DOI:10.1145/3363823

A 50-year history of concurrency.

BY SERGIO RAJSBAUM AND MICHEL RAYNAL

Mastering Concurrent Computing through Sequential Thinking

I must appeal to the patience of the wondering readers, suffering as I am from the sequential nature of human communication.

—E.W. Dijkstra, 1968[12]

ONE OF THE most daunting challenges in information science and technology has always been mastering concurrency. Concurrent programming is enormously difficult because it copes with many possible nondeterministic behaviors of tasks being done at the same time. These come from different sources, including failures, operating systems, shared memory architectures, and asynchrony. Indeed, even today we do not have good tools to build efficient, scalable, and reliable concurrent systems.

Concurrency was once a specialized discipline for experts, but today the challenge is for the entire information technology community because of two disruptive phenomena: the development of networking communications, and the end of the ability to increase processors speed at an exponential rate. Increases in performance come through concurrency, as in multicore architectures. Concurrency is also critical to achieve fault-tolerant, distributed services, as in global databases, cloud computing, and blockchain applications.

Concurrent computing through sequential thinking. Right from the start in the 1960s, the main way of dealing with concurrency has been by reduction to sequential reasoning. Transforming problems in the concurrent domain into simpler problems in the sequential domain, yields benefits for specifying, implementing, and verifying concurrent programs. It is a two-sided strategy, together with a bridge connecting the two sides.

First, *a sequential specification* of an object (or service) that can be ac-

key insights

- **A main way of dealing with the enormous challenges of building concurrent systems is by reduction to sequential thinking. Over more than 50 years, more sophisticated techniques have been developed to build complex systems in this way.**
- **The strategy starts by designing sequential specifications, and then mechanisms to associate a concurrent execution to a sequential execution that itself can be tested against the sequential specification.**
- **The history starts with concrete, physical mutual exclusion techniques, and evolves up to today, with more abstract, scalable, and fault-tolerant ideas including distributed ledgers.**
- **We are at the limits of the approach, encountering performance limitations by the requirement to satisfy a sequential specification, and because not all concurrent problems have sequential specifications.**

cessed concurrently states the desired behavior only in executions where the processes access the object one after the other, sequentially. Thus, familiar paradigms from sequential computing can be used to specify shared objects, such as classical data structures (for example, queues, stacks, and lists), registers that can be `read` or modified, or database transactions. This makes it easy to understand the object being implemented, as opposed to a truly concurrent specification which would be hard or unnatural. Instead of trying to modify the well-understood notion of say, a queue, we stay with the usual sequential specification, and move the meaning of a concurrent implementation of a queue to another level of the system.

The second part of the strategy is to provide *implementation techniques* for efficient, scalable, and fault-tolerant concurrent objects. Locks enforce exclusive accesses to shared data, and concurrency control protocols. More abstract and fault-tolerant solutions that include *agreement* protocols that can be used for replicated data to locally execute object operations in the same order. Reliable *communication protocols* such as atomic broadcast and gossiping are used by the processes to communicate with each other. Distributed data structures, such as blockchains. Commit protocols to ensure *atomicity* properties. Several techniques are commonly useful, such as timestamps, quorums, group membership services, and failure detectors. Interesting liveness issues arise, specified by progress conditions to guarantee that operations are actually executed.

The bridge establishes a connection between the executions of a concurrent program and the sequential specification. It enforces *safety* properties by which concurrent executions appear as if the operations invoked on the object where executed instantaneously, in some sequential interleaving. This is captured by the notion of a consistency condition, which defines the way concurrent invocations to the operations of an object correspond to a sequential interleaving, which can then be tested against the its sequential specification.

