Chapter 1
Consistency Models for Replicated Data

Alan D. Fekete and Krithi Ramamritham

Abstract There are many different replica control techniques, used in different research communities. To understand when one replica management algorithm can be replaced by another, we need to describe more abstractly the consistency model, which captures the set of properties that an algorithm provides, and on which the clients rely (whether the clients are people or other programs). In this chapter we describe a few of the different consistency models that have been proposed, and we sketch a framework for thinking about consistency models. In particular, we show that there are several styles in which consistency models can be expressed, and we also propose some axes of variation among the consistency models.

1.1 Introduction

The research of decades has produced many ideas for managing replicated data, in contexts including distributed systems, databases, and multiprocessor computer hardware. The chapters of this book present many system designs, which differ in how replicas are updated, when they are updated, how failures are handled, and so on. In this chapter, we try to abstract away from particular system designs, to think about the functionality that a replicated system gives to its users.

Even systems that are widely different internally can be functionally interchangeable. For example, users should not be aware of whether the replicas are communicating through a group communication infrastructure (described in Ch 3.2 and Ch 5), or running a consensus protocol (see Ch 3.3 and Ch 4); users shouldn't need to care whether each replica stores a single version of the data, or multiple versions. Different system designs may have different characteristics for performance or fault-tolerance, but the users should be able to take code that runs on one system, and run it on another system without changing the behavior.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 1–17, 2010.

However, in some cases, the functionality itself changes between one system design and another. For example, a common theme in replication research is to seek improved performance by giving up some level of consistency between replicas (see Ch 6.4). For a user, it is very important to know exactly what functionality one can rely on, from each replicated data management system. A replication consistency model is a way to abstract away implementation details, and to identify the functionality of a given system. Like any other careful specification, a consistency model is a way for people to come to a common understanding of each others rights and responsibilities. For a consistency model, the people who must jointly understand the model are the users (who learn what they can rely on the system to provide, and what assumptions they might be inclined to make that are not guaranteed), and the system designers (who document the features of their design that will not change, as performance tuning, and other optimization is done).

A consistency model is a property of system designs, but a particular consistency model is usually presented in terms of a condition that can be true or false for individual executions. That is, we can determine whether or not the condition holds for a possible execution: one pattern of events that might occur at various places and times as the system runs (including information about the type and contents of messages, when and where these messages are sent and received, the state of each replica after each event, etc). If *every* possible execution that can occur for a system design makes the condition true, then we say that the system design itself satisfies the consistency model.

1.1.1 Contributions

The remainder of the chapter is structured as follows. In Section 1.2 we deal with the need to formalize the properties of the sequential data type which is being replicated. In Section 1.3 we explain the strongest consistency model, where the users can't ever discover that the data is replicated. We also introduce two styles for presenting consistency models: an operation-ordering style where there are logical conditions on how operations can be ordered into a sequence that is allowed by the sequential data type, and an ideal-system style where a abstracted state-machine model is given that generates executions which are indistinguishable to the clients from those in the real system. Section 1.4 looks at some weaker notions of consistency, that are provided by many systems as a tradeoff for better performance or better availability. In Section 1.5 we mention some of the ways in which consistency models can be defined for replicated databases; here we have a richer interface, where several operations can be grouped in a transaction. In Section 1.6 we comment on some general issues that are raised by the wide variety of different consistency models that have been proposed. Finally we conclude with a summary of the main ideas of this chapter.

1.2 Defining the Sequential Data Type

Before offering a definition of a particular consistency model, it is important to know what operations the clients will be submitting, and how these operations should be

understood. This aspect of the model is captured by a sequential data type, which is a formalization of the semantics of the operations, or equivalently, of the unreplicated system that users understand or to which they relate the replicated system. Seminal early work on specification of a data type was done by Liskov and Zilles [12].

The simplest possible sequential data type is the read-write single-bit register. Here the operations are to read the value, or to write a new value (overwriting whatever is already present). Thus there would be three legal operations: read(), write(0), write(1). The return value from a read is either 0 or 1; the return value from a write is "OK". With a bit more realism, we can consider the read-write 32-bit register, whose operations are read(), and write(v) for each v from 0x00000000 to 0xFFFFFFFF.

The simple registers described above have read-only operations which do not affect the state, and update operations that modify the state but do not reveal anything about it. It is also possible to have more complicated operations that both modify state, and return a value based on the state. Many hardware platforms actually support operations like that. For example, a CAS-register offers compare-and-swap operations (as well as read and write). The effect of compare-and-swap($v1$, $v2$) depends on the previous value in the register. If the value present is equal to $v1$, the operation changes the value to be $v2$; otherwise the effect is to leave the register unchanged. The return value from this operation is always the value that was present in the register before the operation.

