Algorithmic Basics of Blockchain
Lecture notes

Petr Kuznetsov
Télécom Paris, IP Paris

February 23, 2023

Contents

1. Introduction

What is a blockchain? Depending on where you stand you might want to answer differently. Blockchain turned out to be an amazing phenomenon which enters not only the world of information technology, but also finances, logistics, law, economics, sociology, and even philosophy.

Even in the world of computing systems, there are multiple ways of defining this concept, from software engineering, to programming languages, formal methods and security.

This course is devoted to one particular aspect of the technology, mainly the *algorithmic basics* of blockchain implementations. In other words, we focus rather on the algorithms used to implement blockchains, than on the use of blockchains in high-level applications.

In this perspective, a good way to see blockchain is to imagine a set of users that share a *ledger*, a "Großbuch", that stores a record of their actions. The actions can be *read* and *write* operations on the shared data, transfers of assets between users' accounts, or more sophisticated operations that may involve reading the data and, under some well-specified conditions, modifying it. Such an operations is usually referred as *read-modify-write* and, as we shall see, implementing it in a distributed environment might be quite nontrivial.

Let us assume that there is no central party to store the ledger and the users do not necessarily trust each other. This is a typical assumption in modern computing systems, where system components are subject to failures or even security attacks and there might be no particular parties we can entirely trust.

A standard way to implement a shared data structure in such a system is based on *replication*: every participant stores its own copy (or *replica*) of the shared data. The immediate question is how to guarantee that locally stored replicas of the implemented ledger are *consistently* maintained. In other words, at any moment, every participant should be able to compute from them the "up-to-date" state of the ledger. We shall precisely define what "up-to-date" means in the following chapters but, intuitively, the ledger state should contain all preceding actions and the states computed by different participants should not conflict with each other on the order in which the actions are recorded.

This challenge is addressed by a class of distributed algorithms called *synchronization* algorithms. Informally, in these algorithms, each update of a local copy involves exchange of messages among the participants (via a shared memory or a

computer network).

At a large scale, our goal is to make sure that the set of replicas behave as a *single trusted ledger*, even when some of the replicas do not behave correctly, or even are malicious.

This is a nontrivial challenge, but its nontriviality can be boosted even further if we assume that the system is *open*. The very set of participants is not under the control of a central authority, and everybody can be allowed to join the system and modify the shared data.

Fault-tolerant synchronization algorithms for *permissioned* (with fixed membership) and *permissionless* (with dynamic membership) systems is the primary focus of this course.

Understanding these algorithms allows us not only to reason about advantages and limitations of existing systems, but also to design new, more efficient and reliable ones. We are going to focus on algorithms that underlie the distributed ledger (or blockchain) technology, namely: distributed storage, consensus, replicated state machines, Byzantine fault-tolerance (or BFT), *permissioned* blockchains based on BFT, permissionless blockchains based on proof of work proof of stake, and related topics.

In the coming lectures, we are going to discuss basic model of distributed computing, fundamentals of read-write storage systems, and the folklore "CAP theorem" that, intuitively states that no consistent and available replicated service can tolerate network partitions.

Part I.

Distributed computing in asynchronous systems

2. Distributed storage

Before we plunge into the full-fledged implementations of distributed ledgers, let us consider a simpler abstraction, which is nevertheless very demanded in practice. We are talking about *storage*. For the sake of this lecture, we define a storage system as a collection of shared *variables*. The state of a variable can be fetched by a system participant via a *read* operation and updated by a *write* operation.

An important feature of a *write* operation is that it simply replaces the old state with a new one: every next *write* overwrites the previous one. The process that invokes this operation receives no information on the old state. As a result, the system participants accessing the storage variables with *read* operations may witness different sequences of the variables' states, depending on the order in which these operations are executed with respect to *write* operations. Intuitively, this makes the storage abstraction much weaker that ledger that should, ideally, ensure that the the participants witness the same sequence of operations (or *state transitions*). However, besides having numerous applications, storage can be seen as an intrinsic part of a ledger: after all, the ledger stores *all* written values in an ordered form of a chain of *blocks*. Thus, it is important to understand under which conditions and with which costs storage systems can be implemented. Unlike ledgers, such implementations exist in purely *asynchronous* and *deterministic* models, as we show in this section.

2.1. System model

Before moving further, let us define the model assumptions. We consider a set $\Pi = \{p_1, \ldots, p_n\}$ of *processes*, i.e., independent computing units. Every pair of processes can communicate by sending and receiving messages a dedicated point-to-point *channel*. We assume that the channels are reliable: every sent message is received by its destination after a finite, but *a priori* unknown delay. Besides, the channel is not allowed to duplicate, create or modify messages. (Later we shall discuss ways to relax these assumptions.)

Every process p_i is assigned an algorithm A_i: an automation defined as a set of *states* that the process can take (including its *initial state*), a set of *inputs* it can receive, a set of *outputs* it can produce and a *transition* function that carries each state and input to a new state and the corresponding output. An input can be a call of an operation coming from the application (e.g., a *read*) or a message

received from another process. An output can be a response to an application call (e.g., the read value) or a message sent to another process. In the following, we are going to describe such automata in the form of its *pseudocode*, i.e., an abstract program describing the sequential behavior of a process for each possible input. A *distributed algorithm* (sometimes we simply say *algorithm*) is the tuple $(A_1, \ldots, A_n)$.

The processes are subject of *failures*. Generally, a failure occurs when a process deviates from the algorithm assigned to it. In this chapter we consider the simplest case of such deviation: the faulty process prematurely stops taking steps of its algorithm. This is called a *crash* failure. [1] A process is *correct* if it never fails.

An *execution* of a distributed algorithm (also called a *run*) is a sequence of *events*, i.e., inputs and outputs of the processes that respect the algorithm: a message is sent by a process only if its algorithm prescribes it to do it given the preceding sequence of its inputs. Also, as every received message was previously sent, and no message is received twice.

A typical assumption is that, in every possible execution out of n processes, at most $f < n$ can be faulty. We call such a system f*-resilient*.

2.2. Specifying storage

As we said above, a storage is a collection of shared read-write variables, or *registers*. Let us describe what exactly is meant by a register and define a few variations of its definitions.

A register stores a *value* in a *value set* V and exports two operations: *read* that returns the value and *write*(v), $v \in V$, that replaces the register's value with v and returns *ok*. (For simplicity, we will assume in the following that V is a set of natural numbers.) In a *sequential* execution, when no two operations are applied concurrently.

A *liveness* guarantee of a register implementation stipulates under which condition an operation invoked on it is supposed to return. In this chapter, we require that every operation invoked by a correct process eventually returns. In a system that makes no assumption on the number of failures, this guarantee is known as *wait-freedom* [11], as every process is supposed to *make progress* (to complete its ongoing operation) in a finite number *of its own steps*, regardless of the behavior of other processes, i.e., without waiting for them to take steps.

A *safety* guarantee of a register implementation stipulates what values a *read* operation is allowed to return. In a sequential execution, when no two operations are concurrent, we expect every *read* operation to return *the last written value*.

[1] In the subsequent chapters we shall consider a more general class of *Byzantine* failures, where a faulty process may also perform actions not prescribed by its algorithm, which models, e.g., a process highjacked by a malicious adversary.

Note that the notion of the last written value is not well-defined when the *read* operation is concurrent with a *write* operation, or when some preceding *write* operations are concurrent.

One way to define the behavior of a register implementation in a concurrent system is to require that the *history* of the implementation, i.e., the sequence of invocations and responses of *read* and *write* operations is *linearizable* [13, 4]. A history H is linearizable (with respect to the sequential specification of a read-write register) if all its complete operations and a subset of incomplete operations can be ordered in a sequential history S such that:

- In S every read in returns the last written value, and
- If the response of operation op_1 precedes the invocation of operation op_2 in H, and we write $op_1 <_H op_2$, then $op_1 <_S op_2$.