A brief history and some examples. The history of concurrency is long and the body of research enormous; a few milestones are in the sidebar "A Few Dates from the History of Synchronization." The interested reader will find many more results about principles of concurrency in textbooks.[4,21,35,37] We concentrate here only on a few signifi-

A Few Dates from the History of Synchronization

1965	Mutual exclusion from atomic `read`/write registers	Dijkstra[11]
1965	Semaphores	Dijkstra[13]
1971	Mutual exclusion from non-atomic `read`/write registers	Lamport[23]
1974	Concurrent reading and writing	Lamport,[24] Peterson[33]
1977, 1983	Distributed state machine (DA 2000)	Lamport[25]
1980	Byzantine failures in synchronous systems (DA 2005)	Pease, Shostak, Lamport[31]
1981	Simplicity in mutex algorithms	Peterson[32]
1983	Asynchronous randomized consensus (DA 2015)	Ben-Or,[5] Rabin[34]
1985	Liveness (progress condition) (DA 2018)	Alpern, Schneider[1]
1985	Impossibility of asynchronous deterministic consensus in the presence of process crashes (DA 2001)	Fischer, Lynch, Paterson[16]
1987	Fast mutual exclusion	Lamport[27]
1991	Wait-free synchronization (DA 2003)	Herlihy[19]
1993, 1997	Transactional memory (DA 2012)	Herlihy, Moss,[20] Shavit, Touitou[40]
1995	Shared memory on top of asynchronization message-passing systems despite a minority of process crashes (DA 2011)	Attiya, Bar Noy, Dolev[2]
1996	Weakest information on failures to solve consensus in the presence of asynchrony and process crashes (DA 2010)	Chandra, Hadzilacos, Toueg[8]
2008	Scalability, accountability	Nakamoto[30]

A paper that received the Dijkstra ACM Award in the year X is marked (DA X).

cant examples of sequential reasoning used to master concurrency, providing a sample of fundamental notions of this approach, and we describe several algorithms, both shared memory and message passing, as a concrete illustration of the ideas.

We tell the story through an evolution that starts with mutual exclusion, followed by implementing `read`/write registers on top of message passing systems, then implementing arbitrary objects through powerful synchronization mechanisms. We discuss the modern distributed ledger trends of doing so in a highly scalable, tamper-proof way. We conclude with a discussion of the limitations of this approach: It may be that it is expensive to implement, and furthermore, there are inherently concurrent problems with no sequential specifications.

Mutual Exclusion

Concurrent computing began in 1961 with what was called *multiprogramming* in the Atlas computer, where concurrency was simulated—as we do when telling stories where things happen concurrently—interlacing the execution of sequential programs. Concurrency was born in order to make efficient use of a sequential computer, which can execute only one instruction at a time, giving users the illusion that their programs are all running simultaneously, through the operating system. A collection of early foundational articles on concurrent programming appears in Brinch.[6]

As soon as the programs being run concurrently began to interact with each other, it was realized how difficult it is to think concurrently. By the end of the 1960s a crisis was emerging: programming was done without any conceptual foundation and programs were riddled with subtle errors causing erratic behaviors. In 1965, Dijkstra discovered that *mutual exclusion* of parts of code is a fundamental concept of programming and opened the way for the first books of principles on concurrent programming, which appeared at the beginning of the 1970s.

Locks. A mutual exclusion algorithm consists of the code for two operations—`acquire()` and `release()`—that a process invokes to bracket a section of code called a *critical section*. The usual environment in which it is executed is asynchronous, where process speeds are arbitrary, independent from each other. A mutual exclusion algorithm guarantees two properties.

- *Mutual exclusion.* No two processes are simultaneously executing their critical section.
- *Deadlock-freedom.* If one or several processes invoke `acquire()` operations that are executed concurrently, eventually one of them terminates its invocation, and consequently executes its critical section.

Deadlock-freedom does not prevent specific timing scenarios from occurring in which some processes can never enter their critical section. The stronger starvation-freedom progress condition states that any process that invokes `acquire()` will terminate its invocation (and will consequently execute its critical section).

A mutual exclusion algorithm. The first mutual exclusion algorithms were difficult to understand and prove correct. We describe here an elegant algorithm by Peterson[32] based on `read`/write shared registers. Algorithms for a message-passing system have been described since Lamport's logical clock paper.[25]

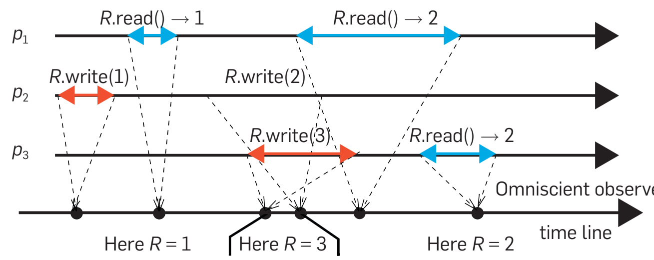
The version presented in Algorithm 1 is for two processes but can be easily generalized to n processes. The two processes $p1$ and $p2$ share three `read`/write atomic registers, $FLAG[1]$, $FLAG[2]$, and $LAST$. Initially $FLAG[1]$,