In distributed computing theory, there has also been consideration of traditional data types such as a queue or stack. Another direction for finding interesting sequential data types is to consider a type with many different locations. For example, a multi-location read-write memory has a set of locations (or addresses) A, and operations such as read(a) for $a \in A$, or write(a, w) for $a \in A$ and w in some finite domain of values. One can then also allow operations that deal with several locations at once. For example, a snapshot memory has an operation snapshot(), which doesn't modify the state but returns a collection of values, one value for each location.

We can formalize a sequential data type by a set of operations O, a set of states S, an initial state s_0, a set of return values R, and two functions[1]: next-state: $O \times S \to S$ and return-value: $O \times S \to R$. For example, a multi-location byte-valued snapshot memory is a sequential data type where $O = \{\text{read}(a)$ for $a \in A$, write(a, w) for $a \in A$ and $w \in W$ (here W is the set of all 8-bit constants), and snapshot() $\}$; the set of states S consists of all functions $A \to W$, the initial state has all locations mapped to the zero word; the return values are elements of W (returned by read operations), the single string "OK" (returned by write operations) and the set of functions $A \to W$ (returned by snapshot operations; thus the return value from a snapshot is actually a state). The next-state function is defined by next-state(read(a),

[1] In this chapter we only deal with data types where the operations are total and deterministic, so an operation has a unique next-state and return value, when applied in a given state. This is implicit in having functions for next-state and return-value, and it sometimes requires adding error return-values and error states to the model, for cases such as performing a pop() on an empty stack. The theory becomes somewhat more complicated if we loosen the model to allow nondeterministic operations (which may move to one of several next-states) or partial operations (which may not be allowed in a particular state).

s) = s, next-state(write(a, w), s) = t where t is the function from A to W such that $t(l)$ = s(l) if $l \neq a$, and $t(a) = w$, and next-state(snapshot(), s) = s. The return-value function is return-value(read(a), s) = $s(a)$ [recall that a state is a function A to W], return-value(write(a, w) = "OK", return-value(snapshot(), s) = s.

Using these functions, or indeed as an alternative formalization, we can look at the possible legal histories of the data type: each history is a sequence of pairs (operation, return-value), where an operation is paired with the return value it receives, when performed in the state that results for all the operations before it in the sequence, done in order. Thus a legal history for a 4-location byte-valued snapshot memory might be the following:

$$\begin{aligned} &(\text{write}(1, 5), \text{“OK”}) \\ &(\text{read}(1), 5) \\ &(\text{read}(2), 0) \\ &(\text{write}(2, 7), \text{“OK”}) \\ &(\text{snapshot}(), (0 \mapsto 0, 1 \mapsto 5, 2 \mapsto 7, 3 \mapsto 0) \\ &(\text{write}(3, 2), \text{“OK”}) \end{aligned} \tag{1.1}$$

One can use the sequential data type to carefully define concepts such as when an operation does not modify the state, or when operations commute with one another. Some techniques for implementing replication are specific to particular sequential types, while other algorithms work oblivious of the sequential type. Similarly, some consistency models rely on a particular sequential type, for example, by using the fact that each operation is on a single location, or by treating read operations differently from writes.

1.3 Strong Consistency

The principal goal of research on consistency models is to help application developers understand the behavior they will see when they interact with a replicated storage system, and especially so they can choose application logic that will function sensibly. The very easiest way a developer can understand the behavior of a replicated system, is to simply ignore the replication: if the application program gets the same behavior as with a single site, unreplicated system, then writing the application logic is no different to conventional programming. This transparency concept can be captured by saying that a system execution is *linearizable*; one similarly says that a system is *linearizable*, if every execution it produces has this property. The term "atomic" is often used instead of linearizable; we do not do so, to avoid confusing with the database meaning of atomic, described in Section 1.5 below. The idea of linearizability is very prominent in research in theory of distributed computing, having been introduced in [8]. Below we give an example to illustrate the concept, and then we discuss some ways to make this precise.

In Figure 1.1, we show one execution that is linearizable, for a system where the underlying serial data is the 4-location byte-valued snapshot memory mentioned

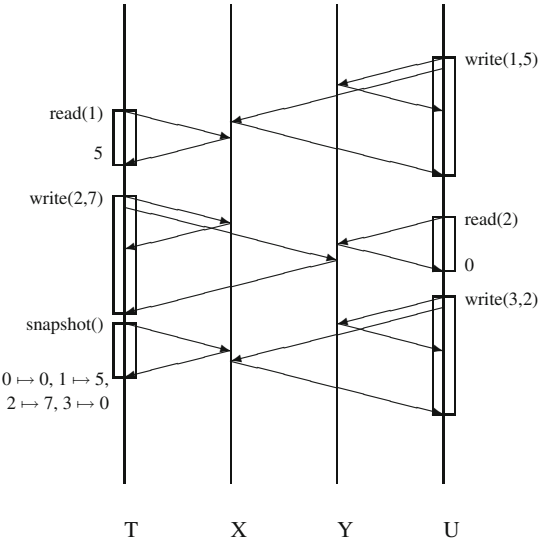

Fig. 1.1 Linearizable Execution.

