Consistency in Distributed Systems

Sebastian Burckhardt(✉)

Microsoft Research, Redmond, USA
sburckha@microsoft.com
http://research.microsoft.com/people/sburckha/

Abstract. Data replication is a common technique for programming distributed systems, and is often important to achieve performance or reliability goals. Unfortunately, the replication of data can compromise its consistency, and thereby break programs that are unaware. In particular, in weakly consistent systems, programmers must assume some responsibility to properly deal with queries that return stale data, and to avoid state corruption under conflicting updates. The fundamental tension between performance (favoring weak consistency) and correctness (favoring strong consistency) is a recurring theme when designing concurrent and distributed systems, and is both practically relevant and of theoretical interest.

In this course, we investigate how to understand and formalize consistency guarantees, and how we can determine if a system implementation is correct with respect to such specifications. We start by examining consensus, a classic problem in distributed systems, and then proceed to study various specifications and implementations of eventually consistent systems.

As more and more developers write programs that execute on a virtualized cloud infrastructure, they find themselves confronted with the subtleties that have long been the hallmark of distributed systems research. Devising message protocols, reading and writing weakly consistent shared data, and handling failures are notoriously challenging, and are gaining relevance for a new generation of developers.

With this in mind, I devised this course to provide a mix of techniques and results that may prove either interesting, or useful, or both. In the first half, I am presenting well-known results and techniques from the area of distributed systems research, including:

- A beautiful, classic result: the impossibility of implementing consensus in the presence of silent crashes on an asynchronous system [7] (Sect. 2.5).
- An algorithm that shows how impossibility is relative, by "achieving the impossible" for all practical purposes: the PAXOS protocol [11] (Sect. 2.6).
- The machinery needed to present these topics: labeled transitions systems and asynchronous protocols (Sect. 2).

In the second half, I focus on the main topic, which are consistency models for shared data. This part includes:

B. Meyer and M. Nordio (Eds.): LASER 2013-2014, LNCS 8987, pp. 84–120, 2015.
DOI: 10.1007/978-3-319-28406-4_4

- A formalization of strong consistency (sequential consistency, linearizability) and a proof of the CAP theorem [1,8] (Sect. 3).
- A general examination and formalization of various models for eventual consistency, which decomposes sequential consistency and introduces the arbitration and visibility relations in its place (Sect. 4.1).
- Several example architectures for implementing various versions of sequential or eventual consistency (Sect. 4.2).

These lecture notes are not meant to serve as a transcript. Rather, their purpose is to complement the slides [2] used in the lectures by providing the technical depth and precision that is difficult to achieve in a lecture. Although the material is technically self-contained, I highly recommend that readers study the slides alongside these lecture notes, because the slides provide additional motivation and contain many more examples and visualizations (such as diagrams or animations) that bring the material to life.

Update: Since giving the original lectures at the LASER summer school, I have expanded and revised much of the material presented in Sects. 3 and 4. The result is now available as a short textbook [3] that provides a thorough introduction to commonly used consistency models and protocols.

1 Preliminaries

We introduce some basic mathematical notations for sets, sequences, and relations. We assume standard set notations for set. Note that we write $A \subseteq B$ to denote $\forall a \in A : a \in B$. In particular, the notation $A \subseteq B$ does neither imply nor rule out either $A = B$ or $A \neq B$. We let $\mathbb{N}$ be the set of all natural numbers (starting with number 1), and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The power set $\mathcal{P}(A)$ is the set of all subsets of A.

Sequences. Given a set A, we let A^* be the set of finite sequences (or "words") of elements of A, including the empty sequence which is denoted ϵ. We let $A^+ \subseteq A^*$ be the set of nonempty sequences of elements of A. Thus, $A^* = A^+ \cup \{\epsilon\}$. For two sequences $u, v \in A^*$, we write $u \cdot v$ to denote the concatenation (which is also in A^*). If $f : A \to B$ is a function, and $w \in A^*$ is a sequence, then we let $f(w) \in B^*$ be the sequence obtained by applying f to each element of w. Sometimes we write A^ω for the set of ω-infinite sequences of elements of A.

Multisets. A finite multiset m over some base set A is defined to be a function $m : A \to \mathbb{N}_0$ such that $m(a) = 0$ for almost all a (= all but finitely many). The idea is that we represent the multiset as the function that defines how many times each element of A is in the set. We let $\mathcal{M}(A)$ denote the set of all finite multisets over A. When convenient, we interpret an element a as the singleton multiset containing a. We use the following notations for typical operations on multisets (using a mix of symbols taken from set notations and vector notations), $\emptyset$ for the empty multiset (= the constant 0 function $\lambda a.0$), $m + m'$ for multiset union (meaning $\lambda a.m(a) + m'(a)$), $m \leq m'$ for multiset inclusion (meaning $\forall a \in A : m(a) \leq m'(a)$), $a \in m$ for multiset membership (meaning $m(a) \geq 1$), and $m - m'$ for multiset difference (meaning $\lambda a.max(0, m(a) - m'(a))$).

Relations. A binary relation r over A is a subset $\mathsf{r} \subseteq A \times A$. For $a, b \in A$, we use the notation $a \xrightarrow{\mathsf{r}} b$ to denote $(a, b) \in \mathsf{r}$, and the notation $\mathsf{r}(a)$ to denote $\{b \in A \mid a \xrightarrow{\mathsf{r}} b\}$. We generalize the latter to sets in the usual way, i.e. for $A' \subseteq A$, $\mathsf{r}(A') = \{b \in A \mid \exists a \in A' : a \xrightarrow{\mathsf{r}} b\}$. We use the notation r^{-1} to denote the inverse relation, i.e. $(a \xrightarrow{\mathsf{r}^{-1}} b) \Leftrightarrow (b \xrightarrow{\mathsf{r}} a)$. Therefore, $\mathsf{r}^{-1}(b) = \{a \in A \mid a \xrightarrow{\mathsf{r}} b\}$ (we use this notation frequently). Given two binary relations r, r' over A, we define the composition $\mathsf{r}; \mathsf{r}' = \{(a, c) \mid \exists b \in A : a \xrightarrow{\mathsf{r}} b \xrightarrow{\mathsf{r}'} c\}$. We let id_A be the identity relation over A, i.e. $(a \xrightarrow{\mathsf{id}_A} b) \Leftrightarrow (a = b)$. For $n \in \mathbb{N}_0$, We let A^n be the n-ary composition $A; A \ldots; A$, with $A^0 = \mathsf{id}_A$. We let $A^+ = \bigcup_{n \geq 1} A^n$ and $A^* = \bigcup_{n \geq 0} A^n$. For some subset $A' \subseteq A$, and a binary relation r over A, we let $\mathsf{r}|_{A'}$ be the binary relation over A' obtained by restricting r, meaning $\mathsf{r}|_{A'} = \mathsf{r} \cap (A' \times A')$.

Orders. A binary relation r over A is a *partial order* if for all $a, b, c \in A$:

- It is irreflexive: $a \not\xrightarrow{\mathsf{r}} a$
- It is transitive: $(a \xrightarrow{\mathsf{r}} b) \wedge (b \xrightarrow{\mathsf{r}} c) \Rightarrow (a \xrightarrow{\mathsf{r}} c)$

Note that partial orders are acyclic (if there were a cycle, transitivity would imply $a \rightarrow a$ for some a, contradicting irreflexivity). We often visualize partial orders as directed acyclic graphs. Moreover, in such drawings, we usually omit transitively implied edges, to avoid overloading the picture.

A partial order does not necessarily order all elements. In fact, that is precisely what distinguishes it from a total order: a partial order r over A is a *total order* if for all $a, b \in A$ such that $a \neq b$, either $a \xrightarrow{\mathsf{r}} b$ or $b \xrightarrow{\mathsf{r}} a$. All total orders are also partial orders.

Many authors define partial orders to be reflexive rather than irreflexive. We chose to define them as irreflexive, to keep them more similar to total orders, and to keep the definition more consistent with our favorite visualization, directed acyclic graphs, whose vertices never have self-loops.

This choice is only superficial and not a deep distinction: consider the familiar notations $<$ and $\leq$. Conceptually, they represent the same ordering relation, but one of them is reflexive, the other one is irreflexive. In fact, if r is a total or partial order, we sometimes write $a <_{\mathsf{r}} b$ to represent $a \xrightarrow{\mathsf{r}} b$, and $a \leq_{\mathsf{r}} b$ to represent $(a \xrightarrow{\mathsf{r}} b) \vee (a = b)$.

A total order can be used to sort a set. For some finite set $A' \subseteq A$ and a total order r over A, we let $A'.\mathsf{sort}(\mathsf{r}) \in A^*$ be the sequence obtained by sorting the elements of A' in ascending $<_{\mathsf{r}}$-order.

2 Models and Machines

To reason about protocols and consistency, we need terminology and notation that helps us to abstract from details. In particular, we need models for machines, and ways to characterize their behavior by stating and then proving or refuting their properties.

2.1 Labeled Transition Systems

Labeled transitions systems provide a useful formalization and terminology that applies to a wide range of machines.

Definition 1. *A* **labeled transition system** *is a tuple* $L = (\mathsf{Cnf}, \mathsf{Ini}, \mathsf{Act}, \rightarrow)$ *where*

- Cnf *is a set of system* configurations*, or system states.*
- $\mathsf{Ini} \subseteq \mathsf{Cnf}$ *is a set of initial states. These represent valid starting configurations of the system.*
- Act *is a set of action labels.*
- $\rightarrow \subset (\mathsf{Cnf} \times \mathsf{Act} \times \mathsf{Cnf})$ *is a ternary* transition relation*. We write* $x \xrightarrow{a} y$ *to denote* $(x, a, y) \in \rightarrow$*.*

When using an LTS to model a system, a configuration represents a global snapshot of the state of every component of the system. Actions are abstractions that can model a number of activities, such as sending or receiving of messages, interacting with a user, doing some internal processing, or combinations thereof. Labeled transition systems are often visualized using labeled graphs, with vertices representing the states and labeled edges representing the actions.

We say an action $a \in \mathsf{Act}$ is *enabled* in state $s \in \mathsf{Cnf}$ if there exists a $s' \in \mathsf{Cnf}$ such that $s \xrightarrow{a} s'$. More than one action can be enabled in a state, and in general, an action can lead to more than one successor state. We say an action a is *deterministic* if that is never the case, that is, if for all $s \in \mathsf{Cnf}$, there is at most one $s' \in S$ such that $s \xrightarrow{a} s'$.

Defining an LTS to represent a concurrent system helps us to reason precisely about its executions and their correctness. An *execution fragment* E is a (finite or infinite) alternating sequence of states and actions:

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \dots$$

and an *execution* is an execution fragment that starts in an initial state. We formalize these definitions as follows.

Definition 2. *Given some LTS* $L = (\mathsf{Cnf}, \mathsf{Ini}, \mathsf{Act}, \rightarrow)$*, an* **execution fragment** *for* L *is a tuple* $E = (\mathsf{len}, \mathsf{cnf}, \mathsf{act})$ *where*

$$\begin{aligned} &\mathsf{len} \in (\mathbb{N}_0 \cup \infty) && \textit{(the length)} \\ &\mathsf{cnf} : \{0 \dots \mathsf{len}\} \to \mathsf{Cnf} && \textit{(the configurations)} \\ &\mathsf{act} : \{1 \dots \mathsf{len}\} \to \mathsf{Act} && \textit{(the actions)} \end{aligned}$$

such that for all $1 \leq i \leq \mathsf{len}$*, we have* $\mathsf{cnf}(i-1) \xrightarrow{\mathsf{act}(i)} \mathsf{cnf}(i)$*. An* **execution** *is an execution fragment* E *satisfying* $E.\mathsf{cnf}(0) \in \mathsf{Ini}$*.*

We define $\mathsf{pre}(E) = E.\mathsf{cnf}(0)$ and $\mathsf{post}(E) = E.\mathsf{cnf}(E.\mathsf{len})$ (we write $\mathsf{post}(E) = \perp$ if $E.\mathsf{len} = \infty$). Two execution fragments E_1, E_2 can be concatenated to form another execution fragment $E_1 \cdot E_2$ if $E_1.\mathsf{len} \neq \infty$ and $\mathsf{post}(E_1) = \mathsf{pre}(E_2)$.

We say a configuration $c \in \mathsf{Cnf}$ is *reachable from* a configuration $c' \in \mathsf{Cnf}$ if there exists an execution fragment E such that $c' = \mathsf{pre}(E)$ and $c = \mathsf{post}(E)$. We say a configuration $c \in \mathsf{Cnf}$ is *reachable* if it is reachable from an initial configuration.

Reasoning about executions usually involves reasoning about *events*. An event is an occurrence of an action (the same action can occur several times in an execution, each being a separate event). Technically, we define the events of an execution fragment E to be the set of numbers $\mathsf{Evt}(E) = \{1, 2, \ldots, E.\mathsf{len}\}$. Then, for events $e, e' \in \mathsf{Evt}(E)$, $e < e'$ means e occurs before e' in the execution, and $E.\mathsf{act}(e)$ is the action of event e.