Algorithm 1. Peterson's mutual exclusion algorithm for two processes.

operation acquire() **is** % invoked by p_i, $i \in \{1, 2\}$
  $FLAG[i] \longleftarrow$ up; $LAST \longleftarrow i$; **let** $j = 3 - i$;
  wait (($FLAG[j]$= down) $\vee$ ($LAST \neq i$));
  return()
end operation.

operation release() **is** $FLAG[i] \longleftarrow$ down; return() **end operation** .

An Atomic (Linearizable) Execution of Processes p_1, p_2, and p_3 on xRead/Write Register R.



From an external observer point of view, it appears as if the operations were executed sequentially, at the sequence of linearization points of the `read`/write operations (indicated by the dotted arrows)

FLAG[2], are down, while *LAST* does not need to be initialized. Both processes can `read` all registers. Moreover, while *LAST* can be written by both processes, only p_i , $i \in \{1, 2\}$, writes to *FLAG*[i]. Atomic means the `read` and write operations on the registers seem to have been executed sequentially (hence, the notion of "last writer" associated with *LAST* is well defined).

When process p_i invokes `acquire()`, it first raises its flag, thereby indicating it is competing, and then writes its name in *LAST* indicating it is the last writer of this register. Next, process p_i repeatedly reads *FLAG*[j] and *LAST* until it sees *FLAG*[j] = `down` or it is no longer the last writer of *LAST*. When this occurs, p_i terminates its invocation. The operation `release()` consists in a simple lowering of the flag of the invoking process. The `read` and write operations on *FLAG*[1], *FLAG*[2], and *LAST* are totally ordered (atomicity), which facilitates the proof of the mutual exclusion and starvation-freedom properties.

Mutual exclusion was the first mechanism for mastering concurrent programming through sequential thinking, and lead to the identification of notions that began to give a scientific foundation to the approach, such as the concepts of progress condition and atomicity. It is the origin of the important area of concurrency control, by controlling access to data using locks (for example, 2-phase locking).

From Resources to Objects

At the beginning, a critical section was encapsulating the use of a physical resource, which by its own nature, is sequentially specified (for example, disk, printer, processor). Conceptually not very different, locks were then used to protect concurrent accesses of simple data (such as a file). However, when critical sections began to be used to encapsulate more general shared objects, new ideas were needed.

Data is not physical resources. A shared object is different from a physical object, in that it does not a priori require exclusive access; a process can `read` the data of a file while another process concurrently modifies it. The lock-free approach (introduced by Lamport[24]), makes possible to envisage implementations of purely digital objects without using mutual exclusion, in a way that operations can overlap in time.

Tolerating crash failures. Additionally, mutual exclusion cannot be used to implement an object in the presence of asynchrony and process crashes. If a process crashes inside its critical section, other processes, unable to tell if it crashed or is just slow, are prevented from accessing the object.

Consistency conditions. Wherever concurrent accesses to share data take place, a consistency condition is needed to define which concurrent operation executions are considered correct. Instead of transforming a concurrent execution into a sequential execution (as in mutual exclusion), the idea is to enforce that, from an external observer point of view, everything must appear as if the operations were executed sequentially. This is the *sequential consistency* notion (or serializability), which has been used since early 1976 in the database context to guarantee that transactions appear to have executed atomically.[38] However, sequential consistency is not composable. The stronger consistency condition of *linearizability* (or *atomicity*) requires that the total order of the operations respects the order on non-overlapping operations.[22] Linearizability is illustrated in the sidebar "An Atomic (Linearizable) Execution of Processes," that describes an execution in which three processes access an atomic `read`/write register R.

Read/Write Register on Top of Message-Passing Systems

Perhaps the most basic shared object is the `read`/write register. In the shared memory context, there are implementations from very simple registers (that only one process can write, and another can `read`), all the way to a multi-writer multireader (MWMR) register, that every process can write and every process can `read`; for example, see Herlihy.[21]

Distributed message-passing systems often support a shared memory abstraction and have a wide acceptance in both research and commercial computing, because this abstraction provides a more natural transition from uniprocessors and simplifies programming tasks. We describe here a classic fault-tolerant implementation of a `read`/write register on top of a message passing system.