above. This execution could arise in a system that has two replicas of the data (at sites X and Y), and clients located at T and U each follow a replica management algorithm: each read or snapshot is done on one replica, each write is done on both replicas, different writes are done in the same order at the replicas, and a write doesn't return to the client until all replicas are modified. A figure like this is called a space-time diagram; it shows time increasing down the page; the activity at each location (either a client or a replica) is shown as happening on a vertical line, and messages are shown as diagonal arrows, starting at a point on the vertical line representing the sender, and ending on the line representing the destination location. We use a rectangle around part of the client's vertical line, to indicate the duration of one operation, from its invocation till the operation returns and the next operation can begin. The details of the operation are written next to the top of the rectangle, and the return value is written next to the bottom of the rectangle (except that in this diagram we omit the uninformative return value "OK" from a write operation)

One approach to giving a precise definition of linearizable execution is based on the operations and their order. We take all the operations that occur in the execution, pairing the operation (name and arguments) with the return value produced in the execution, and then we must find a sequence (total order) of these pairs that is allowed by the sequential data type being replicated. An additional requirement is that the order of operations as they occur in the sequence must not contradict any order information visible to an observer of the system execution. Thus, we define a real-time partial order $<_{E,rt}$ between operations that occur in the execution E, where $p <_{E,rt} q$ means that the duration of operation p (from invocation till it returns) occurs entirely before the duration of operation q; in other words, the return from p

occurs at an earlier real time than the invocation of operation q. Notice that this is a partial order; if two operations overlap, they are unrelated in this order. We can then define that an execution E is *linearizable* provided that there exists a sequence H such that

[L1] H contains exactly the same operations that occur in E, each paired with the return value received in E,

[L2] the total order of operations in H is compatible[2] with the real-time partial order $<_{E,rt}$,

[L3] H is a legal history of the sequential data type that is replicated.

We can apply this definition to the execution in Figure 1.1. For brevity in the discussion, we name an operation by mentioning only the location being written and not the value written; the execution involved has at most one write on each location, so there is no ambiguity introduced by eliding the value. In this execution, the read(1) operation occurs entirely before the operations write(2), read(2), snapshot() and write(3); however it overlaps with write(1). Similarly read(2) occurs entirely before snapshot() and write(3), but it overlaps write(2), and also write(3) overlaps both write(2) and snapshot(). A suitable H is the sequence given as (1.1) in Section 1.2 above. Notice that the order of operations in H is compatible with "occurs entirely before" , for example, read(1) is earlier in H than write(2), read(2), snapshot() and write(3). In this particular example, there is only one possible H that meets all the conditions [L1], [L2], and [L3]. In general, however, there may be several sequences H with these properties. The definition of linearizable merely asks that at least one such H exists.

A different style, based on an ideal system model, has also been used in the literature to define consistency properties. This style of definition involves providing an explicit system model for the unreplicated system that users can imagine they are dealing with. For linearizablity, we consider an ideal system, in which there is a single site where the data is stored, and this site also has a collection of pending requests, and a collection of pending responses. Whenever a client requests an operation, the operation is placed among the pending requests. At any time (non-deterministically) the system can (in one indivisible step) chose a pending request, perform it on the (unique) data, take the result into the collection of pending responses, and remove the chosen operation from the collection of pending requests. At any time, the system can non-deterministically chose a pending response, remove it from the collection, and return that value to the client which had originally requested the operation. This can be formalized as a state-transition machine. The definition of a linearizable execution E is that there exists an execution F of the unreplicated ideal system, such that E and F contain exactly the same steps at all the clients, in exactly the same real-time order.

[2] Two partial orders on the same set are called *compatible* provided that there exists a partial order containing their union. Equivalently, their union has no cycle. In this particular situation, the order in H is a total order, so compatibility is simply expressing that whenever $p <_{E,rt} q$ then p comes before q in H.

1.3.1 Relaxing Inter-Client Operation Ordering

The intuitive justification for the definition of linearizable, is that the replicated system gives the same functionality as the sequential data type. This is built on the idea that there is an "external observer" who is aware of all the activities of all the clients (but not internal activity within the replicas), and we require that what the observer sees in the real system is the same as what they would see in a system with a single copy of the sequential data. In particular, condition [L2] in the operation-order based definition, builds in the notion that the observer knows whether one operation occurs entirely before another, and similarly the ideal system model must have the same order of client steps (that is, the observer can see which client steps occur first, even when these are at different locations). For many issues in a distributed system, it is normal to disregard the order of activity at different locations, in cases where such order can't be determined by any observer within the system. This leads to various relaxations of the consistency model, which allow some operations to appear out of their real-time order. However, we do not usually allow arbitrary changes in the apparent order, and insist that some operations form a session, whose order must be respected in the apparent unreplicated system that users believe they are dealing with.