Given an execution fragment E of an LTS L, we let $\mathsf{trc}(E) \in (L.\mathsf{Act}^* \cup L.\mathsf{Act}^\omega)$ be the (finite of infinite) sequence of actions in E, called the *trace* of E. If all actions of L are deterministic, then E is completely determined by $E.\mathsf{pre}$ and $E.\mathsf{trc}$. For that reason, traces are sometimes called schedules.

In our proofs, we often need to take an existing execution, and modify it slightly by reordering certain actions. Given a configuration c and a deterministic action a, we write $\mathsf{post}(c, a)$ to be the uniquely determined c' satisfying $c \xrightarrow{a} c'$, or $\perp$ if it is not possible (because a is not enabled in c). Similarly, we write $\mathsf{post}(c, w)$, for an action sequence $w \in A^*$, to denote the state reached from c by performing the actions in w, or $\perp$ if not possible. In the remainder of this text, all of our LTS are constructed in such a way that all actions are deterministic.

Working with deterministic actions can have practical advantages. For testing and debugging protocols, we often need to analyze or reproduce failures based on partial information about the execution, such as a trace log. If the log contains the sequence of actions in the order they happened, and if the actions are deterministic, it means that the log contains sufficient information to fully reproduce the execution.

2.2 Asynchronous Message Protocols

An LTS can express many different kinds of concurrent systems, but we care mostly about message passing protocols in this context. Therefore, we specialize the general LTS definition above to define such systems. Throughout this text, we assume that Pid is a set of process identifiers (possibly infinite, to model dynamic creation). Furthermore, we assume that there is a total order defined on the process identifiers Pid. For example, $\mathsf{Pid} = \mathbb{N}$.

Definition 3. *A* **protocol definition** *is a tuple*

$$\Phi = (\mathsf{Pst}, \mathsf{Msg}, \mathsf{Act}, \mathsf{ini}, \mathsf{ori}, \mathsf{dst}, \mathsf{pid}, \mathsf{cnd}, \mathsf{rcv}, \mathsf{snd}, \mathsf{upd})$$

where

– Pst *is a set of process states, with a function*

$$\mathsf{ini} : \mathsf{Pid} \to \mathcal{P}(\mathsf{Pst}) \quad \textit{(initial states)}$$

– Msg *is a set of messages, with properties*

$$\begin{array}{l}\mathsf{ori} : \mathsf{Msg} \to \mathsf{Pid} \quad \textit{(the origin)}\\ \mathsf{dst} : \mathsf{Msg} \to \mathsf{Pid} \quad \textit{(the destination)}\end{array}$$

– Act *is a set of actions, with properties*

$$\begin{array}{ll}\mathsf{pid} : \mathsf{Act} \to \mathsf{Pid} & \textit{(the process)}\\ \mathsf{cnd} : \mathsf{Act} \to \mathcal{P}(\mathsf{Pst}) & \textit{(the condition or guard)}\\ \mathsf{rcv} : \mathsf{Act} \to \bot \cup \mathsf{Msg} & \textit{(received message, if any)}\\ \mathsf{snd} : \mathsf{Act} \times \mathsf{Pst} \to \mathcal{M}(\mathsf{Msg}) & \textit{(sent messages)}\\ \mathsf{upd} : \mathsf{Act} \times \mathsf{Pst} \to \mathsf{Pst} & \textit{(process state update)}\end{array}$$

– *the message received by an action targets the same process:*

$$\forall a \in \mathsf{Act} : (\mathsf{rcv}(a) \neq \bot) \Rightarrow (\mathsf{dst}(\mathsf{rcv}(a)) = \mathsf{pid}(a)).$$

– *only finitely many actions apply at a time:*

$$\forall s \in \mathsf{Pst} : \forall m \in (\bot \cup \mathsf{Msg}) : |\{a \in \mathsf{Act} \mid (\mathsf{cnd}(a) \in s) \wedge (\mathsf{rcv}(a) = m)\}| < \infty.$$

We call actions a that receive no message (i.e. $\mathsf{rcv}(a) = \bot$) *spontaneous*. For convenience, given a protocol definition Φ, we write $\Phi.\mathsf{Pst}$, $\Phi.\mathsf{Msg}$, etc. to denote its components.

Definition 4. *Given a protocol definition Φ as above, we construct a corresponding labeled transition system $L_\Phi = (\mathsf{Cnf}_\Phi, \mathsf{Ini}_\Phi, \mathsf{Act}_\Phi, \to_\Phi)$ as follows:*

– Configurations: $\mathsf{Cnf}_\Phi = (\mathsf{Pid} \to \Phi.\mathsf{Pst}) \times \mathcal{M}(\Phi.\mathsf{Msg})$. The meaning is that each configuration is a pair (P, M) with P being a function that maps each process identifier to the current state of that process, and M being a multiset that represents messages that are currently "in flight". For a configuration c, we write $c.P$ and $c.M$ to denote its components.
– Actions: $\mathsf{Act}_\Phi = \Phi.\mathsf{Act}$.
– Initial states: $\mathsf{Ini}_\Phi = \{(P, \emptyset) \mid \forall p \in \mathsf{Pid} : P(p) \in \Phi.\mathsf{ini}(p)\}$
– Transition Relation: define $\to_\Phi$ such that $(P, M) \xrightarrow{a}_\Phi (P', M')$ iff all of the following conditions hold:
  1. the guard is satisfied: $P(\Phi.\mathsf{pid}(a)) \in \Phi.\mathsf{cnd}(a)$
  2. the received message (if any) is removed: either $\Phi.\mathsf{rcv}(a) = \bot$ and $M' = M$, or $\Phi.\mathsf{rcv}(a) \in M$ and $M' = M - \Phi.\mathsf{rcv}(a)$
  3. the sent messages are added to the message pool: $M' = M + \Phi.\mathsf{snd}(a)$
  4. the local state is updated, all other states remain the same:

$$\forall p \in \mathsf{Pid} : \quad P'(p) = \begin{cases} \Phi.\mathsf{upd}(a, P(p)) & \text{if } p = \Phi.\mathsf{pid}(a) \\ P(p) & \text{otherwise} \end{cases}$$

When reasoning about an execution E of L_Φ, we define the following notational shortcut: $E_{p,i} = E.\mathsf{cnf}(i).P(p)$.

process state
| `preference` : $\{0,1\}$; // initially one of $\{0,1\}$
| `decision` : $\{\bot,0,1\}$; // initially $\bot$
messages
| Proposal(p : Pid, $b : \{0,1\}$) //sent from p to l
| Announcement(q : Pid, $b : \{0,1\}$) //sent from l to q
action propose(p : Pid) **at** p
| **sends** Proposal(p, `preference`)
action announce(p : Pid, $b : \{0,1\}$) **at** l
| **receives** Proposal(p, b)
| **condition** `decision` $= \bot$
| **sends** $\sum_{q \in \mathsf{Pid}}$ Announcement(q, b)
| **updates** `decision` $\leftarrow b$
action learn(q : Pid, $b : \{0,1\}$) **at** q
| **receives** Announcement(p, b)
| **updates** `decision` $\leftarrow b$

Fig. 1. Example strawman protocol for a leader-based consensus, with a fixed leader $l \in \mathsf{Pid}$.

Example. Consider a simple protocol where the processes try to reach consensus on a single bit. We assume that the initial state of each process contains the bit value it is going to propose. We can implement a simple leader-based protocol to reach consensus by fixing some leader process $l \in \mathsf{Pid}$. The idea is based on a "race to the leader", which works in three stages: (1) each process sends a message containing the bit value it is proposing to the leader, (2) the leader, upon receiving any message, announces this value to all other processes, and (3) upon receiving the announced message, each recipient decides on that value.

We show how to write pseudocode for this protocol in Fig. 1. Our notation is somewhere between pseudocode and formulae (see Fig. 1). It defines all the components of Φ listed in Definition 3 in several sections with the following meanings:

- In the **process state** section, we define the set Pst_Φ and the initial state function ini_Φ. The process state is expressed as a product of several named typed variables, and we show the initial value of each variable in the comment at the end of each line.
- In the **messages** section, we define the set Msg and the functions ori and dst. Each message has a name and several named typed parameters. We show how the functions ori and dst (which determine the origin and destination of each message) are defined in the comment at the end of each line.
- The remaining sections define the actions, with one section per action. The entries have the following meaning:
  - The first line of each **action** section defines the action label, which is a name together with named typed parameters. All action labels together constitute the set Act. The comment at the end of the line defines the pid function, which determines the process to which this action belongs.

- The **receives** section defines the rcv function. If there is a receives line present, it defines the message that is received by this action, and if there is no receives line, it specifies that this action is spontaneous.
- The **sends** section defines the snd function. It specifies the message, or the multiset of messages, to be sent by this action. We use the multiset notations as described in Sect. 1, in particular, the sum symbol is used to describe a collection of messages. We omit this section if no messages are sent.
- The **condition** section defines the cnd function, representing a condition that is necessary for this action to be performed. It describes a predicate over the local process state (i.e. over the variables defined in the process state section). We omit this section if the action is unconditional.
- The **updates** section defines the upd function, by specifying how to update the local process state. We omit this section if the process state is not changed.

One could conceivably formalize these definitions and produce a practically usable programming language for protocols; in fact, this has already been done for the programming language used by the Murϕ tool [6], an explicit-state model checker that is suitable for model checking protocols defined in this style, and which inspired our pseudocode formalization.

Consider the consensus protocol shown in Fig. 1. Is this a good protocol? Not really. It's not all that bad: we shall see that it is actually a correct consensus in the absence of failures, and it works even if there are crash failures as long as only non-leader processes fail. However, it is susceptible to leader failures. Also, it has some oddities: participants can keep sending inordinate numbers of propose messages. The decision value is written twice on the leader. Perhaps worst: the protocol is more complicated than necessary. The leader could just send its own proposal immediately to everyone.

2.3 Consensus Protocols

What makes a protocol a consensus protocol? Somehow, we start out with a bit on each participant describing its preference. When the protocol is done, everyone should agree on some bit value that was one of the proposed values. And, there should be progress eventually, i.e. the protocol should terminate with a decision.

We now formalize what we mean by a consensus protocol, by adding functions to formalize the notions of initial preference and of decisions.

Definition 5. *A consensus protocol is a tuple*

$$(\mathsf{Pst}, \mathsf{Msg}, \mathsf{Act}, \mathsf{ini}, \mathsf{ori}, \mathsf{dst}, \mathsf{pid}, \mathsf{cnd}, \mathsf{rcv}, \mathsf{snd}, \mathsf{upd}, \mathsf{pref}, \mathsf{dec})$$

such that

– $(\mathsf{Pst}, \ldots, \mathsf{upd})$ *is a protocol.*

- pref *is a function* $\mathsf{Pid} \times \{0,1\} \to \mathsf{Pst}$ *with the following meaning:* $\mathsf{pref}(p,b)$ *is the initial process state to be used for a process whose initial preference is* b*. We require that for all* p*,* $\mathsf{ini}(p) = \{\mathsf{pref}(p,0), \mathsf{pref}(p,1)\}$*.*
- dec *is a function* $\mathsf{Pst} \to \{\bot, 0, 1\}$*; For a process state* s*,* $\mathsf{dec}(s) = \bot$ *means no decision has been reached, otherwise* $\mathsf{dec}(s)$ *is the decision that has been reached.*

For example, for the strawman protocol, we define $\mathsf{pref}(p,b).\texttt{preference} = b$ and $\mathsf{pref}(p,b).\texttt{decision} = \bot$, and we define $\mathsf{dec}(s) = s.\texttt{decision}$.

Next, we formalize the correctness conditions we briefly outlined at the beginning of this section, and then examine if they hold for our strawman. For an execution E, we define the following properties:

1. **Stability.** If a value is decided at a process p, it remains decided forever:
$$\forall p \in \mathsf{Pid} : \forall i < E.\mathsf{len} : (\mathsf{dec}(E_{p,i}) \neq \mathsf{dec}(E_{p,i+1})) \Rightarrow (\mathsf{dec}(E_{p,i}) = \bot)$$
2. **Agreement.** No two processes should decide differently:
$$\{0,1\} \not\subseteq \{\mathsf{dec}(E_{p,i}) \mid i \leq E.\mathsf{len} \text{ and } p \in \mathsf{Pid}\}$$
3. **Validity.** If a value is decided, this value must match the preference of at least one of the processes:
$$\{\mathsf{dec}(E_{p,i}) \mid i \leq E.\mathsf{len} \text{ and } p \in \mathsf{Pid}\} \subseteq \{\bot\} \cup \{b \mid \exists p : \mathsf{pref}(p,b) = E_{p,0}\}$$
4. **Termination.** Eventually, a decision is reached on all correct[1] processes:
$$\forall p \in (\mathsf{Pid} \setminus F) : \{0,1\} \cap \{\mathsf{dec}(E_{p,i}) \mid i \leq E.\mathsf{len}\} \neq \emptyset$$

Does our strawman protocol satisfy all of these properties, for all of its executions? Certainly, this is true for the first three.