It is relatively easy to build atomic `read`/write registers on top of a reliable asynchronous message-passing system, for example, Raynal,[36] but if processes may crash, more involved algorithms are needed. Two important results are presented by Attiya, Bar-Noy and Dolev:[2]

- An algorithm that implements an atomic `read`/write register on top of a system of n asynchronous message-passing processes, where at most $t < n/2$ of them may crash.
- A proof of the impossibility of

building an atomic read/write register when $t \geq n/2$.

This section presents the algorithm, referred to as the *ABD Algorithm*, which illustrates the importance of the ideas of reducing concurrent thinking to sequential reasoning. A more detailed proof as well as other algorithms can be found.[2,4,37]

Design principles of ABD. Each written value has an identity. Each process is both a client and a server. Let *REG* be the multi-writer multi-reader (MWMR) register that is built (hence, any process is allowed to read and write the register). On its client side a process pi can invoke the operations *REG*.write (v) to write a value v in *REG*, and *REG*.read() to obtain its current value. On its server side, a process p_i manages two local variables: reg_i which locally implements *REG*, and $timestamp_i$, which contains a timestamp made up of a sequence number (which can be considered as a date) and a process identity j. The timestamp $timestamp_i$ constitutes the "identity" of the value v saved in reg_i (namely, this value was written by this process at this time). Any two timestamps $\langle sn_i, i\rangle$ and $\langle sn_j, j\rangle$ are totally ordered by their lexicographical order; namely, $\langle sn_i, i\rangle < \langle sn_j, j\rangle$ means $(sn_i < sn_j) \vee (sn_i = sn_j \wedge i < j)$.

Design principles of ABD: intersecting quorums. A process p_i broadcasts a query to all the processes and waits for acknowledgments from a majority of them. Such a majority quorum set, has the following properties. As $t < n/2$, waiting for acknowledgments from a majority of processes can never block forever the invoking process. Moreover, the fact that any two quorums have a non-empty intersection implies the atomicity property of the read/write register *REG*.

The operation *REG*.write(v). This operation is implemented by Algorithm 2. When a process p_i invokes *REG*.write(v), it first creates a tag denoted (*tag*) which will identify the query/response messages generated by this write invocation. Then (phase 1), it executes a first instance of the query/response exchange pattern to learn the highest sequence number saved in the local variables $timestamp_j$ of a majority of processes p_j. When this is done, pi computes the timestamp *ts* which will be associated with the value v it wants to write in *REG*. Finally (phase 2), p_i starts a second query/response pattern in which it broadcasts the pair (v, *ts*) to all the processes. When it has received the associated acknowledgments from a quorum, p_i terminates the write operation.

On its server side, a process p_i that receives a WRITE_REQ message sent by a process p_j during phase 1 of a write operation, sends it back an acknowledgment carrying the sequence number associated with the latest value it saved in reg_i .When it receives WRITE_REQ message sent by a process p_j during phase 2 of a write operation, it updates its local data reg_i implementing *REG* if the received timestamp is more recent (with respect to the total order on timestamps) than the one saved in $timestamp_i$, and, in all cases, it sends back to p_j and acknowledgment (so p_j terminates its write).

It is easy to see that, due to the intersection property of quorums, the timestamp associated with a value v by the invoking process p_i is greater than the ones of the write operations that terminated before p_i issued its own write operation. Moreover, while concurrent write operations can associate the same sequence number with their values, these values have different (and ordered) timestamps.

The operation REG.read(). Algorithm 3 implements operation *REG*.read(), with a similar structure as the implementation of operation *REG*.write().

Notice that the following scenario can occur, which involves two read operations read1 and read2 on a register *REG* by the processes *p1* and *p2*, respectively, and a concurrent write operation *REG*.write(v) issued by a process *p3*. Let *ts*(v) be the timestamp associated with v by *p3*. It is possible that the phase 1 majority quorum obtained by *p1* includes the pair (v, *ts*(v)), while the one obtained by *p2* does not. If this occurs, the first read operation read1 obtains a value more recent that the one obtained by the second read2, which violates atomicity. This can be easily solved by directing each read operation to write the value it is about to return as a result. In this way, when read1 terminates and returns v, this value is known by a majority of processes despite asynchrony, concurrency, and a minority of process crashes. This phenomenon (called *new/old inversion*) is prevented by the phase 2 of a read operation (as illustrated in the accompanying figure).