A consistency model of this kind is used in research on theory of distributed computing where it is called *sequential consistency*. An example of an execution that is sequentially consistent, but is not linearizable, is shown in Figure 1.2. This execution can be produced by an implementation in which each read or snapshot is done on one replica, each write is done on both replicas, different writes are done in the same order at the replicas, and a write returns to the client as soon as the messages have been sent out to the replicas (in this replica management approach, there is no need for the replica to reply to the client after a write, and indeed the write operation may return before the replicas are actually modified).

An operation ordering definition of sequential consistency uses a client partial order $<_{E,c}$ on the operations, in which $p <_{E,c} q$ means that the p and q occur at the same client and that p returns before q is invoked. We can then define that an execution E is *sequentially consistent* provided that there exists a sequence H such that

[SC1] H contains exactly the same operations that occur in E, each paired with the return value received in E,

[SC2] the total order of operations in H is compatible with the client partial order $<_{E,c}$,

[SC3] H is a legal history of the sequential data type that is replicated.

We see that SC1 is the same as the earlier L1, and SC3 is the same as L3; thus the only difference between the definition of sequential consistency and that of linearizability, is in the change from L2 to SC2; for sequential consistency, H is required to be compatible with a weaker partial order. Thus any execution that is linearizable is also sequentially consistent, but the converse does not hold, as we show in an example.

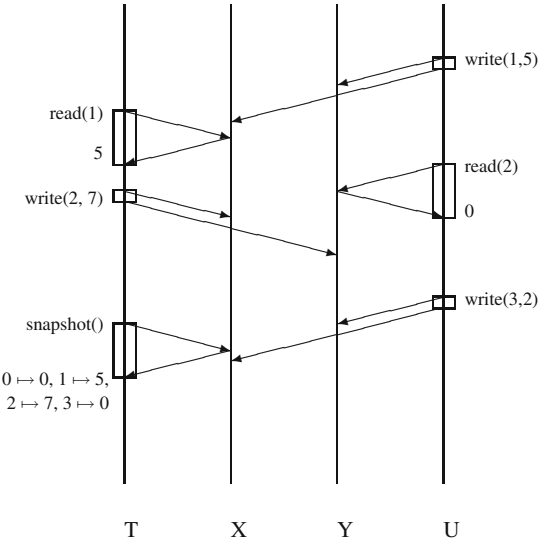

Fig. 1.2 Sequentially Consistent Execution.

The execution of Figure 1.2 is sequentially consistent; the sequence (1.1) above is a suitable choice for H to show this. Note that the operations write(3) and snapshot() are not related by the location partial order because they are at different clients (even though the duration of write(3) occurs entirely before the duration of snapshot()), and so it is acceptable for H to place snapshot() before write(3). The execution of Figure 1.2 is not linearizable, because for any sequence H in which snapshot() has the return value found in the execution, write(3) must be after snapshot(); such an order is not compatible with the real-time partial order on operations, where write(3,2) $<_{E,rt}$ snapshot().

The term sequential consistency is also used by researchers in computer architecture[3]. For this community, the client is envisioned as a CPU issuing instructions that operate on a shared memory (and also other instructions that are purely local to the client, such as those on registers); in contrast to the model we gave above, the client does not see the point at which an operation returns (merely, the return value must be available to the client when a later instruction makes use of it). Thus, in this community, the client order $p <_{E,c} q$ is defined by p and q are operations of the same client, and p is submitted before q is submitted. If each client does not submit any operation until it has the return value from the previous operation at that client, then the two definitions are the same. Sequential consistency is the strongest consistency model provided by common multiprocessor hardware.

[3] Indeed, while the concept was invented by Leslie Lamport who is most prominent as a distributed computing researcher, the paper[10] in which he defined the term is devoted to programming multiprocessor hardware.

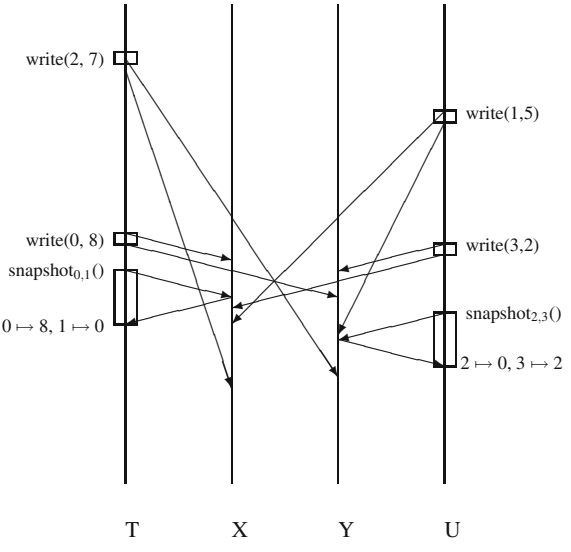

Fig. 1.3 Sequential Consistency Does Not Compose.

One significant aspect of the definition is that sequential consistency is not a compositional property; if we separately replicate two data items, and each is managed to be sequentially consistent, it may happen that the combined data is not sequentially consistent.