1. **Strawman satisfies agreement and stability.** There can be at most one announce event, because only the leader can perform the announce action, and the leader sets the `decided` variable to true after doing the announce, which prevents further announce actions. Therefore, all decide actions must receive a Announcement message sent by the same announce event, thus all the actions that write a decision value write the same value. Decision values are stable: there is no action that writes $\bot$ to the decision variable.
2. **Strawman satisfies validity.** Any announce event (for some bit b) receives a Proposal message that must have originated in some propose event (with the same bit b), which has as a precondition that the variable $\texttt{proposal} = b$. Thus, b matches the preference of that process.

Termination is however not satisfied for all executions. For example, in an execution of length 0, no decision is reached. Perhaps it would be more reasonable to restrict our attention to complete executions:

[1] We talk more about failures later. For now, just assume that the set F of faulty processes is empty.

Definition 6. *An execution fragment E is* complete *if it is either infinite or terminated, i.e. if either $E.\mathsf{len} = \infty$, or if no actions are enabled in $E.\mathsf{post}$.*

Does the strawman satisfy termination on all complete executions? The answer is again no. For example, consider an initial configuration where the preference of process p is 0. Then we can have an infinite execution

$$\mathsf{propose}(p, 0)\ \mathsf{propose}(p, 0)\ \mathsf{propose}(p, 0)\ \mathsf{propose}(p, 0)\ \ldots$$

Clearly, no progress is made and an unbounded number of messages is sent. No decision is reached.

Still, it appears that this criticism is not fair! It is hard to imagine how *any* protocol can achieve termination unless the transport layer and the process scheduler cooperate. Clearly, if the system simply does not deliver messages, or never executes actions even though they are enabled, nothing good can happen. We need *fairness*: some assumptions about the "minimal level of service" we may expect.

Informally, what we want to require is that messages are eventually delivered unless they become undeliverable, and that spontaneous actions are eventually performed unless they become disabled. We say an action $a \in \mathsf{Act}$ *receives* message $m \in \mathsf{Msg}$ if $\mathsf{rcv}(a) = m$. We say $m \in \mathsf{Msg}$ is *receivable* in a configuration s if there exists an action a that is enabled and that receives m.

Definition 7. *A message m is* **neglected** *by an execution E if it is receivable in infinitely many configurations, but received by only finitely many actions. A spontaneous action a is* **neglected** *by an execution E, if it is enabled in infinitely many configurations, but performed only finitely many times.*

Definition 8. *An execution E of some protocol Φ is* **fair** *if it does not neglect any messages or spontaneous actions.*

Definition 9. *A consensus protocol is a* **correct consensus protocol** *if all fair complete executions satisfy stability, agreement, validity, and termination.*

Strawman is Correct. We already discussed agreement and validity. Termination is also satisfied for fair executions, for the following reasons. Because the propose action is always enabled for all p, it must happen at least once (in fact, it will happen infinitely many times for all p). After it happens just once, announce is now enabled, and remains enabled forever if announce does not happen. Thus announce must happen (otherwise fairness is violated). But now, for each q, decide is enabled, and thus must happen eventually.

Fair Schedulers. The definition of fairness is purposefully quite general; it does not describe how exactly a scheduler is guaranteeing fairness. However, it is useful to consider how to construct a scheduler that guarantees fairness. One way to do so is to schedule an action that has maximal seniority, in the sense that it is executing a spontaneous action or receiving a message that has been waiting (i.e. been enabled/receivable but not executed/received) the longest:

Definition 10. *Let Φ be a protocol, let E be a finite execution of L_Φ, and let $a \in \mathsf{Act}_\Phi$ be an action that is enabled in $\mathsf{post}(E)$. Then, we define the* **seniority** *of a to be the maximal number k such that either (1) some message m in $\mathsf{rcv}(a)$ is receivable in $E.\mathsf{cnf}(E.\mathsf{len}-k)$ but has not been received by any action $E.\mathsf{act}(j)$ where $E.(E.\mathsf{len}-k) < j \leq E.\mathsf{len}$, or (2) a is a spontaneous action that is enabled in $E.\mathsf{cnf}(E.\mathsf{len}-k)$ but is not equal to any $E.\mathsf{act}(j)$ where $(E.\mathsf{len}-k) < j \leq E.\mathsf{len}$.*

Lemma 1. *If a scheduler always picks the most senior enabled action, the resulting schedule is fair.*

Proof. Assume to the contrary that there exists an execution that is not fair, that is, neglects a message or spontaneous action.

First, consider that a message m is neglected. This means that the message is receivable infinitely often, but received only finitely many times. Consider the first configuration where it is receivable after the last time it is received, say $E.\mathsf{cnf}(k)$. Since m is receivable in infinitely many configurations $\{E.\mathsf{cnf}(k') \mid k' > k\}$ but never received, there must be infinitely many configurations $\{E.\mathsf{cnf}(k') \mid k' > k\}$ where some enabled action is more senior than the one that receives m (otherwise the scheduler would pick that one). However, an action can only be more senior than the one that receives m if it is either receiving some message that has been waiting (i.e. has been receivable without being received) at least as long as m, or a spontaneous action that has been waiting (i.e. has been enabled without being performed) at least as long as m. But there can only be finitely many such messages or spontaneous actions, since there are only finitely many configurations $\{E.\mathsf{cnf}(j) \mid j \leq k\}$, and each such configuration has only finitely many receivable messages and enabled spontaneous actions, by the last condition in Definition 3; thus we have a contradiction.

Now, consider that a spontaneous action is neglected. We get a contradiction by the same reasoning. □

Independence. The notion of *independence* of actions and schedules is also often useful. We can define independence for general labeled transition systems as follows:

Definition 11. *Let $L = (S, I, \mathsf{Act}, \rightarrow)$ be a LTS. Two actions $a, a' \in L$ are called* independent *if for all configurations $c \in \mathsf{Cnf}$ in which both a and a' are enabled, the following conditions are true:*

- *They do not disable each other: a is enabled in $\mathsf{post}(c, a')$ and a' is enabled in $\mathsf{post}(c, a)$.*
- *Their effect commutes: $\mathsf{post}(c, a \cdot a') = \mathsf{post}(c, a' \cdot a)$.*

For protocols, actions performed by different nodes are independent. This is because executing an action for process p can only remove messages destined for p from the message pool, it can thus not disable any actions on any other process. Actions by different processes always commute, because their effect on the local state targets local states by different processes, and their effects on the message pool commute.

We call two schedules $s, s' \in \mathsf{Act}^*$ independent if for all $a \in s$ and $a' \in s'$, a and a' are independent. Note that if two schedules s, s' are independent and possible in some configuration c, then $\mathsf{post}(c, s \cdot s') = \mathsf{post}(c, s' \cdot s)$. Visually, this can be seen by doing a typical tiling argument.

2.4 Failures

As we probably all know from experience, failures are common in distributed systems. Failures can originate in the transport layer (a logical abstraction of the network, including switches, links, proxies, etc.) or the nodes (computers running the protocol software). Sometimes, the distinction is not that clear (for example, messages that are waiting in buffers are conceptually in the transport layer, but are subject to loss if the node fails).

We now show how, given a protocol Φ and its LTS as defined in Sect. 2.2, Definition 3, we can model failures by adding failure actions to the LTS defined in Definition 4.

Modeling Transport Failures. Failures for message delivery often include (1) reordering, (2) loss, (3) duplication, and (4) injection of messages. In our protocol model, reorderings are already allowed, thus we do not consider them to be a failure. To model message loss, we can add the following action to the LTS:

$$\mathsf{Act}_\Phi^{\mathsf{lose}} = \mathsf{Act}_\Phi \cup \{\mathsf{lose}(m) \mid m \in \mathsf{Msg}\}$$
$$(P, M) \xrightarrow{\mathsf{lose}(m)} (P', M') \Leftrightarrow ((P = P') \wedge (m \in M) \wedge (M' = M - m))$$

Similarly, we can add an action for message duplication:

$$\mathsf{Act}_\Phi^{\mathsf{duplicate}} = \mathsf{Act}_\Phi \cup \{\mathsf{duplicate}(m) \mid m \in \mathsf{Msg}\}$$
$$(P, M) \xrightarrow{\mathsf{duplicate}(m)} (P', M') \Leftrightarrow ((P = P') \wedge (m \in M) \wedge (M' = M + m))$$

We can also model injection of arbitrary messages:

$$\mathsf{Act}_\Phi^{\mathsf{invent}} = \mathsf{Act}_\Phi \cup \{\mathsf{invent}(m) \mid m \in \mathsf{Msg}\}$$
$$(P, M) \xrightarrow{\mathsf{invent}(m)} (P', M') \Leftrightarrow ((P = P') \wedge (M' = M + m))$$

However, we will not talk more about the latter, which is considered a *byzantine* failure, and which opens up a whole new category of challenges and results.

Masking Transport Failures. Protocols can mask message reordering, loss, and duplication by affixing sequence numbers to messages, and using send and receive buffers. Receivers can detect missing messages in the sequence and re-request them. In fact, socket protocols (such as TCP) use this type of mechanism (e.g. sliding window) to achieve reliable in-order delivery of a byte stream. In practice, however, just using TCP is not always good enough, because TCP

connections can themselves fail. Often, resilience against transport failures needs to be built into the protocol in some form.

A common trick to tolerate message duplication in services is to design the service calls to be *idempotent*, meaning that executing a message twice has the same effect as executing it just once. For example, setting the value of some parameter twice is harmless. Properly written REST protocols use the verb PUT to mark such requests as idempotent, allowing browsers and proxies to duplicate them.

Modeling Node Failures. Typical node failures considered by protocol designers are *crash failures* (a process permanently stops at some point), and *crash-recovery* failures (a process stops at some point, then recovers later). Sometimes, byzantine failures are also considered, where faulty nodes exhibit arbitrary behavior, but we are skipping that topic. Typical terminology is to call a process correct if it does never experience a crash failure, and if it encounters only finitely many crash-recovery failures. We let $F \subset \mathsf{Pid}$ be the subset of faulty processes, i.e. processes that *may* be incorrect (it is acceptable for processes in F to be actually correct in any given execution).

In a crash failure, the process state is permanently lost, and the process never takes another action. In a crash-recovery failure, the process can recover some or all of its state from some form of durable storage (if it cannot, there is little reason for a process to continue under the same identity). The part of the state that is lost in crashes is called "soft state". Often, message buffers are soft state, thus it is possible that messages are lost or duplicated if the crash occurred during a transition that receives or sends messages.

In asynchronous systems, it is often important to distinguish between *silent* crashes and *noisy* crashes. Silent crashes mean that other processes have no way to distinguish between a slow response and a crashed process, which can be a real problem as we shall see below. Noisy crashes mean that other processes can use *failure detectors* to get information about whether a crash occurred. In some situations (e.g. inside a data center), it is often quite feasible to build failure detectors, in particular approximate failure detectors, and they can be very helpful for designing protocols. However, in other situations failure detection is impossible. For example, if a server loses contact to a JavaScript app running in somebody's browser, it does not know if this was a temporary connection failure and the app will reconnect at some future time, or if the user has closed the browser and will never return.

In the following, we consider only silent crash failures. To model them, we use a modified definition of fairness: we allow executions to be 'unfair' if this unfairness is consistent with processes crashing, in the sense that crashed processes perform no more actions and receive no more messages after they crash.

Definition 12. *An execution E of L_Φ for some Φ is a* **complete F-fair execution** *if there exists a partial function* $\mathsf{fails} : F \to \bot \cup \{0 \ldots E.\mathsf{len}\}$ *such that*

- *Crashed processes take no steps after they crash: If* $\mathsf{fails}(p) \neq \bot$ *for some* p*, then* $\mathsf{pid}(E.\mathsf{act}(j)) \neq p$ *for all* $j > \mathsf{fails}(p)$*.*
- E *is complete: either* $E.\mathsf{len} = \infty$*, or for all actions* a *that are enabled in* $\mathsf{post}(E)$*,* $\mathsf{fails}(\mathsf{pid}(a)) \neq \bot$*.*
- E *is fair for correct processes: it does not neglect any spontaneous actions* a *except if* $\mathsf{fails}(\mathsf{pid}(a)) \neq \bot$*, and it does not neglect any messages* m *except if* $\mathsf{fails}(\mathsf{dst}(m)) \neq \bot$*.*

2.5 Asynchronous Consensus Under Silent Crash Failures is Impossible

We now show the famous impossibility result for asynchronous consensus protocols under just 1 silent crash failure, following the same proof structure as in Fischer, Lynch and Paterson [7]. Their proof assumes a limited form of protocol where for each process, there is exactly one receive action per message, exactly one spontaneous action, and the actions do not have conditions. We first prove the theorem under the same limitation, and then show how to generalize it to the more general protocols defined above.