We have seen how the combination of intersecting quorums and

Algorithm 2. ABD's implementation of read/write register: write operation.

operation *REG*.write (v) **issued by process** p_i **is**
  build a new tag *tag* identifying this write operation;
  % Phase 1: acquire information on the system state %
  broadcast WRITE_REQ (*tag*);
  wait acknowledgments from a majority of processes,
  each carrying *tag* and a sequence number;
  % Phase 2: update system state %
  $Ts \leftarrow \langle msn + 1, i\rangle$ where *msn* is
  the greatest sequence number previously received;
  broadcast WRITE (*tag*, v, *ts*);
  wait acknowledgments carrying *tag* from a majority of proc.;
  return().

when WRITE_REQ (*tag*) **is received from** p_j, $j \in \{1, \ldots, n\}$ **do**
  send to p_j an acknowledgment carrying *tag*, and
  the sequence number contained in $timestamp_i$.

when WRITE (*tag*) **is received from** p_j, $j \in \{1, \ldots, n\}$ **do**
  if ($timestamp_i < ts$) **then**
    $timestamp_i \leftarrow ts$; $reg_i \leftarrow v$ **end if**;
  send to p_j an acknowledgment carrying *tag*.

timestamps, two ideas useful in other situations, facilitate the implementation of atomic read/write registers in asynchronous message-passing systems where a minority of process may crash. And how sequential thinking for shared registers can be used at the upper abstraction level.

The World of Concurrent Objects

A read/write register is a special case of an object. In general, an *object* is defined by the set of operations that processes can invoke, and by the behavior of the object when these operations are invoked sequentially. These can be represented by an automaton or by a set of sequential traces. In the case of an automaton, for each state, and each possible operation invocation, a transition specifies a response to that invocation, and a new state (the transition is often a deterministic function, but not always). Thus, usual data structures from sequential programing, such as queues and stacks, can be used to define concurrent objects.

Consensus. At the core of many situations where sequential reasoning for concurrent programming is used (including state machine replication) are agreement problems. A common underlying abstraction is the consensus object. Let *CONS* be a consensus object. A process p_i can invoke the operation *CONS*.`propose(`v`)` once. The invocation eventually returns a value v'. This sequential specification for *CONS* is defined by the following properties.

- *Validity.* If an invocation returns v then there is a *CONS*.`propose(`v`)`.
- *Agreement.* No two different values are returned.
- *Termination.* If a process invokes *CONS*.`propose(`v`)` and does not crash, the operation returns a value.

All objects are not equal in an asynchronous, crash-prone environment. Consensus objects are the strongest, in the sense that (together with `read`/write registers), they can be used to implement, despite asynchrony and process crashes, any object defined by a sequential specification. Other important objects, such as a queue or a stack are of intermediate strength: they cannot be implemented by asynchronous processes, which communicate using `read`/write registers only. Such implementations, that require that any operation invoked by a process that does not crash must return (independently of the speed or crashes of other processes), are said to be *wait-free*.

One way of measuring the synchronization power of an object in the presence of asynchrony and process crashes is by its consensus number. The consensus number of an object O is the greatest integer n, such that it is possible to wait-free implement a consensus object for n processes from any number of objects O and atomic `read`/write registers. The consensus number of O is ∞ if there is no such greatest integer. As an example, the consensus number of a Test&Set object or a stack object is 2, while the consensus number of a Compare&Swap or Load/Link&Store/Conditional (LL/SC) object is ∞. We will discuss a LL/SC object later. These ideas where first discussed by Herlihy.[19]