To illustrate this, consider a system with two separate two-location byte-valued snapshot data items. For clarity, we will give one data item locations called 0 and 1, and the other item's locations will be called 2 and 3. Thus the system has an operation (which we will call snapshot$_{0,1}$()) to observe the values in locations 0 and 1; another operation snapshot$_{2,3}$() observes locations 2 and 3. We might have a system design based on read-any, write-all, with writes returning immediately, and we could use a separate communication infrastructure for the messages that deal with each data item; that is, the various replicas all see the same order for messages that deal with writes to locations 0 and 1, and they all see the same order for messages that deal with writes to locations 2 and 3; however, it can occur that the relative order in which replicas receive two messages is different, in the case where one message deals with locations 0 or 1, and the other message deals with the other data item. In Figure 1.3, we show a possible execution of this system.

The execution of Figure 1.3 is not sequentially consistent. The return value of snapshot$_{0,1}$ reports that write(0) appears to occur before the snapshot, and write(1) appears to occur after it; similarly the return value of snapshot$_{2,3}$ shows that write(3) appears to occur before write(2). The client partial order of T shows that write(2) must be before write(0), and the client partial order of U shows that write(1) must be before write(3). These relationships can't all hold at once.

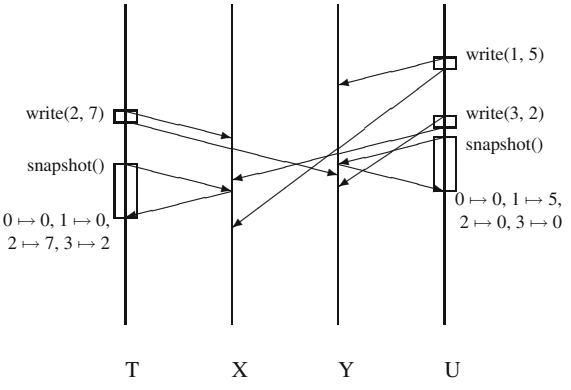

Fig. 1.4 Execution that is not Sequentially Consistent.

On the other hand, if we just consider the data item with locations 0 and 1, the operations on that item within Figure 1.3 are sequentially consistent. A suitable H to demonstrate this is the sequence with write(0) $\text{snapshot}_{0,1}()$ write(1). Similarly the sequence write(3) $\text{snapshot}_{2,3}$ write(2) shows that when we just look at the operations on locations 2 and 3, they are also sequentially consistent.

In contrast, two linearizable replicated items run next to one another forms a linearizable system for the combined data. We say that the property of being linearizable is a compositional property.

1.4 Weak Consistency

Strong consistency can be provided by appropriate hardware and/or software mechanisms, but these are typically found to incur considerable penalties, in latency, availability after faults, etc. In every research community where replication has been used, there have been proposals to offer the clients less guarantees than a transparent single-copy image, in order to deliver better performance. For example, strong consistency designs typically require all replicas to receive messages in the same order. A communication system that keeps messages in the same order typically sends all messages through a single sequencer node or on a common bus, and this becomes a bottleneck for the whole system. If instead we allow messages to be carried independently, and applied to each replica whenever they arrive, performance may be much better, but the clients can see inconsistencies that would never happen with unreplicated data [16]. Consider the execution in Figure 1.4.

In the execution shown in Figure 1.4, there is no possible history H that can meet both SC1 and SC2, to fit with the information returned in the two snapshot() operations. The snapshot() done by client T reveals the effects of the write(2) and write(3) operations, but not the effects of write(1); thus any history that contains this snapshot will have to place write(2) and write(3) before the snapshot(), and place write(1) after the snapshot. In contrast, the snapshot() done by client U sees

the effect of write(1) but not the effects of write(2) or write(3); so any history that contains it will have to place write(1) before both write(2) and write(3). These requirements are contradictory. Thus in this execution, each client sees information that is inconsistent with what the other sees. This execution also gives client U information which is internally inconsistent with the order in which the client itself submits operations: the snapshot() in client U returns information which does not show the previous write(3) operation. We can say that this execution has re-ordered the write(3) and the snapshot() within client U. Effects like these are characteristic of weak consistency models. Different weak models are separated from one another by the precise details of which reorderings are allowed, and which are not, within the activity of a client, and also by whether there are any constraints at all on the information provided to different clients.

We now propose a common framework, within the operation-ordering style, in which different weak consistency models can be defined. This is inspired by the framework for multiprocessor hardware consistency models [17]. We base our framework around each operation o in the execution E having a *justification* J_o, which is a sequence of other operations that occurred in E, such that the return value o received in E is that which the sequential data type would give to operation o when performed in the state which is produced by starting in the initial state and then applying each operation in J_o in turn. Note that we put a condition on the return value of o in relation to J_o, but we do not require (in the general framework) that the earlier operations have the same return values in J_o as they have in E, though this might be an added constraint in particular weak consistency models.