A linearizable register thus creates an illusion of being accessed with atomic, instantaneous operations: every operation in its history H takes effect in an instantaneous moment of time, a *linearization point*, placed within the operation's *interval* (the period of time between its invocation and response events). The sequence of operations in H placed in the order of their linearization points gives the sequential history S above, called a *linearization* of H. Linearizable registers are also called *atomic* register [14].

Implementing an atomic register might, however, be costly. Therefore, we also consider two weaker register definitions that respect the sequential specification in sequential executions but relax the behavior of a *read* operation that is concurrent with a *write* operation. Both definitions assume the *single-writer* setting, i.e., there is a single dedicated process that is allowed to invoke *write* operations.

Without any additional requirements, these assumptions give us a *safe* register. The register allows *any* value to be returned in case the *read* operation is concurrent with a *write* operation. [2]

A *regular* register, in contrast, is only allowed to return the last written value or a concurrently written value. Here the last written value is the argument of the latest *write* operation that terminated before the invocation of the *read* operation (or the initial value if there is no such *write*). This value is well-defined in the single-writer setting. A concurrently written value is the argument of a *write* operation that is concurrent with the *read*. Remember that there can be multiple such concurrent *write* operations.

A surprising observation is that there exist wait-free transformations of safe registers into regular ones, and regular ones into atomic ones [14]. Hence, *computationally*, the safety criteria listed above (*safety*, *regularity* and *atomicity*) are

[2] In retrospect, the register does not look very safe, as it may produce values that have never been written. But this was the name originally chosen by Lamport for this kind of registers [14] and this is how it stayed in the literature ever since.

equivalent. However, the criteria differ in terms of *complexity* of their implementations.

2.3. The ABD Algorithm

Recall that a storage system can be seen as a collection of read-write registers shared by a set of *clients*, i.e., processes that can access the registers for reading and writing. Now we a re ready to describe an algorithm that implements a *single-writer single-reader* atomic register. As the name suggest, there is exactly one client process that can access the implemented register with a *write* operation and exactly one client process that can access the register with a *read* operation.

The ABD algorithm (for its authors, Hagit Attiya, Amotz Bar-Noy and Danny Dolev [3]) is described in Algorithm 1.

To write a new value v, the (single) writer increments its local sequence number t and sends a *write request* $[v, t]$ to all the processes in the system. The operation completes as soon as responses of the type $[ack, t]$ are received from *a majority* ($> n/2$) of the processes.

To read the register value, the (single) reader increments its local sequence number r and sends a *read request* $[?, r]$ to all the processes in the system. As soon as responses of the type $[t', v', r]$ are received from *a majority* ($> n/2$) of the processes, the operation returns the value equipped with the highest timestamp. Notice that, if the *read* operation is concurrent with a write, it is possible that the reader has previously returned a more recent value than the one obtained from the majority. Therefore, the process compares the received value with the last returned one and updates the latter if necessary.

Theorem 2.1 *A single-writer single-reader atomic register can be implemented in a reliable message-passing model where a majority of processes are correct.*

Proof. Let E be an execution of Algorithm 1 and H be the corresponding history. Every request, issued by the writer (line 8) or by the reader (line 13) expects responses from a majority of processes. As a majority of processes are correct, eventually they receive the request and send back matching responses (in lines 10 and 19). Thus, eventually, every *write* or *read* operation invoked by a correct process completes.

Notice that every written or read value is associated with a distinct timestamp assigned by the writer. Therefore, we can order *write* operations in H in the order of these timestamps. Every *read* operation that returns the value with timestamp t' (line 19) can be then placed right after the corresponding *write* operation (or before the first *write* operation if $t' = 0$). The contiguous *read* operations are then sorted according to the order of their appearance in H. (Recall that we only have

one reader, so the *read* operations are totally ordered by the $<_H$ relation.) Notice that by the construction, the resulting history respects the sequential specification of a register: every read returns the last written value.

As the timestamps of values returned by the reader grow monotonically (line 21), this order does not violate the $<_H$ relation on *read* operations. Thus, the only way to violate the $<_H$ relation in the constructed sequential history S is to reverse the order of a *write* operation and a *read* operation.

Let a *read* operation R precede a *write* operation W in H: $R <_H W$. By the algorithm, the value returned by R must have a timestamp lower than the timestamp used by W, as W has not yet started at this moment. Thus, R will be placed before W in S, i.e., $R <_S W$.

Now let a *write* operation W precede a complete *read* operation R in H. Before returning W made sure that a majority of the processes store the written value v equipped with its with timestamp t. In its turn, R must have accessed a majority of the processes and chooses the value with the highest timestamp among the returned values. As any two majorities intersect, there must be at least one process in the majority of R that sends back v (or a more recent value). Thus, R returns v or a more recent value. By the construction of S, we thus have $W <_S R$.

Hence, S respects $<_H$ and we are done. □

2.4. Exercises

1. Show that the ABD algorithm executed by one writer and *multiple* readers implements a *regular* but not atomic register.

2. Turn the algorithm into an atomic 1WNR (single-writer multi-reader) one.

3. How do we implement an atomic NWNR (multi-writer multi-reader) register?

Algorithm 1 The ABD algorithm

Local variables:
  localValue $= 0$ // locally stored copy, initially 0
  localTS $=$ // largest known timestamp
  $t = 0$ // sequence number of the writer
  $r = 0$ // sequence number of the reader

Upon *write*(v) // the writer's code:
  $t++$ // sequence number of the proposal
  send $[v, t]$ to all
  wait until received $[ack, t]$ from a majority
  return *ok*

Upon *read*(v) // the reader's code:
  $r++$ // sequence number of the proposal
  send $[?, r]$ to all
  wait until received $\{[t', v', r]\}$ from a majority
  let v_m be the value with the highest timestamp t_m
  if $t_m >$ *localTS* **then**
    localValue $:= v_m$
    localTS $:= t_m$
  return *localValue*

Upon received $[v', t']$**:**
  if $t' >$ *localTS* **then**
    localValue $:= v'$
    localTS $:= t'$
    send $[ack, t']$ to the writer

Upon received $[?, r']$**:**
  send $[$*localTS*, *localValue*$, r']$ to the reader

3. Quorum systems

Recall that the ABD algorithm assumes that a majority of processes in the system are always correct. Intuitively, this assumption allows us to ensure that a *read* operation cannot "miss" the value written by a preceding *write*.

In this chapter, we are going to generalize this assumption and introduce the notion an abstract *quorum system*.

3.1. A majority is necessary

First we show that storage cannot be implemented in an f-resilient system if half or more processes can fail, i.e., $f \geq n/2$.

Theorem 3.1 *There does not exist an algorithm implementing a single-writer single reader safe register in an f-resilient system with $f \geq n/2$.*

Proof. Suppose the contrary, assume that p_1 is the reader and p_n is the writer. Consider an execution E_1 of the hypothetical algorithm in which processes in set $\mathcal{F}_1 = \{p_1, \ldots, p_{\lceil n/2 \rceil}\}$ are initially faulty. Let process p_n execute a *write*(1) operation. Since $f \geq n/2$ the operation should eventually terminate.

Now consider an execution E_2 in which processes $\mathcal{F}_2 = \{p_{\lceil n/2 \rceil + 1}, \ldots, p_n\}$ are initially faulty. Let p_1 execute a *read*$()$ operation. Again, since $f \geq n/2$ the operation should eventually terminate and return the initial value 0.

Now we can construct an execution E that is exactly like E_1, except that processes in $\mathcal{F}_1$ are correct, but do not take steps until the *write* operation by p_n terminates. Notice that the operation must terminate, as E is indistinguishable from E_1 to processes in $\mathcal{F}_2$. Then we extend E with the steps of processes in $\mathcal{F}_1$ taken in execution E_2 until the *read* operation by p_1 terminates by returning value 0. In other words, E is is indistinguishable from E_2 to processes in $\mathcal{F}_1$. Notice that in the resulting execution, the messages from $\mathcal{F}_2$ to $\mathcal{F}_1$ are delayed until the *read* operation by p_1 terminates.