State Machine Replication

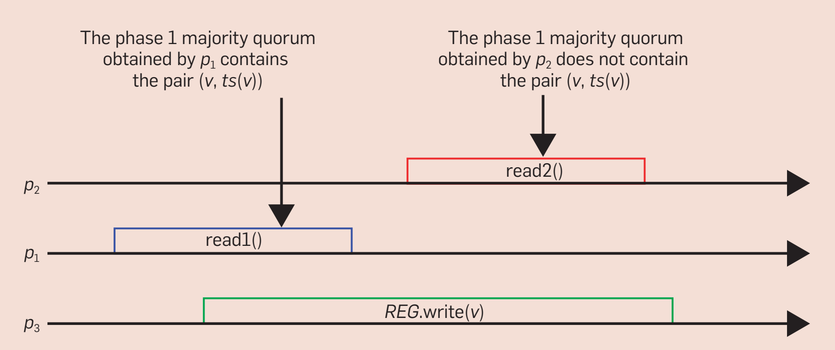
A concurrent stack can be implemented by executing the operations `pop()` and `push()` using mutual exclusion. However, as already indicated, this strategy does not work if processes may crash. The *state machine replication mechanism*[25,39] is a general way of implementing an object by asynchronous processes communicating by message-passing. We will discuss implementations where the processes may fail by crashing; there are also implementations that tolerate arbitrary (Byzantine) failures.[7] We should point out that nondeterministic automata sometimes appear in applications and pose additional challenges for implementations.

The general idea is for the processes to agree on a sequential order of the concurrent invocations, and then each one to simulate the sequential specification automaton locally. We illustrate here the approach with a *total order*

Algorithm 3. ABD's implementation of read/write register: read operation.

operation *REG*.read () **is**
  build a new tag tag identifying this read operation;
  % Phase 1: acquire information on the system state %
  broadcast READ_REQ (tag);
  wait acknowledgments from a majority of processes,
  each carrying tag and a pair ⟨value,timestamp⟩;
  let ts be the greatest timestamp received,
  and v the value associated with this timestamp;
  % Phase 2: update system state %
  broadcast WRITE (tag, v, ts);
  wait ACK_WRITE (tag) from a majority of proc.;
  return (v).
when READ_REQ (tag) **is received from** $p_j, j \in \{1, \ldots, n\}$ **do**
  send to p_j an ack. carrying tag, reg_i and timestamp$_i$.

New/old inversion scenario.



Algorithm 4. TO-broadcast-based universal construction.

when operation $\mathsf{op}_x(param_x)$ **is invoked by** process p_i **do**
  $result_i \leftarrow \perp$; **let** $sent\_msg = \langle \mathsf{op}_x(param_x), i \rangle$;
  TO_broadcast ($sent\_msg$);
  wait ($result_i \neq \perp$); return ($result_i$).

background task T **is**
  repeat forever
    $rec\_msg \leftarrow$ TO_deliver();
    $\langle state_i, res \rangle \leftarrow \delta(state_i, rec\_msg.op)$;
    if ($rec\_msg.proc = i$) **then** $result_i \leftarrow res$ **end if**
  end repeat.

Algorithm 5. Implementing TO-broadcast from consensus.

when p_i invokes TO_broadcast(m) **do** send(m) to itself.

when m **is received for the first time do**
  broadcast(m); $pending_i \leftarrow pending_i \cup \{m\}$.

when ($to\_deliverable_i$ contains messages not yet to-delivered) **do**
  let m = first message $\in to\_deliverable_i$ not yet to-delivered;
  TO_deliver(m).

background task T **is**
  repeat forever
    wait($pending_i \setminus to\_deliverable_i \neq \emptyset$);
    let $seq = (pending_i \setminus to\_deliverable_i)$;
    order the messages in seq;
    $sn_i \leftarrow sn_i + 1$; $res_i \leftarrow CS[sn_i]$.propose($seq$);
    add res_i at the end of $to\_deliverable_i$
  end repeat.

Circumventing Consensus Impossibility

Three ways of circumventing the consensus impossibility:

- The failure detector approach[8] can abstract away synchrony assumptions sufficient to distinguish between slow processes and dead processes.
- In eventually synchronous systems[14] there is a time after which the processes run synchronously. The celebrated Paxos algorithm is an example.[28]
- By using random coins[5] consensus is solvable with high probability.
- Often not all combinations of input values occur.[29]

broadcast mechanism for reaching the required agreement.