In a typical read-any-write-all design (such as in Ch 2) of a replicated system, we can take J_o to be the sequence of operations that had already been applied to the replica where the return value of o was calculated. For the execution shown in Figure 1.4, a justification for the snapshot() at client T could be the sequence write(2) write(3), and a justification for the snapshot() at U could be the sequence write(1). A given consistency definition then places some constraints on the relationships between the justifications of various operations. More precisely, a consistency model requires that justifications can be found for every operation, such that the set of justifications are related as the consistency model constrains them.

As one example of a weak consistency model, the computer architecture community describes some hardware as offering the *release consistency* model[4]. In release consistency, some of the operations are labelled as special: a special read operation is also called an acquire, and a special write operation is called a release. The requirements of an execution E to satisfy release consistency are that there exists a total order $<_{spec}$ on all the special operations, such that

[RC1] for every operation o, the order of special operations in J_o is compatible with $<_{spec}$

[RC2] for every operation o, J_o contains any acquire that occurs before o at the same client

[4] To be precise, the model we define here can be called RCsc; some authors instead give a slightly relaxed meaning which can be called RCpc, as described in [1].

[RC3] for every operation o, if J_o contains a release operation r, and p is any operation that occurs before r at the same client as r, then J_o contains p before r,

[RC4] for every operation o, J_o contains an operation q, and a is an acquire that occurs before q at the same client as q, then J_o contains a before q.

That is, the hardware makes sure that whatever order is seen anywhere in the system respects the order at a single client from an acquire to a subsequent operation at the same client, and from any operation to a subsequent release at the same client. A similar but more complicated memory model with synchronizing operations is that provided for Java programmers by the Java Virtual Machine [13].

To define a weak model within the ideal system style, one needs to come up with a state-transition system whose executions are exactly those where each operation has a justification. The ideal system typically can have a state which keeps all operations that have been requested and not yet dealt with, and also the state keeps a set of operations that have been dealt with, each with its associated justification (and an additional flag to show whether the return value has been given back to the client). A transition is given for a non-deterministic hidden event which deals with one request: it chooses a request that has not been dealt with, and a set of operations to justify it in accordance with all the requirements (such as RC1-RC4 above) on compatibility of justifications. Another transition is used for returning to the client, with the unique return value that is allowed by the sequential data type, if the requested operation were to be done immediately after all the justifying operations in turn.

In the distributed systems community, a model that has attracted a lot of attention is *eventual consistency*. This is the consistency model that allows the system to be highly-available, even in the face of network partition; it is often described as allowing "disconnected operation" [6]. The main implementation technique is to allow the client to update any replicas, and then the information about the change is propagated in the background[5] to other replicas through gossip messages. Because non-commuting updates can arrive at different replicas in different orders, a conflict-resolution mechanism is needed, in which some previously applied operations can be undone when a new message brings information about an operation which ought to have preceded the ones already done; the missing operation is applied, and after that, the replica re-applies the earlier messages that it had just undone. The recent focus on cloud computing services has made eventual consistency very well known[19]. Because eventual consistency is a liveness property rather than a safety property, it is harder to make precise, and any definition will need to deal with an infinitely long execution (representing one way the system can behave, observed without limit). In much of the research literature, eventual consistency is described using phrases similar to "replicas converge towards identical copies in the absence of further updates". This definition is less general than desirable, because it is expressed in terms of implementation detail (the values in the replicas), and because it involves a counter-factual (after all, updates never stop arriving in many executions). We offer here a new definition based on operation justifications; our

[5] This is also called anti-entropy.

definition does hold for a system built with gossip messages and conflict resolution, as described above. An infinitely long execution E of the replicated system satisfies *eventual consistency* if there exist justifications J_o and an sequence of operations F of the sequential data type, such that

[EC1] F contains exactly the same operations that occur in E,
[EC2] for every prefix P of F, there exists a time t in E such that for every operation o that occurs after t, the justification J_o has P as a prefix.

In this definition, we call a prefix P the "agreed past" for all the operations o mentioned in (EC2). The sequence F orders all the operations in the way that conflict resolution places them, and over time, longer and longer prefixes of F become fixed at all the replicas. In a gossip-based implementation, the agreed past for an operation is the beginning of the justification, which contains operations that will never be undone again, because all the previous operations have propagated to every replica.

To make an eventually consistent system more convenient for clients, it is common to place additional constraints on the contents of the justifications. These have been called "session properties" because they allow the client to do several operations one after another, and avoid some confusing situations[18]. One session property is *Read Your Writes*, which constrains the justifications so that for any read operation o submitted by a client, J_o contains all write operations that were submitted by the same client before o. The *Monotonic Reads* property constrains justifications so that if a client submits read operation $o1$ and later submits a read operation $o2$, then J_{o2} includes all of J_{o1}.

