** *is serializability for all reads and writes, including predicate reads.*

For transactional key-value systems without support for predicate reads or range queries, like TAPIR [45], Spanner [14] and IBM's Spinnaker [38], PL-2.99 and PL-3 are equivalent.

10.5 Multi-versioned Models

We now explore models designed for versioned systems with *copies* of each data object reflecting different updates. Rather than using locking to prevent concurrent transactions from seeing each others intermediate state updates, these systems use versions to isolate transactions.

These systems produce *multi-version histories*, as defined by Bernstein [11], where reads can go to any available version of a data object at any point in time. Reads of out-of-date versions produce the "appearance" of executing in the past and thus can lead to equivalent histories with very different orderings, as we have seen with our parallel and replicated system models.

10.5.1 Snapshot Isolation

Berenson [9] defines *snapshot isolation* as an isolation level between read committed and full serializability. Snapshot isolation ensures that a transaction always reads from a consistent snapshot of the system state reflecting all transactions that committed before the current transaction began. It uses a first-committer-wins strategy for writes, aborting any transaction that tries to later commit. We define snapshot isolation using visibility as follows:

Definition 49. ***Snapshot isolation*** *requires that, for any pair of transactions T_a and T_b, if $\exists \alpha_a \in T_a | c_b \prec \alpha_a \in H$, then $\forall \beta_b \in T_b | \beta_b \blacktriangleleft \forall \alpha_a \in T_a$; otherwise, $\forall \beta_b \in T_b | \beta_b \not\blacktriangleleft \alpha_a$, and (2) if there is an overlap in the write sets such that $w_a[x] \in T_a$ and $w_b[x] \in T_b$, then if $\exists \alpha_a \in T_a | c_b \prec \alpha_a \in H$, then $w_a[x] \blacktriangleleft w_b[x]$; otherwise $\forall \beta_b \in T_b | c_a \prec \beta_b \in H$ and $w_b[x] \blacktriangleleft w_a[x]$.*

The first requirement in our definition ensures that all reads reflect a consistent snapshot of the system state; however, snapshot isolation is weaker than serializability, so it does not forbid concurrency. The second property ensures the first-committer-wins property of snapshot isolation by stating that writes must follow a read that reflects the latest version of the data object. This property ensures no lost updates (P4) in snapshot isolation.

While snapshot isolation is not serializable, the only phenomena that it exhibits is write skew (A5B). It is difficult to catalog snapshot isolation using Adya's scale as it is designed for optimistic histories but does not cover multi-version histories. Snapshot isolation does not have phantoms, which would place it at PL-3, but does have item anti-dependency cycles, placing it below PL-2.99.

Some systems only use snapshot isolation for read-only transactions, which is equivalent to serializability since snapshot isolation's anomolies only pertain to its writes. Further, it is possible to detect and avoid write skews in snapshot isolation, which is defined as serializable snapshot isolation [19] and implemented in Postgres [37], making it serializable as well.

10.5.2 Monotonic Snapshot Reads

Oracle's *read consistency* [9], later captured as *Monotonic Snapshot Reads* [2], is another multi-versioned isolation level. Every operation (or SQL statement, which can contain multiple operations) executes on a consistent snapshot of the system reflecting all of the transactions that committed before the operation (or statement) began. We define read consistency as follows:

Definition 50. ***Monotonic Snapshot Reads*** *requires that, for any pair of transactions T_a and T_b, if $\exists \alpha_a \in T_a | c_b \prec \alpha_a \in H$, then $\forall \beta_b \in T_b | \beta_b \blacktriangleleft \alpha_a$; otherwise, $\forall \beta_b \in T_b | \beta_b \not\blacktriangleleft \alpha_a$*

This requirement leads to monotonically increasing views of the database because each operation sees all transactions that committed before it started and guarantees a consistent view of the system state for each operation. However, the requirement does not ensure that all operations in the transaction see the *same* consistent view of the database.

10.6 Replicated Weak Isolation Levels

Recently, researchers have developed weak isolation models for replicated databases. These models also assume a system with multiple copies of the system state; however these copies are *replicas* not versions. As a result, each copy may reflect a different execution history, rather than different points in the execution history of the system.

10.6.1 Parallel Snapshot Isolation.

Sovran [41] defined a new version of snapshot isolation, called *parallel snapshot isolation* (PSI). PSI maintains the snapshot isolation model at a single replica and ensures causal consistency across replicas. As a result, we can define PSI as follows:

Definition 51. ***Parallel Snapshot Isolation*** *(PSI) requires that, for any pair of transactions T_a and T_b, if $\exists \alpha_a \in T_a | c_b \prec \alpha_a \in H$ or $T_b \rightsquigarrow T_a$, then $\forall \beta_b \in T_b | \beta_b \blacktriangleleft \forall \alpha_a \in T_a$; otherwise, $\forall \beta_b \in T_b | \beta_b \not\blacktriangleleft \alpha_a$, and (2) if there is an overlap in the write sets such that $w_a[x] \in T_a$ and $w_b[x] \in T_b$, then if $\exists \alpha_a \in T_a | c_b \prec \alpha_a \in H$, then $w_a[x] \blacktriangleleft w_b[x]$; otherwise $\forall \beta_b \in T_b | c_a \prec \beta_b \in H$ and $w_b[x] \blacktriangleleft w_a[x]$.*

The first requirement ensures all reads reflect a consistent snapshot of the system state *and* that snapshot reflect any transactions that the transactions is causally dependent on, while the second requirements ensures no lost updates.