Definition 13. *A* **simple consensus protocol** *is a consensus protocol*

$$(\mathsf{Pst}, \mathsf{Msg}, \mathsf{Act}, \mathsf{ini}, \mathsf{ori}, \mathsf{dst}, \mathsf{pid}, \mathsf{cnd}, \mathsf{rcv}, \mathsf{snd}, \mathsf{upd}, \mathsf{pref}, \mathsf{dec})$$

such that the only actions are:

$$\mathsf{Act} = \{\mathsf{receive}(p, m) \mid p \in \mathsf{Pid}, m \in \mathsf{Msg}\} \cup \{\mathsf{run}(p) \mid p \in \mathsf{Pid}\},$$

and such that:

$$\mathsf{rcv}(\mathsf{receive}(p, m)) = m \quad \mathsf{rcv}(\mathsf{run}(p)) = \bot \quad \mathsf{pid}(\mathsf{receive}(p, m)) = \mathsf{pid}(\mathsf{run}(p)) = p$$

and where the actions have no guard:

$$\mathsf{cnd}(\mathsf{receive}(p, m)) = \mathsf{cnd}(\mathsf{run}(p)) = \mathsf{Pst}.$$

Theorem 1. *Let* Φ *be a simple consensus protocol and let* Pid *contain at least two processes. Then,* Φ *is not correct in the presence of silent crash failures: in particular, its labeled transition system* $L_\Phi = (\mathsf{Cnf}_\Phi, \mathsf{Ini}_\Phi, \mathsf{Act}_\Phi, \rightarrow_\Phi)$ *has a complete* F*-fair execution that violates either validity, agreement, stability, or termination, and where* $|F| = 1$*.*

Proof. Assume to the contrary that all F-fair executions with $|F| \leq 1$ satisfy validity, agreement, stability, and termination. We then prove (using a sequence of lemmas) that a contradiction results.

The key to the proof is the idea of examining the *valence* of system configuration, meaning how many different decisions are possible when starting in that configuration. For a system configuration $c \in \mathsf{Cnf}_\Phi$, we define $V(c) \subseteq \mathsf{Cnf}_\Phi$ to be the set of decisions reachable from c:

$$V(c) = \{\mathsf{dec}(c'.P(p)) \mid c' \text{ reachable from } c \text{ and } p \in \mathsf{Pid}\} \setminus \{\bot\}$$

Since we assume that the protocol is correct, in particular, terminating, we know that $|V(c)| \geq 1$ for all reachable configurations c. We call a configuration *bivalent* if $|V(c)| = 2$, *univalent* if $|V(c)| = 1$, *0-valent* if $V(c) = \{0\}$, and *1-valent* if $V(c) = \{1\}$.

Lemma 2. *Φ has a bivalent initial configuration.*

Proof. Assume not; then all configurations are univalent. For $b \in \{0, 1\}$, let c_b be the initial configuration where all processes have preference b. Because the protocol satisfies termination and validity, it must be true for both choices of $b \in \{0, 1\}$ that $b \in V(c_b)$, and thus that c_b is b-valent. Let us call two initial configurations c, c' *adjacent* if they differ only in the initial value of a single process, i.e. iff $c.P(p) = c'.P(p)$ for all but one $p \in \mathsf{Pid}$. Since c_0 must be connected to c_1 by a chain of adjacent configurations, there must exist adjacent initial configurations c, c' such that c is 0-valent and c' is 1-valent. Let p be the process on which c, c' differ. Now, run a $\{p\}$-fair scheduler that schedules actions fairly, except that p takes no steps at all. Since p takes no steps, the initial state of p cannot influence the outcome, thus we can run the same schedule with the same outcome on both c and c', contradicting the assumption that c is 0-valent and c' is 1-valent.

Lemma 3. *Let c be a bivalent configuration, and let a be an action that is enabled in c. Then there exists an action sequence $w \in \mathsf{Act}^*$ such that $\mathsf{Exec}(c, w \cdot a).\mathsf{post}$ is a bivalent configuration.*

Proof. For the given c and a, let $C(c, a) \subseteq \mathsf{Cnf}$ be the set of configurations that are reachable from c without performing the action a. Note that a must be enabled in all configurations in $C(c, a)$, since it is either a $\mathsf{receive}$ operation (which stays enabled until it is performed, no matter what other actions are performed meanwhile), or a run operation (which is always enabled). Let $D(c, a)$ be the set of configurations reachable from a configuration in $C(c, a)$ by performing a. If $D(c, a)$ contains a bivalent configuration, we are done. Otherwise, we assume $D(c, a)$ contains only univalent configurations and proceed to provide a contradiction.

First, let's find two configurations c_0, c_1 in $C(c, a)$ such that $c_0 \xrightarrow{a'} c_1$ for some $a' \neq a$, and such that the respective a-successors $d_0 = \mathsf{post}(c_0, a)$ and $d_1 = \mathsf{post}(c_1, a)$ (which are both in $D(c, a)$ and are thus both univalent) have different valence.

- Consider $\mathsf{post}(c, a)$. Since it is in $D(c, a)$, it must be univalent, say b-valent.
- Since c is bivalent, it must be possible to reach a $(1 - b)$-valent configuration c' from c. Let c'' be the last configuration on this path that is still in $C(c, a)$. Then, $x = \mathsf{post}(c'', a)$ must be $(1 - b)$-valent as well: either $c'' = c'$, in which case x is a successor of the $(1-b)$-valent configuration c' and thus also $(1-b)$-valent, or $c'' \neq c'$, in which case x is a univalent conf (because it is in $D(c, a)$) from which a $(1 - b)$-valent configuration (c') can be reached, thus x is also $(1 - b)$-valent.

– Since we have a path from c to c'' entirely within $C(c, a)$, and where $\mathsf{post}(c, a)$ has different valence than $\mathsf{post}(c'', a)$, there must exist c_0, c_1 as claimed.

Now, distinguish cases.

1. If $\mathsf{pid}(a') \neq \mathsf{pid}(a)$, then a and a' are independent actions, thus $d_0 = \mathsf{post}(c_0, a) = \mathsf{post}(c_1, a) = d_1$ which is impossible because d_i are both 1-valent with different valence.
2. If $\mathsf{pid}(a') = \mathsf{pid}(a) = p$ for some $p \in \mathsf{Pid}$, then run some $\{p\}$-fair schedule, starting in c_0, in which p takes no steps, until some decision is reached in a configuration $x = \mathsf{post}(c_0, s)$ for some schedule $s \in \mathsf{Act}^*$ containing no actions by p. Now:
   – The schedule s and the action a are independent, thus $y_0 := \mathsf{post}(c_0, s{\cdot}a) = \mathsf{post}(c_0, a \cdot s)$. Therefore, y_0 is reachable from both $x = \mathsf{post}(c_0, s)$ and $d_0 = \mathsf{post}(c_0, a)$. Because x and d_0 are both univalent, this implies that they have the same valence.
   – Also, the schedule s and the schedule $a' \cdot a$ are independent, thus $y_1 := \mathsf{post}(c_0, s \cdot a' \cdot a) = \mathsf{post}(c_0, a' \cdot a \cdot s)$. Therefore, y_1 is reachable from both $x = \mathsf{post}(c_0, s)$ and $d_1 = \mathsf{post}(c_0, a' {\cdot} a)$, which are both univalent, implying that x and d_1 have the same valence.
   – The previous two points together imply that d_0 and d_1 have the same valence which is a contradiction. □

Using the two lemmas, we will now construct an infinite, fair execution consisting entirely of bivalent configurations, which contradicts the correctness of the protocol.

– We start with some bivalent initial configuration, whose existence is guaranteed by Lemma 2.
– We pick the most senior enabled action a (as defined in Definition 10).
– We execute the action sequence $w \in \mathsf{Act}^*$ (whose existence is guaranteed by Lemma 3), then the action a, and end up in another bivalent configuration.
– Continue with step 2.5.

This construction yields an infinite execution; it is fair because we pick the most senior enabled action in step 2.5 and then execute it after a few more other steps w, which means that there is no neglect (as explained in the proof of Lemma 1). □

Finally, we can lift the restriction and allow general protocols as defined in Definition 5.

Corollary 1. *Let Φ be a consensus protocol and let* Pid *contain at least two processes. Then, Φ is not correct in the presence of silent crash failures: If $|F| > 1$, then L_Φ contains a complete F-fair execution that violates either validity, agreement, stability, or termination.*

Proof (Sketch only). The idea is to construct a simple consensus protocol $\overline{P}$ that simulates P, and whose F-fair executions correspond to F-fair executions of P. Thus, P can not be correct, otherwise we could use it to build a correct simple consensus protocol which we know does not exist.

The messages are the same ($\overline{\mathsf{Msg}} = \mathsf{Msg}$). The local state $\overline{\mathsf{Pst}}$ stores (1) the process state Pst, (2) an "inbox", i.e. a multiset representing messages that are available, and (3) a step counter recording how many times this process has taken a step, and (4) a data structure recording the timestamps (i.e. step counts) for messages in Msg and spontaneous actions in Act, used to calculate the seniority of actions as defined in Definition 10. On $\overline{\mathsf{receive}(p, m)}$, the received message is simply added to the inbox. On $\overline{\mathsf{run}(p)}$, we look for the most senior action, and execute it.

The key requirement is that for every fair execution $\overline{E}$ of $\overline{P}$ we find a corresponding fair execution E of P. Consider a message m: if it does not get neglected in $\overline{E}$, it must be received, meaning that it reaches the inbox; and because $\overline{\mathsf{run}(\mathsf{dst}(m))}$ does not get neglected in $\overline{E}$, it executes infinitely many times. Because the scheduler that is simulated by run is fair, as shown by Lemma 1, the simulated execution is fair as well. □

Ways Around Impossibility. Impossibility results are often called negative results, but in fact, they usually help us to discover new ways in which to change our approach or our definitions, in order to succeed. There are many ways to work around the impossibility result we just proved:

- The result applies only to asynchronous systems. We can solve consensus in synchronous systems, e.g. if we have some bounds on message delays.
- The result assumes that crashes are silent. We can solve consensus if we have failure detectors (for an extensive list of various consensus algorithms, see [5]).
- The result assumes an adversarial scheduler: this means that our proof constructs an extremely contrived schedule to prove nontermination.

The last item is perhaps the most interesting. In the next section, we show an asynchronous protocol for consensus that can be tuned to terminate quite efficiently in practice.

2.6 The PAXOS Protocol

We now have a closer look at the PAXOS protocol for asynchronous consensus by Leslie Lamport [11]. It is a standard mechanism to provide fault tolerance in distributed systems, and variations of the classic protocol are used in many practical systems, e.g. in the Chubby lock service [4] or in Zookeeper [9].

The basic idea is to perform a leader-based consensus: a leader p performs a voting round (whose goal is to reach consensus on a bit) by sending a proposal for a consensus value to all participants, and if p gets a majority to agree with the proposal, p informs all participant about the winning value. Voting rounds can fail for various reasons, but a leader can always start a new round, which can still succeed (i.e. the protocol never gets stuck with no chance of success).

The trick is to (1) design the protocol to satisfy agreement, validity, and stability even if there are many competing leaders, and (2) make it unlikely

types

$\mathsf{Round} = (\mathbb{N}_0 \times \mathsf{Pid})$	using lexicographic order
$\mathsf{Vote} = (\mathsf{Round} \times \{0,1\})$	using lexicographic order

process state

`state` :	$\{N, Q, P\}$	initially N	(leader)
`inbox` :	$\mathcal{P}(\mathsf{Msg})$	initially $\emptyset$	(leader)
`lasttried` :	$\mathbb{N}_0$	initially 0	(leader)
`quorum` :	$\mathcal{P}(\mathsf{Pid_a})$	initially $\emptyset$	(leader)
`lastpromise` :	Round	initially $(0, \mathsf{pid})$	(acceptor)
`lastvote` :	Vote	initially $((0, \mathsf{pid}), b_{\mathsf{pid}})$ for $b_{\mathsf{pid}} \in \{0,1\}$	(acceptor)
`decision` :	$\{\bot, 0, 1\}$	initially $\bot$	(learner)

messages

$\mathsf{Inquiry}(n : \mathbb{N}, p : \mathsf{Pid_l}, q : \mathsf{Pid_a})$	//sent from leader p to acceptor q
$\mathsf{LastVote}(n : \mathbb{N}, p : \mathsf{Pid_l}, q : \mathsf{Pid_a}, v : \mathsf{Vote})$	//sent from acceptor q to leader p
$\mathsf{Proposal}(n : \mathbb{N}, p : \mathsf{Pid_l}, q : \mathsf{Pid_a}, b : \{0,1\})$	//sent from leader p to acceptor q
$\mathsf{Vote}(n : \mathbb{N}, p : \mathsf{Pid_l}, q : \mathsf{Pid_a}, b : \{0,1\})$	//sent from acceptor q to leader p
$\mathsf{Winner}(p : \mathsf{Pid_l}, q : \mathsf{Pid_r}, b : \{0,1\})$	//sent from leader p to learner q

Fig. 2. Types, states and messages for the basic PAXOS consensus protocol.

(using ad-hoc heuristics) that there are many competing leaders at a time, thus termination is likely in practice.