But the resulting execution violates the specification of a safe register: the *read* operation running in the absence of concurrency does not return the last written value. □

The set of processes involved in a *read* (respectively, *write*) operation is called a *read* (respectively, *write*) *quorum*. A *read* operation waits until all processes

in a read quorum report about the most recent written value they know. A *write* operation waits until all processes in a write quorum replace their local copies with the new value.

An f-resilient system is uniform in the sense that *any* set of up to f processes might be faulty. It therefore appears reasonable to choose an integer q, $1 \leq q \leq n$, to be the quorum size, i.e., every operation (*read* or *write*) must involve at least q processes.

To ensure a *read* operation does not miss the last written value, we require that every read quorum intersects with every write quorum. Thus every two quorums must have at least one process in common, which implies $q > n/2$.

On the other hand, we want every operation to eventually terminate, i.e., in any execution, at least q processes should be correct. As up to f processes can be faulty, we have $q \leq n - f$, which implies $n - f > n/2$ or $f < n/2$.

3.2. Quorum systems: definitions

A quorum system defined over a system P of processes is a tuple $(\mathcal{R}_P, \mathcal{W}_P)$, $\mathcal{R}_P \subseteq 2^P$, $\mathcal{W}_P \subseteq 2^P$ (two *sets of subsets* of P) such that:

- Quorum Safety: $\forall W \in \mathcal{W}_P, \forall R \in \mathcal{R}_P, W \cap R \neq \emptyset$;
- Quorum Liveness: in every execution of the system, there exist $W \in \mathcal{W}_P$ and $R \in \mathcal{R}_P$ that contain only correct processes.

The safety requirement implies that *reads* and *writes* cannot "miss" each other. The liveness requirements ensures that every operation eventually completes.

Coming back to the f-resilient system, we can define a quorum system as $\mathcal{W}_P = \mathcal{R}_P = \{S \in 2^P : |S| \geq n - f\}$, i.e., the set of process subsets that can be the correct processes of some execution. But a better choice would be $\mathcal{W}_P = \mathcal{R}_P = \{S \in 2^P : |S| = \lfloor n/2 \rfloor + 1\}$, i.e., the set of all majorities. Intuitively, it is better to maintain small quorum sizes as it imposes less waiting on the processes. Notice that when $f \geq n/2$, no quorum system exists.

In the system defined on the set Π of n processes, $p_1, \ldots, p_n$, where either p_1 is guaranteed to be correct in every execution, one can define a quorum system as $\mathcal{W}_P = \mathcal{R}_P = \{\{p_1\}\}$, i.e., every read or write request should go via p_1.

It is quite common not to distinguish between read and write quorums and to assume that $\mathcal{W}_P = \mathcal{R}_P = \mathcal{Q}_P$, i.e., a quorum system is simply a set of process subsets satisfying the two properties above:

- Uniform Quorum Safety: $\forall Q_1, Q_2 \in \mathcal{Q}_P, Q_1 \cap Q_2 \neq \emptyset$;
- Uniform Quorum Liveness: in every execution of the system, there exist $Q \in \mathcal{Q}_P$ that contains only correct processes.

We call such quorum systems *uniform*.

3.3. Byzantine and federated quorums

In Part III we are going to discuss systems where processes are subject to *Byzantine* failures: a faulty process may deviate arbitrarily from the algorithm it is assigned with. To maintain the desired level of consistency, Byzantine quorum systems were introduced. In addition to the Quorum Liveness property above (in every run, at least one quorum is available), a (uniform) Byzantine quorum system $\mathcal{Q}_P \subseteq 2^P$ requires a stronger safety property:

- Uniform Byzantine Quorum Safety: $\forall Q_1, Q_2 \in \mathcal{Q}_P$, $Q_1 \cap Q_2$ contains at least one *correct* process.

Conventional data-replication services are based on quorums [22, 18]. It is assumed that the participants share the global knowledge about the quorum system.

It has been observed that in many realistic scenarios [21, 19], users do not necessarily hold the same assumptions about the subsets of the system that may be trustworthy. Based on its local knowledge, a participant might have its own idea about which subsets of other participants are trustworthy and which are not. This idea has been formalized under the name of *federated* or *decentralized* quorum systems [10, 6, 24].

3.4. Exercises

1. For a fault-free system ($f = 0$), design a *read-optimized* quorum system in which every read operation involves a single replica.

2. In a system of n processes out of which up to f can be faulty, design a quorum system ensuring a stronger property: $\forall W \in \mathcal{W}_P, \forall R \in \mathcal{R}_P, W \cap R$ contains at least one *correct* process. What can we say about the relation between f and n in this case?

4. Lattice agreement

In this chapter, we show that storage systems can be seen as a special case of a more general problem that we call *lattice agreement*. Intuitively, in this problem, processes can *propose* values and *learn* values (somewhat similar to writing and reading). But we also assume that all these values belong to a *lattice*, i.e., a partially ordered set with a well defined *least upper bound* operation.

4.1. Lattice agreement: definition

An abstract (join semi-)*lattice* is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set partially ordered by the binary relation $\sqsubseteq$ such that for all elements of $x, y \in \mathcal{L}$, there exists the least upper bound for the set $\{x, y\}$. The least upper bound is an associative, commutative, and idempotent binary operation on $\mathcal{L}$, denoted by $\sqcup$ and called the *join operator* on $\mathcal{L}$. We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. With a slight abuse of notation, for a set $L \subseteq \mathcal{L}$, we also write $\bigsqcup L$ for $\bigsqcup_{x \in L} x$, i.e., $\bigsqcup L$ is the join of the elements of L. We also assume that the lattice an *origin* element u_0: $\forall u \in \mathcal{L}, u + 0 \sqsubseteq u$.

Notice that two lattices $(\mathcal{L}_1, \sqsubseteq_1)$ and $(\mathcal{L}_2, \sqsubseteq_2)$ naturally imply a *product* lattice $(\mathcal{L}_1 \times \mathcal{L}_2, \sqsubseteq_1 \times \sqsubseteq_2)$ with a product join operator $\sqcup = \sqcup_1 \times \sqcup_2$. Here for all $(x_1, x_2), (y_1, y_2) \in \mathcal{L}_1 \times \mathcal{L}_2$, $(x_1, x_2)(\sqsubseteq_1 \times \sqsubseteq_2)(y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$.

The (generalized) *lattice agreement* abstraction defined on a lattice $(\mathcal{L}, \sqsubseteq)$, exports a single operation *propose* that takes an element of $\mathcal{L}$ as an argument and returns an element of $\mathcal{L}$ as a response.

Once a value is learned, the process is allowed to propose a new one. When the operation *propose*(x) is invoked by process p we say that p *proposes* v, and when the operation returns v' we say that p *learns* v'. Given an execution, let $u_i^1, u_i^2, \ldots$ denote the sequence of values proposed by process p_i and $v_i^1, v_i^2, \ldots$ denote the corresponding sequence of values learnt by process p_i.

Assuming that no process invokes a new operation before its previous operation returns, the abstraction satisfies the following properties:

- **Comparabillity.** The learnt values are totally ordered by $\sqsubseteq$: $\forall i, j, s, t$: $v_i^k \sqsubseteq v_j^\ell \vee v_j^\ell \sqsubseteq v_i^k$.

- **Validity.** every learnt value is the join of a subset of proposed values, including the proposed value and already committed ones.

  Let V_i^k denote the values that were learned before u_i^k was proposed ($\sqcup V_i^k$ is their join). Then: $\forall i, k : (v_i^k \sqsubseteq_{j,\ell} u_j^\ell) \wedge (u_i^k \sqsubseteq v_i^k) \wedge (\sqcup V_i^k \sqsubseteq v_i^k)$.