Total order broadcast. The TO-broadcast abstraction is an important primitive in distributed computing, which ensures that all correct processes receive messages in the same order.[18,37] It is used through two operations, `TO_broadcast()` and `TO_deliver()`. A process invokes `TO_broadcast(`m`)`, to send a message m to all other processes. As a result, processes execute `TO_deliver()` when they receive a (totally ordered) message.

TO-broadcast illustrates one more general idea within the theory of mastering concurrent programming through sequential thinking: the identification of communication abstractions that facilitate building concurrent objects defined by a sequential specification.

State machine replication based on TO-broadcast. A concurrent implementation of object O is described in Algorithm 4. It is a universal construction, as it works for any object O defined by a sequential specification. The object has operations $\mathtt{op}_x()$, and a transition function δ() (assuming δ is deterministic),where $\delta(state, \mathtt{op}_x(param_x))$ returns the pair $\langle state', res\rangle$, where $state'$ is the new state of the object and res the result of the operation.

The idea of the construction is simple. Each process p_i has a copy $state_i$ of the object, and the TO-broadcast abstraction is used to ensure that all the processes p_i apply the same sequence of operations to their local representation $state_i$ of the object O.

Implementing TO-broadcast from consensus. Algorithm 5 is a simple construction of TO-broadcast on top of an asynchronous system where consensus objects are assumed to be available.[18]

Let `broadcast(`m`)` stand for "**for each** $j \in \{1, \ldots, n\}$ *do* `send(`m`)` to p_j **end for**." If the invoking process does not crash during its invocation, all processes receive m; if it crashes an arbitrary subset of processes receive m.

The core of the algorithm is the background task T. A consensus object $CS[k]$ is associated with the iteration number k. A process p_i waits until there are messages in the set $pending_i$ and not yet in the queue $to\_deliverable_i$. When this occurs, process pi computes this set of messages (seq)

and order them. Then it proposes seq to the consensus instance $SC[k]$. This instance returns a sequence saved in res_i, which is added by p_i at the end of its local queue $to\_deliverable_i$.

When are Universal Constructions Possible?

An impossibility. A fundamental result in distributed computing is the impossibility to design a (deterministic) algorithm that solves consensus in the presence of asynchrony, even if only one process may crash, either in message-passing or read/write shared memory systems.[16] Given that consensus and TO-broadcast are equivalent, the state machine replication algorithm presented above cannot be implemented in asynchronous systems where processes can crash.

Thus, sequential thinking for concurrent computing has studied properties about the underlying system that enable the approach to go through. There are several ways of considering computationally stronger (read/write or message-passing) models,[35,37] where state machine replication can be implemented. Some ways, mainly suited to message-passing systems, are presented in the sidebar "Circumventing Consensus Impossibility." Here, we discuss a different way, through powerful communication hardware.

Systems that include powerful objects. Shared memory systems usually include synchronization operations such as Test&Set, Compare&Swap, or the pair of operations Load Link and Store Conditional (LL/SC), in addition to read/write operations. These operations have a consensus number greater than 1. More specifically, the consensus number of Test&Set is 2, while the consensus number of both Compare&Swap and the pair LL/SC, is $+\infty$. Namely, 2-process (but not a 3-process) consensus can be implemented from Test&Set, despite crash failures. Compare&Swap (or LL/SC) can implement consensus for any number of processes. Hence, for any n, any object can be implemented in an asynchronous n-process read/write system enriched with Compare&Swap (or LL/SC), despite up to n–1 process crashes. Furthermore, there are implementations that tolerate arbitrary, malicious (Byzantine) failures.[7,37]

State machine replication based on LL/SC. To give more intuition about state machine replication, and furthermore, about the way that blockchains work, we present an implementation based on LL/SC. (Another option is based on Compare&Swap, but it is not "self-contained" in the sense it has to deal with the ABA problem.[35])

The intuition of how the LL/SC operations work is as follows. Consider a memory location M, initialized to $\perp$, accessed only by the operations LL/SC. Assume that if a process invokes M.SC(v)

Algorithm 6. Implementing a consensus object CONS from the operations LL/SC.

```
Initially: M = ⊥
operation CONS.propose (v) is
    val_i ← M.LL();
    if (val_i ≠ ⊥) then return(val_i)
              else b_i ← M.SC (v);
                   if b_i then return(v)
                        else val_i ← M.LL(); return(val_i)
              end if
    end if.
```