10.6.2 Monotonic Atomic View Isolation

Bailis defined a new isolation level, called *monotonic atomic view* [7] (MAV) isolation. MAV provides an atomicity guarantee for transactions but not true isolation. MAV is an alternative to other weak isolation models for high availability in the face of network partitions.

Definition 52. ***Monotonic Atomic View Isolation.*** *requires that (1) for all transactions T_a and T_b, if some write $w_a \in T_a$ is visible to some operation in T_b, then all writes $w_a \in T_a$ must be globally visible to all operations in T_b.*

While MAV ensures that every transaction sees either all or none of the effects of a transaction, but it does not ensure a consistent snapshot for each transaction. In fact, MAV provides no consistency guarantees at all (i.e., it provide no guarantees for concurrently executing transactions).

10.6.3 Read Atomic Isolation

Read atomic isolation extends MAV to ensure a consistent snapshot of the system for each transactions. We define it as follows:

Definition 53. ***Read Atomic Isolation.*** *requires that (1) for all transactions T_a, either all operations $\alpha_a \in T_a$ are globally visible to all $\beta_b \in T_b$ or all operations $\alpha_a \in T_a$ are globally* not *visible to all operations $\beta_b \in T_b$.*

While MAV ensures that every operation sees a monotonically progressing view of the database, read atomic isolation ensures a consistent snapshot of the system to all of the operations in the transaction. As a result, read atomic isolation is able to ensure repeatable reads (P2), which Bailis [7] defines as *cut isolation*, while MAV cannot.

11 Summary

In this paper, we have reviewed semantic memory models from architecture, distributed systems and database research. We have categorized and compared their system models. In this section, we summarize the semantic memory models that we have covered and discuss some insights and future directions for researchers.

Table 2 lists the models that we reviewed. Strong semantic memory models differ very little despite the large differences in system model assumptions across research areas. In general, the only difference between models was whether they provided a guarantee of a sequential history or a strict sequential history, as we defined in Section 5.3. Weak semantic memory models differ much more based on the system model assumptions because the system model constrains which *anomalies* are possible in weaker models.

11.1 Co-designing semantic memory models

Researchers are building new systems today that have more complex system models than systems of the past. New distributed storage systems are often replicated, partitioned, parallel and transactional. In order to provide a unified semantic memory model to the application programmer, these systems must integrate several mechanisms,

Table 2: Summary of semantic memory models covered in this paper

	Strict Sequential	Sequential	Causal	Weak	Eventual
Program Ordering	Strict Program Ordering	Relaxed Program Ordering	–	–	–
Coherence	Strict Coherence	Loose Coherence	–	Loose Coherence	Cache Coherence
Consistency	Linearizability	Sequential Consistency 1-copy Serializability	Causal Consistency Causal+ Consistency	Total Store Ordering Processor Consistency Partial Store Ordering Weak Consistency Release Consistency	Read Your Writes Monotonic Reads Writes Follow Reads Monotonic Writes Lazy Replication
Isolation	Strict Serializability	Serializability	–	Snapshot Isolation Monotonic Snapshot Read Repeatable Read Read Committed Read Uncommitted	
Distributed Isolation	”	”	Parallel Snapshot Isolation	”	Monotonic Atomic View Read Atomic

each with a semantic memory models and different system models. For example, Gray [23] designed a protocol to provide serializable isolation for a distributed and replicated system by layering a distributed transaction protocol – two-phase commit and two-phase locking – on top of a replication protocol – Paxos [30].

Building new systems by layering past system mechanisms makes sense because it re-uses previously understood mechanisms and enforces abstractions between different system components. However, without understanding the semantic memory model and system model of *each* layer, system designers can easily end up with a system that has violations of the system model assumed by higher levels or redundant constraints in the semantic memory model at lower levels. Ignoring the way that these models interact can either lead to violations of the final system's semantic memory model or wasted performance.

For example, in a system with a strict coherence model, layering a weak consistency model on top may not have the expected performance. If the coherence model synchronizes all copies of each data object after every write, then regardless of the consistency model, the system will have sequential consistency. Likewise, if the coherence model is very weak, it may violate the consistency model and introduce unexpected anomalies. Bennett [8] recognized this interaction when the developing of loose coherence, which provides a coherence guarantee matched to the consistency model.

Likewise, in a system with a strong isolation model, a strong consistency model is not necessary because there are few valid interleavings of concurrent operations. Instead, threads only need to synchronize between transactions. In fact, all accesses in a locking-based database are *data-race-free*, so a weak consistency model like release consistency is sufficient. However, a coherence model that is too weak will violate the assumptions of the system model of the isolation mechanism. For example, most isolation mechanisms do not take replication into account, so a coherence model that executes operations in different orders at each replica would violate the isolation model's system model.