- **Liveness.** Every *propose* operation invoked by a correct process eventually returns.

Lattice agreement enables natural implementations of objects that can be *updated* (by proposing values) and *read* (by learning values) and for which concurrent updates can be *merged* using a well-defined *join* operation. A typical example is the *add-only set* abstraction which allows values to only be added. Assuming that the set stores integers, its lattice can be just the set of of all sets of integers with $\sqsubseteq = \subseteq$ and $\sqcup = \cup$. Intuitively, when two values a and b are concurrently proposed to be added to an empty set, the corresponding learnt values must belong to one of these sets: $\{\{a\}, \{a, b\}\}$, $\{\{b\}, \{a, b\}\}$. In particular, the learnt values cannot be (incomparable) $\{a\}$ and $\{b\}$: these values cannot be observed in a linearizable set history.

Applications that can benefit from lattice agreement also include *counters*, *atomic snapshots* [1] and *max-registers* [2]. For example, the *max-register* [2] data type with two operations: *writeMax* writes a value and *readMax* returns the largest value written so far. Its state space can be represented as a lattice $(\sqsubseteq, \sqcup)$ of its values, where $\sqsubseteq = \leq$ and $x \sqcup y = \max(x, y)$. Intuitively, a linearizable implementation of max-register must ensure that every read value is a join of previously proposed values, and all read values are totally ordered (with respect to $\leq$).

4.2. Lattice agreement: algorithm

The intuition behind our lattice-agreement implementation (Algorithm 2) is the following. To propose a new value v, a process p proposes it to everybody and waits until it receives responses from a quorum. A response from a process q can be either *ACK*, which means that the lattice value stored at q is v or *precedes* v, or *NACK*, otherwise. Every *NACK* message contains a merge of the local value with v.

If p receives a quorum of *ACK* responses, the value is considered to be installed and the *propose* operation returns. Otherwise, p merges the values received with *NACK* messages with v and repeats the attempt to install the merged value. The procedure is repeated until either (1) a quorum accepts the proposed value, or (2) another process *learns* a value that contains v (the input of p's *propose2* operation

Intuitively, eventually either an attempt to install the current merged value succeeds or some process or some process *learns* a value containing v. Indeed, if no

process concurrently learns new values, eventually all processes with active *propose* operations will converge to the same merged value: the join of their proposed values. As no process will see a conflicting value, all of these *propose* operations will complete. Thus, the only reason that for a *propose* operation invoked by a correct process p to never terminate is that a concurrent process completes infinitely many *propose* operations. But then this concurrent process would eventually include the value proposed by p into its *learnt* value, which will allow p to terminate too.

A formal correctness proof is left as an exercise.

4.3. Applications

Lattice agreement is a powerful abstractions that enables multiple interesting asynchronous fault-tolerant implementations. In this section, we briefly review some of them.

4.3.1. Timestamped value

It is often convenient to equip a value to be stored in a shared variable with a *timestamp*. This will allow to distinguish "recent" values from "old" ones.

Formally, let us define a lattice defined over $\mathcal{L}_{TV} = \mathbb{N} \times V$, where $\mathbb{N}$ is the set of natural timestamps and V is the *value set*. We assume that V is provided with a total order $\leq_V$ and the lowest value v_0. We define a (total) lexicographic order $\sqsubseteq_{TV} = (\leq_{\mathbb{N}}, \leq_V)$: $(t, v) \sqsubseteq_{TV} (t', v')$ if $t < t'$ or $t = t'$ and $v \leq_V v'$. Respectively, $(t, v) \sqcup_{TV} (t', v')$ is the highest tuple of the two with respect to $\sqsubseteq_{TV}$.

A *timestamped value* can be *read* and *updated* with a new one. In a sequential execution, every *read* operation returns the *highest* written value so far.

We can be easily implement a linearizable timestamped value using lattice agreement on the afore defined lattice as follows. To update the timestamped value, a process simply executes *propose*$((t, v))$. Intuitively, if t is sufficiently high, the shared timestamped value will be updated. To read the timestamped value, a process executes *propose*$((0, v_0))$ and returns the learnt value.

4.3.2. Atomic snapshot

An atomic snapshot object maintains a vector of m values and two operations *update* and *snapshot*. In a sequential execution, *update*(v, i) replaces position i in the vector with v and *snapshot* that returns the vector. We assume that the set V of values is totally ordered and equipped with an initial value v_0.

In the concurrent implementation, every position vector $i = 1, \ldots, m$ is represented as a timestamped value. Let us consider $\times_{i=1,\ldots,m}(\mathcal{L}^i_{TV}, \sqsubseteq_{TV^i}, \sqcup_{TV^i})$, the composition of lattices for these timestamped values.

To take a snapshot, a process simply executes *propose*(u_0), where u_0 is the initial value of the composed lattice and returns the result of the *propose* operation.

To update position i with new calue v, the process first takes a snapshot. Let $(t_1 v_1), \ldots, (t_i, v_i), \ldots, (t_m, v_m)$ be the returned value. The process then executes *propose*$((t_1 v_1), \ldots, (t_i + 1, v), \ldots, (t_m, v_m))$.

4.4. Exercises

1. Show that the weak counter object (with *inc* and *read* operations) can be implemented via Lattice Agreement. Find the matching lattice.

2. What if in Algorithm 2, we allow a process to return a value v' once it verifies that v' succeeds its input v in the lattice partial order? In other words, what if we replace the condition *proposedValue* $\sqsubseteq v'$ in line 32 with $v \sqsubseteq v'$? Can you construct a counter-example: an execution that violates the properties of lattice agreement?

3. What if we drop the condition *readyToLearn* in line 32 of Algorithm 2.

4. Complete the proof of correctness of the LA implementation (Algorithm 2).

Algorithm 2 Lattice agreement

Local variables:
  $bufferedValue = u_0$ // join of known proposals
  $proposedValue = u_0$ // the current proposal
  $acceptedValue = u_0$ // join of accepted proposals
  $readyToLearn = false$ // ready to adopt a learn value

Upon $propose(v)$ // **process** p **proposes** v**:**
  $t++$ // sequence number of the proposal
  $readyToLearn := false$
  $proposedValue := v$
  $bufferedValue := bufferedValue \sqcup v$
  send $[PROPOSE, v, t, p]$ to all

Upon received $[PROPOSE, v', t', p']$**:**
  if $acceptedValue \sqsubseteq v'$ **then**
    $acceptedValue := v'$
    send $[ACK, v', t', p']$ to p' // accept the proposal
  else
    $acceptedValue := acceptedValue \sqcup v'$
    send $[NACK, acceptedValue, t', p']$ to p' // reject the proposal

Upon received $[ACK/NACK, v', t, p]$ **from a quorum:**
  if no $[NACK, v', t, p]$ received **then**
    send $[LEARN, bufferedValue]$ to all
    return $bufferedValue$ // learn a new value
  else
    $t++$
    if not $readyToLearn$ **then**
      $readyToLearn := true$
      $proposedValue := bufferedValues$
    send $[PROPOSE, bufferedValue, t, p]$ to all // send a new proposal

Upon received $[NACK, v', t, p]$**:**
  $bufferedValue := bufferedValue \sqcup v'$

Upon received $[LEARN, v']$**:**
  if $proposedValue \sqsubseteq v'$ and $readyToLearn$ **then**
    send $[LEARN, v']$ to all // "reliable" broadcast
    return v' // adopt a new value

Part II.

State-machine replication and Paxos

5. Consensus and replicated state machine

Until now, we focused on systems that can be implemented in a purely asynchronous fault-tolerant environments: storage and lattice agreement. To achieve consistency in the presence of failures, these implementations assume a set of quorums (process subsets) that is *consistent* (every two quorums overlap) and *available* (in every execution, some quorum only contains correct processes).