There are three roles of participants (leaders, acceptors, learners) which we represent by three different process subsets $\mathsf{Pid_l}, \mathsf{Pid_a}, \mathsf{Pid_r}$ of Pid. Leaders conduct the organizational part of a voting round (solicit, collect, and analyze votes); acceptors perform the actual voting; and learners are informed about the successful outcome, if any. It is perfectly acceptable (and common in practice) for a process to play multiple roles. If everybody plays every role we have $\mathsf{Pid_l} = \mathsf{Pid_a} = \mathsf{Pid_r} = \mathsf{Pid}$. The number of acceptors must be finite ($|\mathsf{Pid_a}| < \infty$) so that they can form majorities.

Some key ideas include:

- Voting rounds are identified by a unique *round* identifier. This identifier is a tuple (n, p) consisting of a sequence number n and the process identifier p of the leader for this round. There is just one leader for each round, but different rounds can be initiated by different leaders, possibly concurrently.
- Each round has two and a half phases. In the first phase, the leader sends an inquiry message to all acceptors. The acceptors respond with a special message containing the last vote they cast (in a previous round), or a pseudo-vote containing their initial preference (if they have not cast any votes in a real round yet).
- When the leader has received a last-vote message from a quorum (i.e. at least half) of acceptors, it starts the second phase. In this phase, it proposes a consensus value and asks the quorum to vote for it.
- If the leader receives votes from all members of the quorum, it informs all learners about the successful outcome.

action answer$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l}, q : \mathsf{Pid}_\mathrm{a}, v : \mathsf{Vote})$ **at** q (acceptor)
| **receives** $\mathsf{Inquiry}(n, p, q)$
| **condition** $(\mathtt{lastpromise} < (n, p)) \wedge (\mathtt{lastvote} = v)$
| **sends** $\mathsf{LastVote}(n, p, q, v)$
| **updates** $\mathtt{lastpromise} \leftarrow (n, p)$

action accept$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l}, q : \mathsf{Pid}_\mathrm{a}, b : \{0, 1\})$ **at** q (acceptor)
| **receives** $\mathsf{Proposal}(n, p, b)$
| **condition** $\mathtt{lastpromise} = (n, p)$
| **sends** $\mathsf{Vote}(n, p, q, b)$
| **updates** $\mathtt{lastvote} \leftarrow ((n, p), b)$

action learn$(q : \mathsf{Pid}_\mathrm{r}, b : \{0, 1\})$ **at** q (learner)
| **receives** $\mathsf{Winner}(p, q, b)$
| **updates** $\mathtt{decision} \leftarrow b$

Fig. 3. The acceptor actions and the one learner actions for the basic PAXOS consensus protocol.

action inquire$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l})$ **at** p (leader)
| **condition** $(\mathtt{state} = N) \wedge (\mathtt{n} = \mathtt{lasttried} + 1)$
| **sends** $\sum_{q \in \mathsf{Pid}_\mathrm{a}} \mathsf{Inquiry}(n, p, q)$
| **updates** $\mathtt{state} \leftarrow Q; \mathtt{lasttried} \leftarrow n$

action propose$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l}, b : \{0, 1\}, Q : \mathcal{P}(\mathsf{Pid}_\mathrm{a}), lv : Q \to \mathsf{Vote})$ **at** p (leader)
| **condition** $\mathtt{inbox} \geq \sum_{q \in Q} \mathsf{LastVote}(n, p, q, lv(q))$
| **condition** $(\mathtt{state} = Q) \wedge (\mathtt{lasttried} = n) \wedge (|Q| > |\mathsf{Pid}_\mathrm{a}|/2)$
| **condition** $\max\{lv(q) \mid q \in Q\} = (\_, b)$
| **sends** $\sum_{q \in Q} \mathsf{Proposal}(n, p, q, b)$
| **updates** $\mathtt{state} \leftarrow P; \mathtt{quorum} \leftarrow Q; \mathtt{inbox} \leftarrow \emptyset$

action announce$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l}, b : \{0, 1\}, Q : \mathcal{P}(\mathsf{Pid}_\mathrm{a}))$ **at** p (leader)
| **condition** $\mathtt{inbox} \geq \sum_{q \in Q} \mathsf{Vote}(n, p, q, b)$
| **condition** $(\mathtt{state} = P) \wedge (\mathtt{lasttried} = n) \wedge (\mathtt{quorum} = Q)$
| **sends** $\sum_{q \in \mathsf{Pid}_\mathrm{r}} \mathsf{Winner}(p, q, b)$
| **updates** $\mathtt{state} \leftarrow N; \mathtt{inbox} \leftarrow \emptyset$

action receive$(m : \mathsf{Msg})$ **at** $\mathsf{dst}(m)$ (leader)
| **receives** m
| **updates** $\mathtt{inbox} \leftarrow \mathtt{inbox} + m$

action abandon$(n : \mathbb{N}, p : \mathsf{Pid}_\mathrm{l})$ **at** p (leader)
| **condition** $(\mathtt{lasttried} = n) \wedge (\mathtt{state} \in \{P, V\})$
| **updates** $\mathtt{state} \leftarrow N; \mathtt{inbox} \leftarrow \emptyset$

Fig. 4. The leader actions for the basic PAXOS consensus protocol.

We show the definitions of local states (for each role) and of message formats in Fig. 2. The actions are shown in Figs. 3 and 4.

The following properties of the protocol are key to ensure consensus even under concurrent voting rounds:

- Rounds are totally ordered (lexicographically based on the order, then the process id). Participants are no longer allowed to participate in a lower round once they are participating in a higher round.
- When transitioning from the first phase (gather last vote messages) to the second phase (send out proposal messages), the leader chooses the consensus value belonging to the highest vote among all the last-vote messages. This ensures that if a prior round was actually successful (i.e. it garnered a majority of votes), the new round uses the same bit value.

The following lemma formalized these intuititons, and constitutes the core of the correctness proof.

Lemma 4 (Competing Leaders). *If E is an execution and*

$$\mathsf{announce}(n, p, b, Q) \in \mathsf{trc}(E) \quad \textit{and} \quad \mathsf{propose}(n', p', b', Q', lv) \in \mathsf{trc}(E),$$

and $(n, p) < (n', p')$, then $b = b'$.

Proof. By contradiction. Assume the lemma is not true, then there exist E, p, n, b, Q, p', n', b', Q', lv falsifying the condition, and without loss of generality we can assume (n', p') are chosen minimal among all such. To perform $\mathsf{propose}(n', p', b', Q', lv)$, the leader p' received several LastVote messages; Let $((n'', p''), b') = \max_{q \in Q} lv(q)$ be the maximal vote received. Distinguish cases:

- $(n'', p'') < (n, p)$ this is impossible: because Q and Q' must intersect, there exists $q \in Q \cap Q'$. Since q must have voted for the round (n, p) before answering in the round (n', p') (otherwise it would not have voted), the LastVote message sent from q to p' must contain a vote whose round is no lower than (n, p) (note that the `lastvote` variable is always monotonically increasing).
- $(n'', p'') = (n, p)$ in that case, $b' = b$ because all votes for the same round have the same bit value. Contradiction.
- $(n'', p'') > (n, p)$. Because p is at least 1, so is p'', thus $((n'', p''), b')$ is a vote for a non-zero round, so there must exist some $\mathsf{propose}(n'', p'', b', \_, \_)$ in the execution. Because we chose (n', p') minimal among all such violating the lemma, this implies that $b = b'$. Contradiction.

The following lemma shows that no matter how many crashes occur, how many messages are lost, or how many leaders are competing, safety is always guaranteed.

Theorem 2. *All executions of PAXOS satisfy agreement, validity, and stability.*

Proof. Validity is easy because all votes can be tracked back to some initial vote, which is the preference of some processor. Stability and agreement follow because if we had two $\mathsf{announce}(n, p, b, Q)$ and $\mathsf{announce}(n', p', b', Q')$ with $b \neq b'$, and suppose that $(n, p) < (n', p')$ without loss of generality, then there must also be a $\mathsf{propose}(n', p', b', Q', lv')$, which contradicts Lemma 4.

Of course, termination is not possible for arbitrary fair schedules in the presence of failures because of Theorem 1. However, the following property holds: success always remains possible as long as there remains some non-crashed leader, some non-crashed learner, and at least $\lceil |\mathsf{Pid}_a/2| \rceil$ non-crashed acceptors. The reason is that:

- A leader cannot get stuck in any state: if it is waiting for something (such as the receipt of some message), and that something is not happening (for example, due to a crash), the leader can perform the spontaneous action abandon to return to a neutral state, from which it can start a new, higher round.
- If a leader p starts a new round (n, p) that is larger than any previous rounds, and if no other leaders are starting even higher rounds, and if at least $\lceil |\mathsf{Pid}_a/2| \rceil$ acceptors remain, and if there are no more crashes, then the round succeeds.

The PAXOS algorithm shown, and the correctness proof, are both based on the original paper by Lamport [11]. Since then, there have been many more papers on the subject, and many alternative (e.g. disk-based) and optimized (e.g. for solving continuous consecutive consensus problems) versions of PAXOS exist.

3 Strong Consistency and CAP

In this section we examine how to understand the consistency of shared data. We explore the cost of strong consistency (in terms of reliability or performance). We develop abstractions that help system implementors to articulate the consistency guarantees they are providing to programmers.

3.1 Objects and Operations

We assume that the shared data is organized as a collection of named ***objects*** Obj. As in the last section, we assume a set of ***processes*** Pid. The sets of objects and processes may be infinite, to model their dynamic creation. Processes interact with the shared data by performing ***operations*** on objects. Each object $x \in \mathsf{Obj}$ has a ***type*** $\tau = \mathsf{type}(x) \in \mathsf{Type}$, whose ***type signature*** $(\mathsf{Op}_\tau, \mathsf{Val}_\tau)$ determines the set of supported operations Op_τ and the set of their return values Val_τ. We assume that a special value $\perp \in \mathsf{Val}_\tau$ belongs to all sets Val_τ and is used for operations that return no value.

Example 1. An **integer register** `intreg` can be defined as follows: $\mathsf{Val}_{\texttt{intreg}} = \mathbb{Z} \cup \{\perp\}$, and $\mathsf{Op}_{\texttt{intreg}} = \{\texttt{rd}\} \cup \{\texttt{wr}(a) \mid a \in \mathbb{Z}\}$

Example 2. A **counter object** `ctr` can be defined as follows: $\mathsf{Val}_{\texttt{ctr}} = \mathbb{Z} \cup \{\perp\}$, and $\mathsf{Op}_{\texttt{ctr}} = \{\texttt{rd}, \texttt{inc}\}$.

Sequential Semantics. The type of an object, as defined above, does not actually describe the semantics of the operation, only their syntax. We formally specify the sequential semantics of a data type τ by a function

$$\mathcal{S}_\tau : \mathsf{Op}_\tau \times \mathsf{Op}_\tau^* \to \mathsf{Val}_\tau,$$

which, given an operation and sequence of prior operations, specifies the expected return value. For a register, read operations return the value of the last preceding write, or zero if there is no prior write. For a counter, read operations return the number of preceding increments. Thus, for any sequence of operations ξ:

$$\begin{aligned} \mathcal{S}_{\texttt{intreg}}(\texttt{rd}, \xi) &= a, \quad \text{if } \texttt{wr}(0)\,\xi = \xi_1\,\texttt{wr}(a)\,\xi_2 \text{ and} \\ &\qquad \xi_2 \text{does not contain } \texttt{wr} \text{ operations;} \\ \mathcal{S}_{\texttt{ctr}}(\texttt{rd}, \xi) &= (\text{the number of } \texttt{inc} \text{ operations in } \xi); \end{aligned}$$

Our definition of the sequential semantics uses sequences of prior operations (representing all earlier updates), rather than the current state of an object, to define the behavior of reads. This choice is useful: for many implementations, there are multiple versions of the state, and these versions are often best understood as the result of using various update sequences (such as logs), subsequences, or segments.

Moreover, for objects such as the integer register, only the last update matters, since it overwrites completely all information in the object. For the counter, however, all updates matter. Similarly, if considering objects that have multiple fields and support partial updates, e.g. updates that modify individual fields, it is not enough to look at the last update to determine the current state of the object.

In general, operations may both read and modify the state. Operations that return no value are called *update-only* operations. Similarly, we call an operation o of a type τ *read-only* if it has no side effect, i.e. if for all $o' \in \mathsf{Op}_\tau$ and $u, v \in \mathsf{Op}_\tau^*$, we have $\mathcal{S}_\tau(o', u \cdot o \cdot v) = \mathcal{S}_\tau(o', u \cdot v)$.

What is an Object? There is often some ambiguity to the question of what we should consider to be an object. For example, consider a cloud table storage API that provides tables that store records (consisting of several fields that have values) indexed by keys. Then:

- We can consider each record to be an object, named by the combination of the table name and the key, and supporting operations for reading and writing fields or removing the object.
- We can consider the whole table to be an object, named by the table name. Operations specify the key (and the field, if accessing individual fields).
- We can consider each field to be an object, named by the combination of the table name, the key, and the field name. This approach seems most consistent with the types shown above (integer registers, counters).
- We can consider the entire storage to be a single object, and have operations to target a specific (table, key, field) combination.