The key observation here is that many semantic memory models developed in the past no longer constrain a system for application programmer but are instead layered into a complex system. Thus, it has become necessary for researchers to think beyond their own layers and consider how their system model and semantic memory model would fit into a larger system. Rather than designing isolated semantic memory models for a specific system model, we should be co-designing *integrated* semantic memory models developed specifically to be easily layered with other models.

11.2 Concurrency Control in Distributed Systems

Many large-scale distributed systems today are massively parallel. Distributed storage system have large numbers of web servers concurrently accessing data and distributed web applications serve large numbers of interactive clients around the world. However, application programmers have few tools for *control* the concurrency in these systems.

There is a significant amount of work on providing various types of consistency for distributed systems [27, 33, 31, 26, 13, 15]. However, while consistency models provides a good way for programmers to reason about the execution of concurrent operations in distributed systems, they provide no way for application programmers to control the concurrent execution. This lack of control makes reasoning about massively concurrent execution in these distributed systems difficult. For example, given a read-update-write in two threads, $r_A[x]w_A[x]$ and $r_B[x]w_B[x]$, a distributed system today, even one with strong consistency, could produce any interleaving of these operations. This lack of control for application programmers also translates to a lack of information for the system. If a distributed system knew that certain operations were data-race-free, then it could easily avoid expensive distributed coordination.

We have successfully deployed multi-processors with weak consistency for decades, largely because these systems provide application programmers, compilers and the operating system with mechanisms for controlling the system's behavior with barriers, fences and other synchronization mechanisms. In many cases, application programmers do not have to interact with these weak memory models because they have another subsystem, like the OS, layered on top.

Further, strong consistency models, like that provided by classic algorithms like Paxos [30], are often not necessary in replicated, distributed systems because they make up the lowest level in a system's layered architecture. As observed by Saltzer [40], strong guarantees at the lowest levels are often redundant. Thomas [44] and TAPIR [45] both show that a replication protocol with a strong guarantee is not necessary if there is a strong isolation model layered on top.

However, we must provide concurrency control mechanisms that *prevent* conflicts as in multiprocessor architectures, not eventual consistency models that require application programmers to resolve conflicts after they arise. Resolving conflicts is often difficult, if not impossible, and requires application-specific knowledge in many general cases. This requirement for preventing conflicts means that, while the amount of distributed coordination can be reduced, it cannot be entirely eliminated to provide availability with partitions [22].

The key observation here is that strong consistency without concurrency control is both hard for application programmers to use and hides useful application-specific information from the system. In massively concurrent distributed systems, we must develop consistency models with concurrency controls, rather than strong or eventual consistency models.

References

[1] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Transactions on Computers*, 29(12):66–76, 1996.

[2] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D., MIT, Cambridge, MA, USA, March 1999. Also as Technical Report MIT/LCS/TR-786.

[3] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering*, 2000.

[4] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[5] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the International Conference on Software Engineering*, pages 562–570, 1976.

[6] ANSI. *American National Standard for Information Systems – Database Language – SQL*, 1992.

[7] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases*, 2014.

[8] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of PPOPP*, 1990.

[9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the ACM SIGMOD Conference*, 1995.

[10] Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the ACM SIGMOD Conference*, pages 923–928, 2013.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *Proc. of the European Conference on Object Oriented Programming*, pages 283–307, 2012.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[14] James C. Corbett et al. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2012.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Symposium on Operating System Principles*, 2007.

[16] G. Delp, A. Sethi, and D. Farber. An analysis of memnet&mdash;an experiment in high-speed shared-memory local networking. In *Proceedings of the ACM Symposium on Computer Communications*, 1988.

[17] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *Proceedings of the ACM Symposium on Computer Communications*, 2012.

[18] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 1976.

[19] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[20] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Symposium on Operating System Principles*, 1989.

[21] K Gharachorloo, D Lenoski, J Laudon, P Gibbons, A Gupta, and J Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of International Symposium on Computer Architecture*, 1990.

[22] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[23] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 2006.

[24] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In *In Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[25] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

[26] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference*, 2010.

[27] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Proceedings of the ACM European Conference on Systems*, 2013.

[28] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 1992.

[29] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(9):690–691, 1979.

[30] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[31] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[32] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[34] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Transactions on Computers*, 24(8):52–60, 1991.

[35] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 1988.

[36] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[37] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. In *Proceedings of the International Conference on Very Large Data Bases*, 2012.

[38] Jun Rao, Eugene J Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the International Conference on Very Large Data Bases*, 2011.

[39] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 2005.

[40] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[41] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Symposium on Operating System Principles*, 2011.

[42] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data.

In *Proceedings of the International Conference on on Parallel and Distributed Information Systems*, pages 140–150, 1994.

[43] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Symposium on Operating System Principles*, 1995.

[44] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[45] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurhty, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. Technical Report UW-CSE-14-12-01, University of Washington, 12 2014.

[46] S. Zhou, M. Stumm, and T. McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the International Conference on Distributed Computing Systems*, 1990.