Let us now come back to our original problem: how to implement a distributed ledger. We begin with *consensus*, the problem that lies at the heart of ledger implementations.

First, we formally define the consensus problem that consists for a set of processes We then recall the impossibility of asynchronous fault-tolerant consensus [17, 8].

We then discuss a weaker abstraction, *commit-adopt* (CA), that, in contrast, can be implemented in this model, assuming an underlying read-write storage system. Intuitively, commit-adopt always ensures safety properties of consensus but liveness is only guaranteed when no two different values are proposed. Surprisingly, commit-adopt, combined with Ω, the *leader-election* oracle, allows for implementing consensus.

We then introduce a more convenient abstraction, *obstruction-free consensus* (OFC) that can be accessed with proposed values multiple times and ensures that, if there is a time after which it is only accessed by a single process, then a value is decided. Consensus on top of OFC can be implemented more efficiently than using CA and read-write registers.

5.1. Consensus

In the consensus problem, processes *propose* values in an *input* set V and *decide* on values in V. Every process proposes at most once and decides at most once.

An algorithm solves consensus if the following properties are satisfied:

Agreement No two processes decide differently.

Validity Every decided value was previously proposed.

Termination Every correct process eventually decides.

Consensus, as we recall in the next section, can be used to implement a *replicated state machine*. So, in a certain sense, consensus is a *universal* abstraction. Should consensus be implementable in an *asynchronous* and *fault-tolerant* way, we would get a way to construct a fault-tolerant and asynchronous implementation of *any* service given its sequential specification. In Section 5.2, we describe one way to do it.

Unfortunately, there does not exist an asynchronous consensus algorithm that tolerates just a single faulty process, as has been originally shown by Fisher, Lynch and Paterson [8] for the message-passing model. Loui and Abu-Amara [17] generalized this result to the read-write shared-memory model.

There are two major ways to circumvent this impossibility: to relax the problem or to strengthen the model. For example, one can relax termination by saying that a correct process must eventually decide *with probability* 1. Indeed, starting from the breakthrough paper by Ben-Or [5], a large range of *randomized* consensus algorithms was proposed. We can also require progress to be made by a correct process in a *subset* of executions, and in Chapter 6, we discuss one example of such a relaxation—obstruction-free consensus (OFC).

To strengthen the model, we typically assume that the system is not purely asynchronous. A convenient way to add *some* synchrony to the system is to assume an *oracle* that provides the processes with hints about failures that currently happen. Such oracles are sometimes called *failure detectors*, as suggested by Hadzilacos and Toueg [7]. In Section 6.2, we discuss how the *eventual leader* oracle Ω can be help in transforming OFC into consensus.

5.2. State-machine replication

In this section, we show that if, in a system of n processes, one can solve consensus, then it is possible to implement *any* object given its sequential specification. Following the original algorithm proposed by Herlihy [11], we call this implementation *universal construction*.

An *object type* can be represented as a tuple (Q, q_0, O, R, δ), where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and δ is a binary relation on $O \times Q \times R \times Q$, total on $O \times Q$: $(o, q, r, q') \in \delta$ if operation o is applied when the object's state is q, then the object *can* return r and change its state to q'. Note that for *non-deterministic* object types, there can be multiple such pairs (r, q') for any given o and q.

Given an object type $\tau = (Q, q_0, O, R, \delta)$, we provide a linearizable implementation of τ using reliable communication channels and atomic consensus objects.

For simplicity, we assume that deterministic object types, δ can be seen as a function $O \times Q \rightarrow R \times Q$ that associates each state and operation with a unique response and a unique resulting state. The state of a deterministic object is thus determined by a sequence of operations applied to the initial state of the object. The universal construction of an object of a deterministic type is presented in Figure 5.1

Shared abstractions:
$C_1, C_2, \ldots$, instances of consensus implementations

Local variables, for each process p_i:
integer seq_i, initially 0 { *the number of executed requests of* p_i }
integer k_i, initially 0 { *the number of batches of executed requests* }
sequence $linearized_i$, initially empty { *the sequence of executed requests* }
set $published_i$, initially empty { *the set of published requests* }

Code for operation op executed by p_i:
1 $seq_i \leftarrow seq_i + 1$
2 send (op, i, seq_i) to all { *publish the request* }
3 $published_i \leftarrow published_i \cup \{op, i, seq_i)\}$
4 **repeat**
5 $requests \leftarrow published_i - \{linearized_i\}$ { *choose not yet linearized requests* }
6 $k_i \leftarrow k_i + 1$
7 $decided \leftarrow C[k_i].propose(requests)$
8 $linearized_i \leftarrow linearized_i.decided$ { *append decided requests* }
9 **until** $(op, i, seq_i) \in linearized_i$
10 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Upon receive (o, j, s) { *A new request published* }
11 $published_i \leftarrow published_i \cup \{(o, j, s)\}$

Figure 5.1.: Universal construction for deterministic objects

Every process p_i maintains local variables $published_i$ that stores the published requests p_i is aware of, $linearized_i$ that stores a sequence of operations that p_i executed on the implemented object do far, and seq_i that contains the number of operations invoked by p_i Whenever p_i has a new operation op to be executed on the implemented object, p_i "publishes" (op, i, seq_i) by sending it to everybody. As long as p_i is aware of operations that were invoked (by p_i itself or any other process) , p_i tries to agree on the order in which operations must be executed by using the "next" consensus object $C[k_i]$ that was not yet accessed by p_i. If the set of operations returned by $C[k_i]$ contains op, p_i deterministically computes the response of op using the specification of the implemented object and $linearized_i$.

Otherwise, p_i proceeds to consensus object $C[k_i + 1]$.

As a result, the processes agree on the evolution of the implemented object's state: thanks to the use of consensus instances, they execute operations on it in the same order.

5.3. Exercises

1. Prove that the algorithm in Figure 5.1 is a linearizable implementation of an object type (Q, q_0, O, R, δ).

6. Solving consensus

The goal of this chapter is to describe a consensus algorithm in the eventually synchronous message-passing model, assuming that a majority of processes are correct. But before presenting the algorithm, we make a short detour and describe a similar, slightly simpler abstraction of *commit-adopt*. We show that the abstraction can be implemented in read-write shared memory model in the *wait-free* manner, i.e., assuming that any number of processes can fail.

6.1. Commit-Adopt

The *commit-adopt* abstraction [9] can be seen as a relaxation of consensus. When accessed concurrently by multiple processes, it may return a special *abort* response. In the absence of contention, the abstraction must eventually return a decided value.

6.1.1. Definition

CA, like consensus, exports one operation *propose*(v) that, unlike in consensus, returns $(commit, v')$ or $(adopt, v')$, where v' and v are in a (possibly unbounded) *value set* V. If *propose*(v) invoked by a process p_i returns $(adopt, v')$, we say that p_i *adopts* v'. If the operation returns $(commit, v')$, we say that p_i *commits* on v'. Intuitively, a process commits on v' when it is sure that no other process can commit on a value different from v'. A process adopts v' when it suspects that another process might have committed v'.

Formally, CA guarantees the following properties:

C1 every returned value is a proposed value,

C2 if all processes propose the same value then no process adopts,

C3 if a process commits on a value v, then every process that returns adopts v or commits on v, and

C4 every correct process returns.

```
Shared objects:
    A, B, store-collect objects, initially ⊥;

propose(v)
1   est ← v;
2   A.store(est);
3   V ← A.collect();
4   if all values in V are est then
5       B.store([true, est]);
6   else
7       B.store([false, est]);
8   V ← B.collect();
9   if all values in V are [true, ∗] then
10      return (commit, est)
11  else if V contains (true, v′) then
12      est ← v′;
13  return (adopt, est)
```

Figure 6.1.: A commit-adopt algorithm

Implementing Commit-Adopt. The commit-adopt abstraction can be implemented using two (wait-free) *store-collect* objects, A and B, as follows. Recall that a store-collect object A can be seen as an array of registers $A[1], \ldots, A[n]$. To store a value v in A, process p_i writes v in $A[i]$, and to collect it reads $A[1], \ldots, A[n]$ and returns the resulting vector of values.