We propose the following definition, or perhaps we should say guideline:

- An object is the largest unit of data that can be written atomically without using transactions.
- A transactional domain is the largest unit of data that can be written atomically by using transactions.

Traditional databases follow a philosophy without objects (nothing can be written outside of a transaction) and large transactional domains (the entire database), which requires strong transaction support. Cloud storage and web programming rely more commonly on moderately to large sized objects, and transactional domains that do not contain all data (transaction support is typically nonexistent, or at best limited). The reason is that the latter approach is easier to guarantee as a scalable service. Unfortunately, it is also harder to program.

3.2 Strong Consistency

Intuitively, programmers expect operations on shared data to be *linearizable*. Informally, this means that when they call into some API to read or write a shared value, they expect a behavior that is consistent with (i.e. observationally undistinguishable from):

- a single copy of the shared data being maintained somewhere.
- the read or write operations being applied to that copy somewhere in between the call and the return.

Unfortunately, guaranteeing these conditions can be a performance and reliability problem, if communication between processes is expensive and/or unavailable. Many systems thus relax the consistency. A good test to see whether a system is indeed linearizable (in fact, sequentially consistent) is shown in Fig. 5. On an linearizable or sequentially consistent system, when running programs A and B (one time each), there is at most one winner. Why? Informally, it is because under sequential consistency, *all* operations are organized into some global sequence. In this case, it means that the two writes must happen in *some* order — we don't know which one, but the system will decide on one or the other, which implies that either A or B (or both) do not win:

- If the system decides that A's write to x happens before B's write to y, then it must also happen before B's read from x, thus the value read must be 1, so B does not win.
- If the system decides that B's write to y happens before A's write to x, then it must also happen before A's read from y, thus the value read must be 1, so A does not win.

This reasoning seems still a bit informal - talking about 'happens before' without a solid foundation can get quite confusing. In order to give a more rigorous reasoning, we first need a precise definition of what sequential consistency and linearizability mean.

Program (A)	Program (B)
$\mid x.\texttt{wr}(1)$; $//a_1$ $\mid$ if $(y.\texttt{rd} = 0)$ $//a_2$ $\mid \mid$ print "A wins";	$\mid y.\texttt{wr}(1)$; $//b_1$ $\mid$ if $(x.\texttt{rd} = 0)$ $//b_2$ $\mid \mid$ print "B wins";

Fig. 5. The Dekker Litmus test, using two integer registers x, y (which are initially 0). If we run these two concurrently on a sequentially consistent or linearizable system, there is at most one winner.

Abstract Executions. To specify consistency models, we use *abstract executions*. The basic idea is very simple:

1. A consistency model is formalized as a set of abstract executions, which are mathematical structures (visualized using graphs) consisting of operation events (vertices) and relations (edges), subject to conditions. Abstract executions capture "the essence" of an execution (that is, what operations occurred, and how those operations are related), without including low-level details (such as exactly what messages were sent when and where).
2. We describe what it means for a concrete execution of a system to *correspond* to an abstract execution.
3. We say that a system is correct if *all* of its concrete executions correspond to some abstract execution of the consistency model.

The advantage of this approach is that we can separately (1) determine whether programs are correct for a given consistency model, without needing to know details about the system architecture, and (2) determine whether a system correctly implements some consistency model, without knowing anything about the program that is running on it. Consistency models can be thought of as a contract between the programmer and the system implementor.

For sequential consistency, we define abstract executions in two steps. First, we define operation graphs.

Definition 14. *An operation graph is a tuple* $(\mathsf{Evt}, \mathsf{pid}, \mathsf{obj}, \mathsf{op}, \mathsf{rval}, \mathsf{po})$ *where*

- Evt *is a set of events.*
- $\mathsf{pid} : \mathsf{Evt} \to \mathsf{Pid}$ *describes the process on which the event happened.*
- $\mathsf{po} \subseteq \mathsf{Evt} \times \mathsf{Evt}$ *is a partial order (called process order) that describes the order in which events happened on each process. We require that* po *is a union of total orders for each process, that is, there exist for each* $p \in \mathsf{Pid}$ *a total order* $\mathsf{po}_p \subseteq (\mathsf{pid}^{-1}(p) \times \mathsf{pid}^{-1}(p))$ *such that* po *is their union:* $\mathsf{po} = \bigcup_{p \in \mathsf{Pid}} \mathsf{po}_p$.
- $\mathsf{obj}, \mathsf{op}, \mathsf{rval}$ *are event attributes (i.e. functions* Evt*) describing the details of the operation: each event* $e \in \mathsf{Evt}$ *represents an operation* $\mathsf{op}(e) \in \mathsf{Op}_{\mathsf{type}(\mathsf{obj}(e))}$ *on an object* $\mathsf{obj}(e) \in \mathsf{Obj}$*, which returns the value* $\mathsf{rval}(e) \in \mathsf{Val}_{\mathsf{type}(\mathsf{obj}(e))}$.

Operation graphs capture the relevant interactions between the system and the client program. However, they do not explain the underlying reasons. Looking just at the operation graph, it can be difficult to determine the order in

which the system processed operations. Abstract executions contain this additional information: in the case of sequential consistency, a total order over all operations:

Definition 15. *Define the set $\mathcal{A}_{SC}$ of sequentially consistent abstract executions to consist of all tuples*
$(\mathsf{Evt}, \mathsf{pid}, \mathsf{obj}, \mathsf{op}, \mathsf{rval}, \mathsf{po}, \mathsf{to})$*, where*

- $(\mathsf{Evt}, \ldots, \mathsf{po})$ *is an operation graph.*
- $\mathsf{to} \subseteq \mathsf{Evt} \times \mathsf{Evt}$ *is a total order.*
- to *is consistent with process order:* $\mathsf{po} \subseteq \mathsf{to}$.
- *The return value of each operation matches the sequential specification $\mathcal{S}_\tau$ (as defined in Sect. 3.1), applied to the sequence of* to*-prior operations:*

$$\forall e \in \mathsf{Evt} : \mathsf{rval}(e) = \mathcal{S}_{\mathsf{type}(\mathsf{obj}(e))}(\mathsf{op}(e), (\mathsf{to}^{-1}(e) \cap \mathsf{obj}^{-1}(\mathsf{obj}(e))).\mathsf{sort}(\mathsf{to}))$$

In pictures, we usually draw abstract executions by (1) creating a vertex for each event, and aligning events into columns corresponding to process identifiers, and (2) adding arrows to represent to ordering edges.

We can now define sequential consistency; note that we purposefully omit a precise definition of what a concrete execution is, but simply assume that it contains operation events that can be meaningfully related to the abstract execution.

Definition 16. *A concrete execution of some system is sequentially consistent if there exists an abstract sequentially consistent execution, with corresponding operation events, process order, and attributes.*

Dekker Explanation. We can now explain why under sequential consistency, there can never be two winners in the Dekker litmus test (Fig. 5). Suppose there were two winners. This would mean that in the corresponding abstract execution, there are four events $\{a_1, a_2, b_1, b_2\}$ (meaning that $\mathsf{pid}(a_1) = \mathsf{pid}(a_2) = a$, $\mathsf{pid}(b_1) = \mathsf{pid}(b_2) = b$, $\mathsf{obj}(a_1) = \mathsf{obj}(b_2) = x$, $\mathsf{obj}(b_1) = \mathsf{obj}(a_2) = y$, $\mathsf{op}(a_1) = \mathsf{op}(b_1) = \mathtt{wr}(1)$, $\mathsf{op}(a_2) = \mathsf{op}(b_2) = \mathtt{rd}$, $\mathsf{rval}(a_2) = \mathsf{rval}(b_2) = 0$, and $\mathsf{po} = \{(a_1, a_2), (b_1, b_2)\}$).

Now we can argue that there is no way to construct to without creating a cycle and thus a contradiction:

- Because $\mathsf{rval}(a_2) = 0$, it cannot be the case that $b_1 \xrightarrow{\mathsf{to}} a_2$ (because that would imply a return value of 1). Therefore, because to is a total order, $a_2 \xrightarrow{\mathsf{to}} b_1$.
- Because $\mathsf{rval}(b_2) = 0$, it cannot be the case that $a_1 \xrightarrow{\mathsf{to}} b_2$ (because that would imply a return value of 1). Therefore, because to is a total order, $b_2 \xrightarrow{\mathsf{to}} a_1$.
- Because $\mathsf{po} \subseteq \mathsf{to}$, $a_1 \xrightarrow{\mathsf{to}} a_2$ and $b_1 \xrightarrow{\mathsf{to}} b_2$.

Linearizability. Sometimes, systems use a slightly stronger consistency model than sequential consistency, called *linearizability*. The difference is that for linearizability, we additionally require that the order to must not contradict the order of operation calls and operation returns in the concrete execution.

Definition 17. *A concrete execution of some system is* linearizable *if there exists a corresponding abstract sequentially consistent execution, such that for any two operations* $e, e' \in \mathsf{Evt}$ *in the abstract execution satisfying* $e \xrightarrow{\mathsf{to}} e'$*, it is not the case that* $\mathsf{return}(e') < \mathsf{call}(e)$ *in the concrete execution.*

Note that any linearizable concrete execution is also sequentially consistent. The converse is not true in general; we will show an example in the next section.

There is an alternative popular interpretation of linearizability that roughly goes as follows: The abstract execution must be consistent with a placement of *commit events* of operations, which are placed somewhere in between call and return. The two definitions are equivalent: (1) if the order matches commit events, then it cannot violate the condition above, and (2) if the condition above is not violated, we can find a commit event placement.

3.3 CAP Theorem

The CAP theorem explores tradeoffs between **C**onsistency, **A**vailability, and **P**artition tolerance, and concludes that, while it is possible to provide any two of these properties, it is impossible to provide all three. It was conjectured by Brewer [1] and proved by Gilbert and Lynch [8]. Our proof here follows the same simple reasoning as the one by Gilbert and Lynch, but we use sequential consistency instead of linearizability.

We use the following meaning of the three terms. *Consistency* means sequential consistency as defined above. *Availability* means that all operations on objects eventually complete. *Partition Tolerance* means that the system keeps operating even if the network becomes *permanently partitioned*, i.e. if there exists a subset of isolated processes $\mathsf{Iso} \subseteq \mathsf{Pid}$ such that the processes in Iso and the processes in $\mathsf{Pid} \setminus \mathsf{Iso}$ cannot communicate in any way.

Theorem 3 (CAP). *No system with at least two processes can provide sequential consistency, availability, and partition tolerance.*

Proof. Assume such a system exists. Consider two processes $a, b \in \mathsf{Pid}$ and a permanent network partition $\mathsf{Iso} = \{a\}$ that isolates process a. We run three independent experiments, called A, B, and AB. In experiment A, process a runs the program (A) shown in Fig. 5, while process b does nothing. In experiment B, process b runs the program (B) shown in Fig. 5, while process a does nothing. In experiment AB, both processes run the respective program. Then:

- In experiment A, availability and partition tolerance imply that the code executes to completion. Consistency means that process a prints "A wins" (because there is only one process accessing the data, the semantics is equivalent to standard sequential semantics).
- There is no way for process a to distinguish between experiments A and AB, thus it must print "A wins" in experiment AB as well.
- For the symmetric reason, process b must print "B wins" in experiment AB.

– Thus, in experiment AB, both "A wins" and "B wins" are printed, which is not sequentially consistent. Contradiction.

Although the theorem above is narrowly stated, the proof reveals a somewhat wider impact:

– The proof reveals the performance impact of strong consistency: it shows that the partitions *have to talk to each other* before completing the execution of the program. Thus, if communication is expensive (for example, if two data centers have to talk to each other across a far distance), clients are forced to wait.
– Simply knowing about the partition is not helpful. Even if the processes have perfect information about the existence of a network partition, the above reasoning holds. This is different from the situation with consensus in asynchronous systems with crash failures, where the impossibility of distinguishing between failure and slow response is key, and a perfect failure detector can make consensus possible.

C+A is Possible. Consistency and Availability can be easily guaranteed. A whole range of solutions are possible:

– (Single Copy). The simplest idea is to just pick one process to store the data, then forward all read and write operations to that process. In the absence of partitions, we can always reach this process from everywhere.
– (Primary Replication). In this case, we allow all processes to store a copy of the data, and to also read data locally. However, (1) all writes must be first performed on a designated replica, the *primary* replica, before applying them to a secondary replica, and (2) all writes must be applied to the secondary replicas in the same order that they were applied to the primary replica. Primary replication can greatly enhance the latency and the throughput of read operations, but write operations remain slow.

C+P is Possible. We can guarantee consistency and partition tolerance by simply stalling the execution of write requests if the primary copy cannot be reached.

A+P is Possible. It is trivial to guarantee availability and partition tolerance without consistency, for example, by giving each process its own isolated copy of the data. However, this is hardly meaningful.