The *causal consistency* session model is a restriction of eventual consistency, based on the causal order between operations, defined by Lamport [11], where $o1 < o2$ means that information from o1 can flow to o2; formally, the Lamport causality partial order is the transitive closure of the order of operations at each client, and the partial order which relates a message send and the receipt of the same message. In causal consistency, the justification J_o must contain all operations that come before o in the causal partial order; also, whenever q occurs within J_o, and $p < q$ in the causal partial order, then p occurs in J_o before q. An interesting twist on causal consistency was proposed by Ladin et al [9], who allow the application code to choose, for each operation, a subset of the causal precedent operations, and then require only this subset to be in the justification for the operation, and always ordered before that operation in justifications.

Another style of additional constraint in replicated distributed systems is to bound the divergence between replicas, so clients see values that are reasonably "fresh". Several researchers introduced these ideas around the same time, one example was [3]. Again, this model is sometimes explained in terms of the implementation internals (limiting how far apart the values are at a given time, or limiting the time period between when an update is applied at different replicas). It can also be expressed as a constraint on the justifications: for example, we may require that every operation's justification should include all operations that were submitted more than Δ time units before the read (time bound) or that the total impact of the opera-

tions submitted before o but not in J_o justification, is to change the return value of o by less than Δ (value bound).

1.5 Transactions

Replication has been extensively studied in the database research community (see Ch 12). For this field, we need to consider not just separate operations, but also how these operations are combined into transactions. A transaction contains a set of the operations performed during an execution, and each operation belongs to exactly one transaction. Some systems require all operations of a transaction to be submitted by the same client; in other systems, a transaction may be distributed among several clients. Sometimes each operation, when it is submitted, carries with it the identifier of the transaction it is part of; in other systems, a client submits special operations that begin and complete a transaction; each operation is then part of the transaction whose begin operation most recently preceded the operation at that client.

There are two distinct outcomes for a transaction. The transaction may complete with a *commit*, or with an *abort*. The term *atomic* is used by the database community[6], to say that all the operations of a transaction are performed (if the transaction commits), or it is as if none of the operations are performed (if the transaction aborts). This can be achieved by the system implicitly undoing each operation of the transaction (in reverse chronological order) at the end of a transaction which aborts.

Strong consistency for a replicated database system means that the system looks like a single-site, unreplicated database. The sequential data type of a database has operations (usually given in the SQL language), these read and update sets of items, determined by predicates on the contents of those items, there are also operations to insert new items, and to delete items chosen by a predicate. This is a fairly sophisticated data type, compared to those usual in computer architecture or distributed computing. Many system optimizations depend on knowing when operations commute on the sequential type, but determining the commutativity between operations of a database takes care: for example, an insert may fail to commute with a read of a set of items, when the inserted item satisfies the predicate that selects the items being read.

Any database will have a mechanism for concurrency control, which provides *isolation* between the transactions. Isolation constrains the values observed in operations, and this is often done by delaying an operation of one transaction until other, conflicting, transactions have completed. The traditional criterion for correct concurrency control is serializability, so *one-copy serializability (1SR)* is a natural consistency model for replicated databases, first defined by Bernstein and Goodman [5]. A recent alternative isolation approach is called Snapshot Isolation (abbreviated as SI) [4]; this is provided by several widely-used platforms, and so *one-copy snapshot isolation (1SI)* is also considered by researchers, where a replicated system is indistinguishable from an unreplicated system with SI for concurrency control.

[6] Notice that this is a different meaning from atomic in the distributed systems community, where the term is used for a linearizable system or component.

In the 1SR model, for a given execution E of the replicated system, we can find a single history H, called a serial history: H is valid for the sequential data-type, H contains all the operations of the committed transactions of E (each paired with its return value), and in H, all the operations of a transaction occur together in the same order that these operations occurred in E. Typically one also demands *external consistency*: that the history H should respect the partial order between transactions that do not overlap in E. If we regard a transaction as a single super-operation, 1SR with external consistency is essentially the same as linearizability of the super-operations.

As for other systems, performance is greatly degraded by the requirements to keep strong consistency, and most commercial platforms provide weak consistency models when replicating data. A common approach is to designate a master copy of the data, where updates are done first; all the updates of a transaction then propagate lazily as a group from the master to other replicas. Thus any read that uses a replica may see an obsolete state of the data. The consistency model provided by a system design with master-copy and lazy propagation, can be expressed in the framework of justifications from Section 1.4: each read operation being justified by a sequence of transactions, with the additional restriction that between any two operations' justifications, one is a prefix of the other. However, different reads in a transaction may have different sets of transactions that justify them.

Many other sophisticated consistency models have been proposed for "extended transactions" which can share some data in managed ways, while being isolated on other ways. A powerful framework for presenting these definitions is ACTA [15].

1.6 Discussion

The operation-ordering style of definition is much more widely used than the ideal-system style. Operation ordering definitions are generally much more succinct, and they are good at suggesting many variant models by slight changes of the constraints. Ideal-system models, on the other hand, allow one to use the extensive theory of formal methods, when proving that a particular system design provides a given consistency model. Examples of such proofs are in [2, 7, 14].