In our commit-adopt implementation, when p_i proposes a value v, it first stores v in A and then collects A. If no value other than v was found in A, p_i stores $id[true, v]$ in B. Otherwise, p_i stores $[false, v]$ in B. If all values collected from B are of the form $[true, *]$, then p_i commits on its own input value. If this is not the case and at least one of the collected values is $[true, v']$, then p_i adopts v'. Intuitively, going first through A guarantees that there is at most one such value v'. If p_i cannot commit or adopt a value from another process, it simply adopts its own input value.

Theorem 6.1 *The algorithm in Figure 6.1 implements commit-adopt.*

Proof We show that the algorithm satisfies properties C1-C4 of commit-adopt.

Property C1 follows trivially from the algorithm and the Validity property of store-collect (see Section **??**): every returned value was previously proposed by some process. If all processes propose the same value, then the conditions in the clauses in lines 4 and 9 hold true and, thus, every process that returns must commit—property C2 is satisfied. Property C4 is implied by the fact that the algorithm contains only finitely many steps and every store-collect object is wait-free.

To prove C3, suppose, by contradiction, that two processes, p_i and p_j, store two different values, v' and v'', respectively, equipped with flag $true$ in B (line 5). Hence, the collect operation performed by p_i in line 3 returns only values v. By the up-to-dateness property of store-collect and the algorithm, p_i has previously stored v' in A (line 2). Similarly, p_j has stored v'' in A.

Again, by the up-to-dateness property of store-collect, the $A.store(v'')$ operation performed by p_j does not precede the $A.collect()$ operation performed by p_i. (Otherwise p_i would find v'' in A.) Hence, $inv[A.collect()]$ by p_i precedes $resp[A.store(v'')]$ by p_j in the current execution. But, by the algorithm $resp[A.store(v')]$ precedes $inv[A.collect()]$ at p_i and, $resp[A.store(v'')]$ precedes $inv[A.collect()]$ at p_j. Hence, $resp[A.store(v')]$ by p_i precedes $inv[A.collect()]$ by p_j and, by up-to-dateness of store-collect, p_j should have found v' is A—a contradiction.

Therefore, no two different values can be written to B with flag $true$. Now suppose that a process p_i commits on v. If every process that returns either commits or adopts a value in line 12, then property C3 follows from the fact that no two different values with flag $true$ can be found in B. Suppose, by contradiction that some process p_j does not find any value with flag $true$ in B (9) and adopts its own value. By the algorithm, p_j has previously stored $(false, v'')$ in line 7. But, again, $B.store([true, v'])$ performed by p_i does not precede $B.collect()$ performed by p_j, thus, $B.store((false, v''))$ performed by p_j precedes $B.collect()$ performed by p_i. Hence, p_i should have found $(false, v'')$ in B—a contradiction. Therefore, if a process commits on v', no other process can commit on or adopt a different value—property C3 holds. $\square_{Theorem\ 6.1}$

6.1.2. Solving Consensus with Commit-Adopt and Ω

Commit-adopt can be viewed as a way to establish *safety* in shared-memory computations.

For example, consider a protocol where every process goes through a series of instances of commit-adopt protocols, $CA_1, CA_2, \ldots$, one by one, where each instance receives a value adopted in the previous instance as an input (for CA_1—the initial input value). One can easily see that once a value v is committed in some CA instance, no value other than v can ever be committed (properties C1 and C3 above). On the other hand, if at most one value is proposed to some CA instance, then this value must be committed by every process that takes enough steps (property C2 above).

This algorithm can be viewed as a *safe* version of consensus: every committed value is a proposed value and no two processes commit on different values (properties C1, C2, and C3 above). Given that every correct process goes from one CA instance to the other as long as it does not commit (property C4 above), we can

boost the liveness guarantees of this protocol using external oracles.

In fact, the algorithm *per se* guarantees termination in every *obstruction-free* execution, i.e., assuming that eventually at most one process is taking steps. Note that, as its liveness properties are only guaranteed in obstruction-free runs, the algorithm is not subject to the FLP impossibility proof (see Exercise **??**).

Moreover, we can build a consensus algorithm that terminates *almost always* if we allow processes to toss coins when choosing an input value for the next CA instance. Similarly, one can (deterministically) solve consensus using the Ω failure detector: before going to the next CA instance, every process waits until the "leader" (provided by Ω) writes its current value. The value of the leader is then used as a proposal for the next CA instance. The reader is invited to sort out the details of this algorithm (see Exercise 1).

6.2. Obstruction-free consensus

We now describe *obstruction-free consensus* (OFC), a variant of consensus that can be implemented in the asynchronous model. Similarly to the CA abstraction, it is accessed with a $propose$ operation that, under some unfavourable conditions, may not return a decided value. Unlike the CA abstraction, OFC is not *one-shot*: it can be accessed with a $propose$ operation an unbounded number of times, until a value is decided.

OFC abstract away *Synod*, the "optimistic" consensus protocol in Paxos [15] state-machine replication protocol.

6.2.1. Definition

Recall that Synod implements *obstruction-free consensus* exporting one operation, *propose*, that takes an *value* in some *value set* V as an argument and returns a value in V or a special value *abort* $\notin V$ as a response. If an operation invoked by process p returns $v \in V$, we say that p *decides* v. If an operation returns *abort*, we say that p *aborts*. The abstraction is *long-lived*: a process can invoke the *propose* operation as many times as it wants.

The abstraction provides the following guarantees:

Validity Every decided value is a proposed value.

Agreement No two processes decide differently.

OF-Termination

1. If a correct process p_i proposes, it eventually decides or aborts.
2. If a correct process decides, no correct process aborts infinitely often.

3. If there is a time after which a single correct process p_i proposes a value sufficiently many times, p_i eventually decides.

Notice that here the term *obstruction-freedom* is used in a slightly different way, compared to the conventional *shared-memory* definition that guarantees progress under the assumption that exactly one process is taking steps of from some point on [12]. Here we guarantee progress under the assumption that exactly one process *proposes* from some point on, while still expecting that all other correct processes keep taking steps of the algorithm (without proposing). In particular, every correct process is expected to respond to request messages sent by the proposer.

6.2.2. Synod: implementing OFC

The pseudocode of the Synod algorithm is presented in Algorithm 3.

To propose a value a process p_i selects the next *distinct* ballot number (line 2) and asks a majority of processes to accept or reject it (line 6). If some process has seen a larger or equal ballot number (either in the *read* phase or in the *impose* phase), the *propose* operation aborts (line 14). Otherwise, we check if some process in the majority has a decision estimate with a positive *impose ballot* number (line 18) and if so, the process selects the value with the highest impose ballot number as its *proposal* (line 19).

Then p_i issues an *impose* request (line 22). Again, if some process responds by saying that it has seen a higher read or impose ballot number, the operation aborts. Otherwise, p_i sends its proposal to all and decides (line 33).

Validity. The Validity property follows immediately from the algorithm: only proposed value can be decided.

Agreement. To prove the Agreement property, suppose that p_i is the process to decide a value v with the *lowest* associated ballot number b. Notice that every process has its own (disjoint) set of ballot numbers to choose, so such a process is well-defined.

Let p_j be the process that sends a $[IMPOSE, b', v']$ message (line 22) such that b' is the lowest ballot number higher than b attached to an *IMPOSE* message in the considered execution. We are going to show that v' must be v. By induction, this will imply that no two processes decide different values in any execution of our algorithm, as before deciding on a value v' with ballot b', p_j must have previously sent the corresponding *IMPOSE* message.