C'+A+P' is Possible. The most useful approximation to CAP is to use a weaker form of consistency (eventual consistency) in conjunction with a weaker from of partition tolerance (resilience against temporary network partitions). Informally, it means that the shared data remains available for reading and writing even in the presence of network partitions. When the network partition

heals, processes reconcile conflicting updates that happened during the network partition, and converge to a common state. Understanding specifications and implementations of eventual consistency is the main topic for the remainder of this course.

4 Eventual Consistency Models and Mechanisms

Weakening the consistency guarantees can improve performance and availability, but it can also create problems for unaware programmers. Understanding exactly what can go wrong, and how to write programs that are resilient, remains an important challenge. One of the key difficulties is that there are many subtle variations of consistency models, and myriads of architectures and optimizations that all have slightly different effects. We study this problem by approaching it from two sides:

- In Sect. 4.1, we show how to generalize sequentially consistent abstract executions to eventually consistent abstract executions, and show how to express various guarantees (causality, consistent prefix, read my writes, monotonic reads) and combinations of guarantees.
- In Sect. 4.2, we take a closer look at a few selected architectures that implement some form of consistency, and show how to specify their behavior using abstract executions.

4.1 Eventual Consistency Models

The following simple definition of quiescent consistency is often used to describe eventually consistent systems:

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state.

However, quiescent consistency is very weak. For example, it (1) does not specify what happens if clients never stop issuing updates, which is common in reactive systems such as services, and (2) does not in any way restrict the intermediate values. Few programs will work correctly under quiescent consistency, and most architectures provide much stronger guarantees. Thus, we need a better way to define eventual consistency models.

To devise a better model for eventual consistency, we start by deconstructing our definition of sequential consistency (Definition 15). In that definition, we use a total order to to figure out what value an operation e on some object $x = \mathsf{obj}(e)$ should return:

$$\forall e \in \mathsf{Evt} : \mathsf{rval}(e) = \mathcal{S}_{\mathsf{type}(x)}(\mathsf{op}(e), (\mathsf{to}^{-1}(e) \cap \mathsf{obj}^{-1}(x)).\mathsf{sort}(\mathsf{to})) \quad (1)$$

The key observation is that the total order to is playing two independent roles:

1. It is used to determine what prior operations are *visible* to e. In (1), this is the part $\mathsf{to}^{-1}(e)$, which returns the set of all operations e' such that $e' \xrightarrow{\mathsf{to}} e$.
2. It is used to *arbitrate* between conflicting operations. In (1), this is the part $\mathsf{sort}(\mathsf{to})$: it ensures that everyone is using the same order to sort conflicting operations (e.g. multiple writes to the same location).

Definition 18. *Given a type τ, we say two operations $o_1, o_2 \in \mathsf{Op}_\tau$ are* write-conflicting *if there exists an operation $o \in \mathsf{Op}_\tau$ and operation sequences $u, w \in \mathsf{Op}_\tau^*$ such that $\mathcal{S}_\tau(o, u \cdot o_1 \cdot o_2 \cdot w) \neq \mathcal{S}_\tau(o, u \cdot o_2 \cdot o_1 \cdot w)$. Given an operation graph $(\mathsf{Evt}, \ldots, \mathsf{obj}, \mathsf{op}, \ldots)$, we say that two events $e_1, e_2 \in \mathsf{Evt}$ are* write-conflicting *(written as $\mathsf{wconflict}(e_1, e_2)$) if (1) $\mathsf{obj}(e_1) = \mathsf{obj}(e_2)$, and (2) $\mathsf{op}(e_1)$ and $\mathsf{op}(e_2)$ are write-conflicting.*

We now define eventually consistent abstract executions, similar to (Definition 15), but using two separate relations; a *visibility relation* is used to determine what operations are visible, and an *arbitration order* is used to determine how to order conflicting operations.

Definition 19. *Define the set $\mathcal{A}_{\mathrm{EC}}$ of* **eventually consistent abstract executions** *to consist of all tuples $(\mathsf{Evt}, \mathsf{pid}, \mathsf{obj}, \mathsf{op}, \mathsf{rval}, \mathsf{po}, \mathsf{vis}, \mathsf{ar})$, where*

1. *$(\mathsf{Evt}, \ldots, \mathsf{po})$ is an operation graph.*
2. *The visibility relation $\mathsf{vis} \subseteq \mathsf{Evt} \times \mathsf{Evt}$ is an acyclic, irreflexive relation.*
3. *Operations become eventually visible: for all $e \in \mathsf{Evt}$, $e \xrightarrow{\mathsf{vis}} e'$ for almost all $e' \in \mathsf{Evt}$ (i.e. all but finitely many).*
4. *The arbitration order $\mathsf{ar} \subseteq \mathsf{Evt} \times \mathsf{Evt}$ is a partial order.*
5. *The arbitration order orders all conflicting operations that are visible to another operation: for all $e_1, e_2, e \in \mathsf{Evt}$:*

$$((e_1 \xrightarrow{\mathsf{vis}} e) \wedge (e_2 \xrightarrow{\mathsf{vis}} e) \wedge \mathsf{wconflict}(e_1, e_2)) \Rightarrow ((e_1 \xrightarrow{\mathsf{ar}} e_2) \vee (e_2 \xrightarrow{\mathsf{ar}} e_1))$$

6. *There are no causal cycles: $\mathsf{po} \cup \mathsf{vis}$ is acyclic.*
7. *The return value of each operation matches the sequential specification $\mathcal{S}_\tau$ applied to visible operations in arbitration order:*

$$\forall e \in \mathsf{Evt} : \mathsf{rval}(e) = \mathcal{S}_{\mathsf{type}(\mathsf{obj}(e))}(\mathsf{op}(e), (\mathsf{vis}^{-1}(e) \cap \mathsf{obj}^{-1}(\mathsf{obj}(e))).\mathsf{sort}(\mathsf{ar}))$$

Note how the return value is determined in condition 7: first, it determines the set of visible events on the same object $\mathsf{vis}^{-1}(e) \cap \mathsf{obj}^{-1}(\mathsf{obj}(e))$, then it sorts this set into a sequence using ar, and then applies the sequential semantics. Although the sorting is not quite deterministic (since ar is not necessarily a total order), the value of the whole expression *is* deterministic because condition 5 ensures that ar determines at least the order of write-conflicting operations.

For an abstract eventually consistent execution A, we define the *happens-before* order hb_A, sometimes also called the causal order, to be the partial order $\mathsf{hb}_A = (A.\mathsf{po} \cup A.\mathsf{vis})^+$ (note that we rely on the acyclicity guaranteed by condition 6). The happens-before order tracks potential causal dependency chains:

if two operations are issued by the same process ($a \xrightarrow{\mathsf{po}} b$), or if the first operation is visible to the second ($a \xrightarrow{\mathsf{vis}} b$), the second may causally depend on the first.

How do these concepts map into practical implementations? Consider a typical implementation where each process maintains a replica of the shared state. Updates performed on a replica are broadcast to other replicas in some way. Visibility and arbitration are often determined in one of the following ways:

- Arbitration is typically determined either by (1) some timestamp, or (2) the order in which updates are processed on some primary replica.
- Visibility is typically determined by two factors, (1) the timing of when a process learns about an update (a process learns about a local update immediately, and about a remote update when it receives a message), and (2) the time at which a process chooses to make that update visible to subsequent queries (which could be as soon as it learns about it, or delayed, for example until an update is confirmed by the primary replica).

Eventual consistency is much stronger than quiescent consistency, but still quite weak. Most of the time, systems guarantee additional properties. In particular, the following guarantees are common. We start with a table giving the formal definition, and explain them below. These guarantees are not mutually exclusive; quite to the contrary, most systems provide a combination.

Guarantee	Condition
Sequential consistency	$\mathsf{vis} = \mathsf{ar}$
Read my writes	$\mathsf{po} \subseteq \mathsf{vis}$
Consistent prefix	ar is total, and $\forall e : \exists e' : \mathsf{vis}^{-1}(e) = \mathsf{ar}^{-1}(e')$
Monotonic reads	$(\mathsf{vis}; \mathsf{po}) \subseteq \mathsf{vis}$
Causal visibility	$\mathsf{hb} \subseteq \mathsf{vis}$
Causal arbitration	$\mathsf{hb} \subseteq \mathsf{ar}$

Sequential Consistency. We already defined this in the last section. Formally, sequential consistency means that arbitration and visibility are one and the same.

Read My Writes. If the same process performs two operations, it may expect that the first operation is visible to the second. For example, if we increment and then read a counter on the same process, read-my-writes guarantees that the read does not return zero.

Consistent Prefix. Sometimes it is acceptable to read a stale value, as long as that value appears as a past value of some timeline of values that everyone agrees on. Consistent prefix means just that: (1) a timeline is maintained (ar is a total order), and (2) the visible updates for any event e match some prefix of ar.

Monotonic Reads. One may except that once an update has become visible to an operation on some process, it should remain visible to all future operations on the same process.

Causal Visibility. If an operation has a causal chain to another operation, we may expect the second operation to see the first. Causal Visibility implies monotonic-reads and read-my-writes.

Causal Arbitration. If an operation has a causal chain to another operation, we may expect that the second one is ordered after the first in arbitration order.

We now illustrate these guarantees on a couple of examples.

Score Example. First, let us look at a sports example (following Doug Terry's baseball example [13]). Consider a match in which a home team and a visitors team score points, and the respective scores are stored in integer registers $\{\mathtt{h}, \mathtt{v}\} \subseteq \mathsf{Obj}$. Furthermore, assume that we are using a system where ar is a total order based on timestamps that reflect the real time at which operations are performed. Now, consider an abstract execution in which there are seven write events and two read events, ordered by ar as follows (note that we are not assuming that they are all issued by the same process):

$$\begin{array}{l}
\mathtt{h.wr}(1)\\
\mathtt{v.wr}(1)\\
\mathtt{h.wr}(2)\\
\mathtt{h.wr}(3)\\
\mathtt{v.wr}(2)\\
\mathtt{h.wr}(4)\\
\mathtt{h.wr}(5)\\
\text{print } (\mathtt{v.rd}() + \text{“-”} + \mathtt{h.rd}())
\end{array}$$

How do the various guarantees impact what possible scores could be printed at the end? Here is a table listing all the possibilities:

Sequential Consistency	2-5
Eventual Consistency	0-0, 0-1, 0-2, 0-3, 0-4, 0-5,
	1-0, 1-1, 1-2, 1-3, 1-4, 1-5,
	2-0, 2-1, 2-2, 2-3, 2-4, 2-5
Consistent Prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5

What if a process prints the score twice? By default, each read can print any of the options above. However, if the system guarantees monotonic reads or causal visibility, the second read can only report scores that are higher than they were in the first read.

Causality Example. Not all systems guarantee causal arbitration or causal visibility. This can lead to odd behaviors. For example, consider a chat application where participants $\{\text{Alice}, \text{Bob}, \text{Carol}\} \subseteq \mathsf{Pid}$ append to a list (the "wall"), or read the list. Alice asks a question, and Bob sees it and answers it. Finally, Carol looks at the chat and sees Bob's answer. But what about Alice's question?

Alice	Bob	Carol
e_1: wall.append("Anyone?")	e_2: print wall.rd	e_4: print wall.rd
	e_3: wall.append("Bob here.")	

Since Bob saw Alice's question, we know $e_1 \xrightarrow{\mathsf{vis}} e_2$, and since Carol saw Bob's answer, we know $e_3 \xrightarrow{\mathsf{vis}} e_4$. However:

- If the system does not guarantee causal visibility, then it is possible that $e_1 \not\xrightarrow{\mathsf{vis}} e_4$. Thus, Carol does not see Alice's question, even though she saw Bob's answer. However, if the system *does* guarantee causal visibility, then $e_1 \xrightarrow{\mathsf{vis}} e_2 \xrightarrow{\mathsf{po}} e_3 \xrightarrow{\mathsf{vis}} e_4$ implies $e_1 \xrightarrow{\mathsf{hb}} e_4$ which implies $e_1 \xrightarrow{\mathsf{vis}} e_4$.
- If the system does not guarantee causal arbitration, then it is possible that Carol sees both appends ($\{e_1, e_3\} \subseteq \mathsf{vis}^{-1}(e_4)$), but that they appear in the wrong order ($e_3 \xrightarrow{\mathsf{ar}} e_1$). However, if the system *does* guarantee causal arbitration, then $e_1 \xrightarrow{\mathsf{vis}} e_2 \xrightarrow{\mathsf{po}} e_3 \xrightarrow{\mathsf{vis}} e_4$ implies $e_1 \xrightarrow{\mathsf{hb}} e_4$ which implies $e_1 \xrightarrow{\mathsf{ar}} e_4$.

Causal visibility is easily violated in systems that do not use primary replication, but broadcast updates directly. However, even in such systems, causal visibility guarantees are possible and sensible, as shown in the COPS paper and algorithm, titled "Don't settle for eventual" [12].

Causal arbitration can easily be violated if arbitration is based on physical timestamps (i.e. timestamps provided by physical clocks on the various devices), and if those clocks exhibit skew. Often, systems use logical clocks (such as Lamport clocks) which are by construction consistent with the happens-before relation, thus avoiding this problem.