For users of systems which offer weak consistency, a vital need is some way to know how to use the system without being inconvenienced by the lack of strong guarantees. For multiprocessor hardware, there are several results that assist with this; typically, each result says that if a program is written in a particular way, it can work correctly with hardware which provides a particular weak consistency model. For example, if the client programs are written to enclose any uses of common variable within acquire-release blocks, then an execution on release-consistent memory is equivalent to an execution on sequentially consistent memory. However, we do not yet have similar guidance for common weak models in distributed computing, such as eventual consistency with read your writes. As cloud computing platforms offer this type of consistency model, the lack of such guidance becomes a threat to sensible use of these platforms.

If we have an ideal-system style definition of a weak consistency model, this can support formal proof of results about correct use of a system. For example, one can give an abstracted model C of any client that is programmed in a certain way, and one can then consider the composition of such clients and the weak-consistency-ideal-system W. We need to prove that the composition of C and W refines the composition of C and a strong ideal-system S. Here, "refines" is a relationship between transition machines (often also called "satisfies"), where A refines B means that every execution of A is a possible execution of B.

Whichever style of consistency model one prefers, it is important for both users and system designers, to have a clear mutual understanding of the properties that are guaranteed for the clients, and those that are not.

1.7 Conclusion

We have shown how one can express the functionality provided by different replica management algorithms, in a way that hides internal detail and only depends on aspects that are observable by the clients. The same consistency model might be offered by a system using quorums, or by a system using a group communication infrastructure, and the clients will be portable from one system to the other. We have shown examples of strong consistency models (where there is a single history of the sequential data type that is compatible with all the observations made by clients in the replicated system's execution) and weaker models, where different clients, or even different operations of one client, have different justifications in terms of other operations which seem to have happened first. We have considered two styles for defining a consistency model, either using properties of the orderings of operations, or a state-transition model for an ideal system which can generate the required observations. In either style, however, we do not mention the actual implementation details of sites, replicas, and messages. Many of the consistency models include special operations that are used for synchronization between clients (some, such as transaction begin and commit operations, do nothing else and are treated as no-ops on the sequential data type, while in some models synchronization is an extra feature on some operations such as reads or writes that affect the sequential data type). Another axis on which models vary is the session properties, which enforce some amount of ordering between activities at a single client; an example is a requirement to keep the effect of all the clients' own earlier writes evident in any value seen by that client.

We hope that whenever another chapter of this book describes a replica management algorithm, the reader will find it useful to think about what functionality is being given, by identifying how the system fits into the framework of consistency models we have explained.

Acknowledgements We thank Shirley Goldrei for careful proofreading.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
2. Afek, Y., Brown, G., Merritt, M.: A lazy cache algorithm. In: SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, pp. 209–222. ACM Press, New York (1989)
3. Alonso, R., Barbará, D., Garcia-Molina, H., Abad, S.: Quasi-copies: Efficient data sharing for information retrieval systems. In: Schmidt, J.W., Missikoff, M., Ceri, S. (eds.) EDBT 1988. LNCS, vol. 303, pp. 443–468. Springer, Heidelberg (1988)
4. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of Data, pp. 1–10. ACM Press, New York (1995)
5. Bernstein, P.A., Goodman, N.: Serializability theory for replicated databases. J. Comput. Syst. Sci. 31(3), 355–374 (1985)
6. Demers, A.J., Greene, D.H., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H.E., Swinehart, D.C., Terry, D.B.: Epidemic algorithms for replicated database maintenance. In: Proc ACM Conference on Principles of Distributed Computing (PODC'87), pp. 1–12 (1987)
7. Gibbons, P.B., Merritt, M., Gharachorloo, K.: Proving sequential consistency of high-performance shared memories (extended abstract). In: SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures, pp. 292–303. ACM Press, New York (1991)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
9. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. 10(4), 360–391 (1992)
10. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programm. IEEE Trans. Comput. 28(9), 690–691 (1979)
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
12. Liskov, B., Zilles, S.: Specification techniques for data abstractions. SIGPLAN Not. 10(6), 72–87 (1975)
13. Manson, J., Pugh, W., Adve, S.V.: The java memory model. SIGPLAN Not. 40(1), 378–391 (2005)
14. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Trans. Comput. 48(2), 227–235 (1999)
15. Ramamritham, K., Chrysanthis, P.K.: A taxonomy of correctness criteria in database applications. The VLDB Journal 5(1), 85–97 (1996)
16. Saito, Y., Shapiro, M.: Optimistic replication. Comput. Surveys 37(1), 42–81 (2005)
17. Steinke, R.C., Nutt, G.J.: A unified theory of shared memory consistency. J. ACM 51(5), 800–849 (2004)
18. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems, pp. 140–150. IEEE Computer Society Press, Los Alamitos (1994)
19. Vogels, W.: Eventually consistent. Commun. ACM 52(1), 40–44 (2009)