By the algorithm, before deciding, p_i has successfully completed its *impose* phase, and a majority of processes must have previously adopted p_i's impose ballot number b (line 27). As p_j, before sending an *IMPOSE* message with impose ballot b', has successfully completed its *read* phase, a majority of processes must have previously accepted p_j's read ballot number b' (line 11). Recall that, by our assumption, $b' > b$.

Let p_k be any process in the intersection of these two majorities. As $b < b'$, p_k must have received the impose request from p_i *before* the read request from p_j. Indeed, otherwise p_k would not accept the impose request from p_i with ballot number b as it previously adopted a higher ballot number b'. Thus, p_i must have received a *GATHER* response from p_k containing (b, v) as $(imposeballot, estimate)$ (line 16). As b' is the lowest ballot higher than b attached to an *IMPOSE* message, no process in the majority responding to p_j's read request can send a *GATHER* message containing (b'', v'') such that $b < b'' < b'$. Thus, p_j will select v as its proposal for the *impose* phase (line 19). Hence, $v = v'$ and we have the Agreement property.

OF-Termination. Part (1) of the OF-Termination property follows directly from the fact that a majority of processes are correct: eventually, every read or impose request reaches a majority of processes that will respond by sending either an *ABORT* message (resulting in aborting the *propose* operation) or a "meaningful" response) resulting in deciding.

Since a correct process sends its decision value to all before deciding (line 32), we also have part (2) of the OF-Termination property.

Finally, if there is a time after which, there is a single process p_i to propose, then, eventually, no process will see a read or impose ballot equal or higher to the ballot numbers issued by p_i (lines 8 and 24). Indeed, from some point on, p_i will be the only process in the system to increase its ballot numbers (line 2). Thus, eventually p_i will complete both read and impose phases of the *propose* operation and decide (line 33), which implies part (3) of OF-Termination.

6.2.3. Consensus with OFC and Ω

Notice that the only reason for a process to abort its *propose* operation is the presence of concurrent operations. If there is a time after which no concurrent operations are invoked, the remaining correct process must decide after finitely many attempts.

One can easiy ensure this condition in every execution by allowing a process to invoke a *propose* operation *only* if the process *considers itself a leader*, i.e., Ω outputs its identifier. If the operation aborts, the process waits until Ω outputs its identifier and invokes a new instance of *propose*. There is a time after which,

there will be exactly one correct process that considers itself leader. The leader is eventually guaranteed to decide. One it does, it sends the decided value to everybody. Once a process receives the decided value it Proving correctness of this consensus algorithm is left as a simple exercise (Exercise 2).

6.3. Atomic broadcast vs. replicated state machines

State-machine replication (SMR) can be generalized even further. Instead of reaching agreement on the order in which operations are executed on the implemented state machine, we can agree on the order in which abstract *messages* are delivered. As we shall see, this is closer to the abstract notion of a distributed ledger implemented by blockchain algorithms.

The *atomic broadcast* abstraction export a call *broadcast*(m) (to broadcast a message m) and an up-call *deliver*(m). Assuming that all broadcast messages are distinct (e.g., carry unique identifiers), it guarantees the following properties:

Validity: if a correct process invokes *broadcast*(m), then eventually every correct process executes *deliver*(m);

No duplication: for a given message m, a process executes *deliver*(m) at most once;

No creation: if a process executes *deliver*(m), then some process previously executed *broadcast*(m);

Total order: if a process delivers message m and then message m', then no process delivers m' before m.

To implement a replicated state machine using atomic broadcast, every process p_i broadcasts its requests r equipped with its identifier. As soon as a new request is delivered, the process applies it to the current state of its local copy of the state machine. If the request is equipped with the identifier of p_i, the corresponding response is returned to the application.

To implement a broadcast from state machine replication, we can consider a *ledger* state machine that maintains a sequence of *messages* as a state and exports two operations: *append*(m) that appends message m to the end of the sequence and *read*$()$ that returns the sequence. To broadcast a message m, a process simply calls *append*(m). In parallel, the process periodically invokes *read*$()$ and delivers *new* messages in the returned sequence in the order of their appearance.

Proving correctness of these two reduction algorithms is left to Exercise 3.

6.4. Exercises

1. Give an algorithm that solves consensus using read-write registers and the Ω failure detector.

2. Prove that the algorithm sketched in Section 6.2.3 indeed solves consensus using OFC and Ω.

3. Show that SMR and AB are equivalent: one can implement SMR from AB, and vice versa.

Algorithm 3 Synod: code for process p_i

1: **Local variables:**
2: *ballot* $:= i - n$; *proposal* $:=$ *nil*; *readballot* $:= 0$;
3: *imposeballot* $:= i - n$; *estimate* $:=$ *nil*; *states* $:= [nil, 0]^n$

4: **Upon** ***propose***(v) **// process** p_i **proposes** v**:**
5: *proposal* $:= v$; *ballot* $:=$ *ballot* $+ n$; *states* $:= [nil, 0]^n$
6: send $[$*READ*, *ballot*$]$ to all

7: **Upon received** $[$***READ***, ***ballot***$']$ **from** p_j**:**
8: **if** *readballot* $>$ *ballot*$'$ or *imposeballot* $>$ *ballot*$'$ **then**
9: send $[$*ABORT*, *ballot*$']$ to p_j // reject the ballot number
10: **else**
11: *readballot* $:=$ *ballot*$'$
12: send $[$*GATHER*, *ballot*$'$, *imposeballot*, *estimate*$]$ to p_j // accept the ballot number

13: **Upon received** $[$***ABORT***, ***ballot***$]$ **from some process:**
14: return *abort*

15: **Upon received** $[$***GATHER***, ***ballot***, ***estballot***, ***est***$]$ **from** p_j**:**
16: *states*$[p_j] :=$ *estballot*, *est*$]$

17: **Upon** $|$***states***$| > n/2$**:** // received a majority of responses
18: **if** $\exists$*states*$[p_k] =$ *estballot*, *est*$]$ with *estballot* > 0 **then**
19: select *states*$[p_k] = [est, estballot]$ with highest *estballot*
20: *proposal* $:=$ *est* // choose a potentially decided value
21: *states* $:= [nil, 0]^n$
22: send $[$*IMPOSE*, *ballot*, *proposal*$]$ to all

23: **Upon received** $[$***IMPOSE***, ***ballot***$'$, $v]$ **from** p_j **:**
24: **if** *readballot* $>$ *ballot*$'$ or *imposeballot* $>$ *ballot*$'$ **then**
25: send $[$*ABORT*, *ballot*$']$ to p_j
26: **else**
27: *estimate* $:= v$; *imposeballot* $:= ballot'$
28: send $[$*ACK*, *ballot*$']$ to p_j

29: **Upon received** $[$***ACK***, ***ballot***$]$ **from a majority:**
30: send $[$*DECIDE*, *proposal*$]$ to all

31: **Upon receive** $[$***DECIDE***, $v]$**:**
32: send $[$*DECIDE*, $v]$ to all
33: return v

Part III.

Byzantine failures and permissioned blockchains

7. Byzantine fault tolerance

In 967AD, the emperor (basileus) of the Byzantine Empire Nikethoros II decides to resolve the "Bulgarian issue" and finally integrate the rebellion kingdom into the empire. Instead of engaging the imperial army into an open conflict, the basileus decides to delegate the task to Sviatoslav I, the grand prince of Kiev Rus'. To this purpose, Nikethoros sends Kalokir to Kiev with a mission to buy Sviatoslav's military services, in exchange for half a ton of gold. Kalokir, secretly wishing to usurp the throne, convinces Sviatoslav to turn against Nikethoros. In turn, Sviatoslav chooses to seize the opportunity to conquer Bulgaria and keep it to himself. That was the beginning of a war between the three nations.