4.2 Eventual Consistency Mechanisms

We now discuss four protocols $\Phi_a, \Phi_b, \Phi_c, \Phi_t$ that provide various levels of consistency, as shown in the table below:

Eventually Consistent Protocol Φ_t. First, we look at the protocol with the weakest guarantees, which is quite simple (Fig. 6). Each process keeps a set `known` of known updates. When performing an update, this update is added to the local set, and also broadcast to all other processes; when they receive the update, they add it to their set. All updates are timestamped, using Lamport's scheme based on logical clocks [10]. When computing the return value of an operation,

	Primary replication			Direct broadcast
	Φ_a	Φ_b	Φ_c	Φ_t
Sequential consistency	✓	—	—	—
Read my writes	✓	—	✓	✓
Consistent prefix	✓	✓	—	—
Monotonic reads	✓	✓	✓	✓
Causal visibility	✓	✓	✓	—
Causal arbitration	✓	✓	✓	✓
Available under partitions	—	✓	✓	✓

types
| $\mathsf{Update} = \mathbb{N}_0 \times \mathsf{Pid} \times \mathsf{Obj} \times \mathsf{Op}$ ordered lexicographically
process state
| $\texttt{known}$: $\mathcal{P}(\mathsf{Update})$ // initially $\emptyset$
| $\texttt{clock}$: $\mathbb{N}_0$ // initially 0
messages
| $\mathsf{Inform}(u : \mathsf{Update}, q : \mathsf{Pid})$ //sent from $u.\mathsf{second}$ to q
action $\mathsf{perform}(p : \mathsf{Pid}, n : \mathbb{N}_0, x : \mathsf{Obj}, o : \mathsf{Op}, r : \mathsf{Val})$ **at** p
| **condition** $(n = \texttt{clock}) \wedge (r = \mathcal{S}_{\mathsf{type}(x)}(o, \mathsf{ops}_x(\texttt{known}).\mathsf{sort}))$
sends $\bigcup_{q \in \mathsf{Pid}} \mathsf{Inform}((n, p, x, o), q)$
updates $\texttt{known} \leftarrow \texttt{known} \cup \{(n, p, x, o)\}; \texttt{clock} \leftarrow \texttt{clock} + 1;$
action $\mathsf{learn}(u : Update, q : \mathsf{Pid})$ **at** q
| **receives** $\mathsf{Inform}(u, q)$
updates $\texttt{known} \leftarrow \texttt{known} \cup \{u\}; \texttt{clock} \leftarrow \max\{\texttt{clock}, (u.\mathsf{first} + 1)\};$

Fig. 6. Eventually consistent protocol Φ_t based on direct broadcast and Lamport timestamps.

the updates are sorted according to timestamps, and filtered according to the object they target (we define the function ops_x to filter updates from a sequence that target object x), then fed into the function $\mathcal{S}$ which tells us what value to return.

It is easy to show that this protocol is eventually consistent; to construct a corresponding abstract execution, we simply use one event per perform action. For the arbitration order, we use the lexicographic order over timestamps. For the visibility order, we say that e is visible to e' if the update tuple for e is in the known set when e' is performed.

Without further optimizations, this protocol is not practical since it consumes too much space. However, it is easy to see that for most data types, we can reduce the known set. For example, when working with registers, it is enough to keep only the latest update for each object, without altering the semantics. This is known as Thomas' rule [14].

types
| $\mathsf{Update} = \mathsf{Pid}_{\mathrm{sec}} \times \mathsf{Obj} \times \mathsf{Op} \times \mathbb{N}_0$

process state
| $\mathtt{busy}$: $(\bot \cup \mathsf{Update})$ initially $\bot$ (secondary)
| $\mathtt{localcount}$: $\mathbb{N}_0$ initially 0 (secondary)
| $\mathtt{confirmed}$: Update^* initially ϵ (secondary)

messages
| $\mathsf{Update}(u : \mathsf{Update})$ //sent from secondary $u.\mathsf{first}$ to primary p
| $\mathsf{Inform}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ //sent from primary p to secondary q'

action $\mathsf{read}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, r : \mathsf{Val})$ **at** q (secondary)
| **condition** $(\mathtt{busy} = \bot) \wedge (o$ is a read-only operation$)$
| **condition** $r = \mathcal{S}_{\mathsf{type}(x)}(o, \mathsf{ops}_x(\mathtt{confirmed}))$

action $\mathsf{update}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, l : \mathbb{N}_0)$ **at** q (secondary)
| **condition** $(\mathtt{busy} = \bot) \wedge (o$ is a update-only operation$) \wedge (l = \mathtt{localcount})$
| **sends** $\mathsf{Update}(q, x, o, l)$
| **updates** $\mathtt{busy} \leftarrow (q, x, o, l); \mathtt{localcount} \leftarrow \mathtt{localcount} + 1;$

action $\mathsf{perform}(u : \mathsf{Update})$ **at** p (primary)
| **receives** $\mathsf{Update}(u)$
| **sends** $\bigcup_{q' \in \mathsf{Pid}} \mathsf{Inform}(u, q')$

action $\mathsf{learn}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ **at** q' (secondary)
| **receives in-order** $\mathsf{Inform}(u)$
| **updates** $\mathtt{confirmed} \leftarrow \mathtt{confirmed} \cdot u$
| **updates** if $\mathtt{busy} = u$ then $\mathtt{busy} \leftarrow \bot$

Fig. 7. Sequentially consistent protocol Φ_a based on primary replication, supporting local reads on secondaries, for some primary process $p \in \mathsf{Pid}$ and secondary processes $\mathsf{Pid}_{\mathrm{sec}} \subseteq \mathsf{Pid}$.

Sequentially Consistent Protocol Φ_a. Figure 7 shows a protocol based on primary replication. Operations are performed at the secondary replicas, with identifiers $\mathsf{Pid}_{\mathrm{sec}} \subseteq \mathsf{Pid}$. Each secondary replica stores a sequence `confirmed` of updates it received from the primary replica, using in-order delivery. Read-only operations are performed locally on secondary replicas, by consulting the updates stored in `confirmed`. Other operations issued on secondary replicas are broken down into beginoperation and endoperation. beginoperation sends the update to the primary. Nothing else can happen on the secondary, until this same update is confirmed by the primary.

Executions are sequentially consistent. To obtain an abstract execution, define the events to be the actions read and perform, and define $\rightarrow$ to be the total order that we obtain by (1) taking the total order in which the perform events appear in the execution, and (2) inserting read into this chain anywhere after the last update confirmed before the local read, and before the next update confirmed after the local read.

types
| $\mathsf{Update} = \mathsf{Pid}_{\mathrm{sec}} \times \mathsf{Obj} \times \mathsf{Op} \times \mathbb{N}_0$

process state
| `localcount` : $\mathbb{N}_0$ initially 0 (secondary)
| `confirmed` : Update^* initially ϵ (secondary)

messages
| $\mathsf{Update}(u : \mathsf{Update})$ //sent from secondary $u.\mathsf{first}$ to primary p
| $\mathsf{Inform}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ //sent from primary p to secondary q'

action $\mathsf{read}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, r : \mathsf{Val})$ **at** q (secondary)
| **condition** (o is a read-only operation)
| **condition** $r = \mathcal{S}_{\mathsf{type}(x)}(o, \mathsf{ops}_x(\texttt{confirmed}))$

action $\mathsf{update}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, l : \mathbb{N}_0)$ **at** q (secondary)
| **condition** (o is a update-only operation) $\wedge$ ($l = \texttt{localcount}$)
| **sends** $\mathsf{Update}(q, x, o, l)$
| **updates** $\texttt{localcount} \leftarrow \texttt{localcount} + 1$

action $\mathsf{perform}(u : \mathsf{Update})$ **at** p (primary)
| **receives in-order** $\mathsf{Update}(u)$
| **sends** $\bigcup_{q' \in \mathsf{Pid}} \mathsf{Inform}(u, q')$

action $\mathsf{learn}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ **at** q' (secondary)
| **receives in-order** $\mathsf{Inform}(u)$
| **updates** $\texttt{confirmed} \leftarrow \texttt{confirmed} \cdot u$

Fig. 8. Consistent prefix protocol Φ_b based on primary replication, for some primary process $p \in \mathsf{Pid}$ and secondary processes $\mathsf{Pid}_{\mathrm{sec}} \subseteq \mathsf{Pid}$.

Note that Φ_a is not linearizable, even though it is sequentially consistent. The reason is that it is possible that a read operation o_1 is logically ordered before a write operation o_2 by the order $\rightarrow$ (i.e. the read does not see the write), but that the completion of the write operation $\mathsf{endoperation}(q, \_, o_2\_)$ appears before the beginning (=ending) of the read operation $\mathsf{read}(q, \_, o_1, \_)$ in the execution, thus contradicting the definition of linearizability.

Consistent Prefix Protocol Φ_b. Figure 8 shows another protocol based on primary replication. This time around, the protocol supports availability even in the presence of network partitions: both reads and writes are satisfied locally (assuming that all operations are either read-only or update-only operations). The protocol is similar to Φ_a, but update operations do not block, but allow the client to continue immediately. Update notifications are sent to the primary using in-order delivery, and broadcast back. They are received in-order and appended to the `confirmed` sequence.

The protocol is eventually consistent: we construct the arbitration order the same way as for Φ_a. For the visibility order, we define $\mathsf{vis}^{-1}(o)$ to be $\mathsf{ar}^{-1}(u)$

types
| $\mathsf{Update} = \mathsf{Pid}_{\mathrm{sec}} \times \mathsf{Obj} \times \mathsf{Op} \times \mathbb{N}_0$

process state
| `localcount` : $\mathbb{N}_0$ initially 0 (secondary)
| `pending` : Update^* initially ϵ (secondary)
| `confirmed` : Update^* initially ϵ (secondary)

messages
| $\mathsf{Update}(u : \mathsf{Update})$ //sent from secondary $u.\mathsf{first}$ to primary p
| $\mathsf{Inform}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ //sent from primary p to secondary q'

action $\mathsf{read}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, r : \mathsf{Val})$ **at** q (secondary)
| | **condition** (o is a read-only operation)
| | **condition** $r = \mathcal{S}_{\mathsf{type}(x)}(o, \mathsf{ops}_x(\texttt{confirmed}) \cdot \mathsf{ops}_x(\texttt{pending}))$

action $\mathsf{update}(q : \mathsf{Pid}_{\mathrm{sec}}, x : \mathsf{Obj}, o : \mathsf{Op}, l : \mathbb{N}_0)$ **at** q (secondary)
| | **condition** (o is a update-only operation) $\wedge$ ($l = \texttt{localcount}$)
| | **sends** $\mathsf{Update}(q, x, o, l)$
| | **updates** $\texttt{localcount} \leftarrow \texttt{localcount} + 1; \texttt{pending} \leftarrow \texttt{pending} \cdot (q, x, u, l)$

action $\mathsf{perform}(u : \mathsf{Update})$ **at** p (primary)
| | **receives in-order** $\mathsf{Update}(u)$
| | **sends** $\bigcup_{q' \in \mathsf{Pid}} \mathsf{Inform}(u, q')$

action $\mathsf{learn}(u : \mathsf{Update}, q' : \mathsf{Pid}_{\mathrm{sec}})$ **at** q' (secondary)
| | **receives in-order** $\mathsf{Inform}(u)$
| | **updates** $\texttt{confirmed} \leftarrow \texttt{confirmed} \cdot u$; if $q = q'$ then $\texttt{pending} \leftarrow \texttt{pending.remove}(u)$

Fig. 9. Read-my-writes protocol Φ_c based on primary replication, for some primary process $p \in \mathsf{Pid}$ and secondary processes $\mathsf{Pid}_{\mathrm{sec}} \subseteq \mathsf{Pid}$.

where u is the last operation in `confirmed` at the time o is performed. Thus, the protocol satisfies consistent prefix.

Read-My-Writes Protocol Φ_c. Figure 9 shows yet another protocol based on primary replication. This time, we want to support read-my-writes, so we locally store a sequence `pending` of operations that have been sent to the primary, but not confirmed yet. When performing reads or writes locally, we use not only the updates in `confirmed`, but also append the updates in `pending`.

References

1. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC 2000 (2000)
2. Burckhardt, S.: Consistency in distributed systems. LASER Summer School Slide Decks (2013). http://sdrv.ms/1dWFsBQ
3. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1–2), 1–150 (2014)
4. Burrowsm, M.: The Chubby lock service for loosely-coupled distributed systems. In: Operating Systems Design and Implementation, pp. 335–350 (2006)

5. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer, Heidelberg (2011)
6. Dill, D.L.: The murphi verification system. In: Computer Aided Verification, pp. 390–393 (1996)
7. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**, 374–382 (1982)
8. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)
9. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, p. 11. USENIX Association, Berkeley (2010)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
11. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**, 133–169 (1998)
12. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle foreventual: scalable causal consistency for wide-area storage with COPS. In: SOSP 2011 (2011)
13. Terry, D.: Replicated Data Consistency Explained Through Baseball (2011)
14. Thomas, R.H., Beranek, B.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. **4**, 180–209 (1979)