The term *Byzantine failure*[1] intends to capture *arbitrary* deviations of faulty processes from their expected behavior, prescribed by the algorithm assigned to them. Until know, we focused on fault-tolerant distributed systems, where the only deviation a faulty process can exhibit is a *crash*: the process simply stops taking steps from some point on. In contrast, a Byzantine process may take "incorrect" steps: for example, it may send different messages to different processes, while its algorithm prescribes sending the same message to all. This kind of behavior is known as *equivocation*, and it is considered potentially dangerous for most protocols based on broadcasting messages, such as storage, lattice agreement and consensus.

7.1. Replicated services

In this chapter, we discuss protocols that tolerate Byzantine failures. Our primary application is a *replicated service*. Imagine a set of a collection of *clients* that execute operation on a shared object with a given sequential specification. To make the object fault-tolerant, we *replicate* its state over a set of servers that we call *replicas*.

[1] The term *Byzantine* was introduced to the distributed computing community in 1982 by Lamport, Shostak and Pease [16] who condired the *Byzantine generals problem* . Allegedly, they originally were going to talk about Albanian generals but then decided to cast the problem to the past.

Notice that we separate the entities of the algorithm that interact with the application (clients) from the entities that maintain the shared object state (replicas). In practice, this separation can be *logical*: the two entities may run on the same machine. However, the separation is important, as it allows us to define different failure assumptions for the clients and for the replicas. The algorithm we are going to consider will be able to tolerate an arbitrary number of faulty clients, but only a bounded fraction of faulty replicas.

On the high level, a replicated service typically operates as follows. To perform an operation on the object, every client sends a *request* message (containing the operation, the client identifier and a sequence number) to all replicas and enters a *waiting mode*. The replicas engage in an agreement protocol to decide in which order the operations proposed by different clients must be ordered. Once the client's operation is *decided* (its postion is the order is determined), the replicas send a *response* message (containing the request and the result of operation) back to the client. The client then returns the result of the operation beck to the application.

We expect that the correct clients get *consistent* service. Intuitively, there must exist a sequential history S that is indistinguishable from the actual history of invocations and responses *for every correct process*. The criterion is similar to linearizability [13, 4], except that here we do not restrict the local histories of faulty (Byzantine) processes. Indeed, as Byzantine processes may arbitrarily deviate from the algorithm, we cannot restrict their behavior anyhow. Furthermore, we expect that every correct client with a pending operation is eventually "served"—the operation is provided with a response. As we shall see below, to meet these requirements, safety and liveness, one has to assume that the number f of faulty replicas should be less than one third of the total number of replicas n: $f < n/3$ (see Exercise 2 in Chapter 3.

7.2. Cryptographic primitives

Arguably, the most dangerous behavior a Byzantine process may exhibit is the one that is indistinguishable from that of a correct process, to some correct process. In particular, a Byzantine process may lie about having received certain messages from other processes. One may insulate this type of misbehavior, we typically rely on *cryptographic primitives*.

More precisely, we assume an *asymmetric* cryptographic system. Each process owns a *public* key and a *private* key. Public keys are common knowledge, while private keys are not supposed to be revealed to other processes. A process may equip every message it emits with a *digital signature*, constructed using its private key. The signatures can be *verified* using the public key, i.e., every other process can check its *authenticity*, i.e., it is indeed sent by the process that is claimed

to send it. The signatures are unforgeable: it is computationally infeasible to construct a signature of a process without possessing its private key. Starting from the next chapter, we are going to assume that every message sent by a correct process is properly signed.

7.3. Synchronous Byzantine agreement with non-authenticated channels

To get a glimpse of why it is difficult to reach agreement in a distributed system, consider first a *synchronous* system, where both the communication delay and the relative local computation speeds are bounded, and the bounds are known to the processes. For simplicity, assume that computation unfolds in *synchronous rounds*. At the beginning of round r a process, sends its round-r message to every other process and by the end of the round it is guaranteed to receive round-r messages from *every correct process*.

Let us consider a system of three processes, C (Commander), L_1 (Lieutenant 1) and L_2 (Lieutenant 2), that intend to solve the *coordinated attack* problem. In the problem, Commander issues a *command*: *attack* or *retreat* and the goal for honest participants is to agree on a command. The difficulty here is that Commander or one of the lieutenants might be Byzantine. In particular, the Byzantine Commander may *equivocate* by sending different commands to the two Lieutenants. The following scenario illustrates a fundamental difficulty of reaching agreement:

(1) C is Byzantine: it sends command *retreat* to L_1 and command *attack* to L_2.

(2) L_1 and L_2 exchange the received commands with each other.

(3) L_1 cannot distinguish the execution from that in which C is correct and proposes *retreat*, while L_2 is Byzantine and only *claims* that it received command *attack* from C.

This scenario has been used to show that no system can reach agreement in a system of n processes out of which f can be Byzantine if $n \leq 3f$.

Of course, this scenario assumes that no third party can verify the *authenticity* of a message, i.e., to make sure that a forwarded message has indeed been sent the process who is claimed to be its sender. Asymmetric cryptography gives us an authentication mechanism: a process may *sign* its messages using its *private key* (kept in secret) and any third party can *verify* the signature using the process' *public key* (publicly known).

In the scenario above, Lieutenant 1 can then detect the misbehavior of Commander. Assuming that every process signs all its outgoing messages, the lieutenant will witness two conflicting messages from C. In a synchronous system,

authentication allows us to reach agreement in $O(f)$ rounds, for all $f < n$. In a *partially synchronous* system, however, we still need an additional assumption on n and f, as we show below.

7.4. Authenticated channels and Byzantine quorums

7.5. Further reading

The problem of reaching agreement in the presence of arbitrary faults was introduced by Robert Shostak in 1978 under the name of *interactive consistency* [23]. The lower bound of $3f + 1$ replicas to tolerate f Byzantine failures was established by Pease, Lamport and Shostak in 1980 [20] for the synchronous network without the use of asymmetric cryptography.

7.6. Exercises

1.

8. PBFT: Practical Byzantine Fault Tolerance

8.1. Further reading

8.2. Exercises

9. Hyperledger Fabric

9.1. Execution and ordering

9.2. Further reading

9.3. Exercises

Part IV.

Permissionless Blockchains

10. Proof of work and Bitcoin

11. Proof of stake and Casper

11.1. Stake assumptions

11.2. Casper: safety

11.3. Casper: liveness

11.4. Further reading

12. Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *PODC*, pages 36–45, 2009.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *J. ACM*, 42(2):124–142, Jan. 1995.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, 2004.

[5] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *PODC '83: Proceedings of the annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

[6] C. Cachin and B. Tackmann. Asymmetric distributed trust. In *OPODIS*, 2019.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] E. Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

[10] Á. García-Pérez and A. Gotsman. Federated byzantine quorum systems. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, pages 17:1–17:16, 2018.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, 1991.

[12] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[13] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[14] L. Lamport. On interprocess communication; part I: Basic formalism; part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[15] L. Lamport. The Part-Time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[17] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[18] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(?):203–213, 1998.

[19] D. Mazieres. The stellar consensus protocol: A federated model for internet-level consensus, 2016.

[20] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[21] D. Schwartz, N. Youngs, and A. Britto. The ripple protocol consensus algorithm, 2018.

[22] M. Vukolic. The origin of quorum systems. *Bulletin of the EATCS*, 101:125–147, 2010.

[23] J. H. WENSLEY, L. LAMPORT, J. GOLDBERG, , C. B. WEINSTOCK, M. W. GREEN, K. N. LEVITT, P. M. MELLIAR-SMITH, and R. E. SHOSTAK. Sift: design and analysis of a fault-tolerant computer for aircraft control". microelectronics reliability. *Microelectronics Reliability*, 66(10):1240–1255, 1979.

[24] Álvaro García-Pérez and M. A. Schett. Deconstructing stellar consensus. In *OPODIS*, 2019.