Universal Construction based on LL/SC

```
when the operation op_x (param_x) is locally invoked do
    sn_i ← sn_i + 1; BOARD[i] ← ⟨op_x (param_x), sn_i⟩;
    apply();
    state_i ← STATE.LL(); return(state_i.res[i]).

internal procedure apply() is
    state_i ← STATE.LL();
    board_i ← [BOARD[1], BOARD[2], . . . , BOARD[n]];
,   for ℓ ∈ {1, . . . , n} do
        if (board_i [ℓ]. sn = state_i . sn_i[ℓ] + 1)
          then ⟨state_i .value, state_i.res[ℓ]⟩ ←
                δ (state_i .value, pairs_i [ℓ].op); % line A
                state_i.sn[ℓ] ← state_i .sn[ℓ] + 1 % line B
        end if
    end for;
    success ← STATE.SC(state_i);
    if (¬success) then
        state_i i ← STATE.LL();
        if (sn_i = state_i .sn [i] + 1)
          then same as lines A and B with ℓ = i;
                STATE.SC(state_i )
        end if
    end if
```

it has previously invoked M.LL(). The operation M.LL() is a simple read of M which returns the current value of M.LL(). When a process pi invokes M.SC(v) the value v is written into M if and only if no other process invoked M.SC() since its (p_i) last invocation of M.LL(). If the write succeeds M.SC() returns `true`, otherwise it returns `false`.

Algorithm 6 is a simple implementation of consensus from the pair of operations LL/SC, which tolerates any number of process crashes.

In the sidebar "Universal Construction Based on LL/SC," there is a shared-memory, LL/SC based universal construction.[15] Looking at the algorithm, one begins to get a feeling for the distributed ledgers discussed next.

While resources are physical objects, data is digital objects.

Distributed Ledgers

Since ancient times, ledgers have been at the heart of commerce, to represent concurrent transactions by a permanent list of individual records sequentialized by date. Today we are beginning to see algorithms that enable the collaborative creation of digital distributed ledgers with properties and capabilities that go far beyond traditional physical ledgers. All participants within a network can have their own copy of the ledger. Any of them can append a record to the ledger, which is then reflected in all copies in minutes or even seconds. The records stored in the ledger can stay tamper-proof, using cryptographic techniques.

Ledgers as universal constructions. Mostly known because of their use in cryptocurrencies, and due to its *blockchain* implementation,[30] from the perspective of this paper a *distributed ledger* is a byzantine fault-tolerant replicated implementation of a specific *ledger* object. The ledger object has two operations, `read()` and `append()`. Its sequential specification is defined by a list of blocks. A block X can be added at the end of the list with the operation `append(`X`)`, while a `read()` returns the whole list. In the case of a cryptocurrency, X may contain a set of transactions.

Thus, a ledger object, as any other object, can be implemented using a Byzantine failures-tolerant state machine replication algorithm. Conversely, a ledger can be used as a universal construction of an object O defined by a state machine with a transition function δ. To do so, when a process invokes `append(`X`)`, X consists of a transition to be applied to the state machine. The state of the object is obtained through a `read()` invocation, which returns the sequence of operations which have been sequentially appended to the ledger, and then locally applying them starting from the initial state of the object (see Raynal[37] for more details).

Three remarkable properties. The apparently innocent idea of a `read()` operation that returns the list of commands that have been applied to the state machine, opens the discussion of one of the remarkable points of distributed ledgers that has brought them to such wide attention. The possibility of guaranteeing a *tamperproof* list of commands. The blockchain implementation is by using cryptographic hashes that link each record to the previous one (although the idea has been known in the cryptography community for years).

The ledger implementation used in Bitcoin showed it is possible to have a state machine replication tolerating Byzantine failures that scales to hundreds of thousands of processes. The cost is temporarily sacrificing consistency—forks can happen at the end of the blockchain, which implies that the last few records in the blockchain may have to be withdrawn.

The third remarkable property brought to the public attention by distributed ledgers is the issue of who the participants can be. As opposed to classic algorithms for mastering concurrency through sequential thinking, the participants do not have to be a priori-known, can vary with time, and may even be anonymous. Anyone can append a block and `read` the blockchain (although there are also *permissioned* versions where participants have to be registered, and even hybrid models). In a sense, a distributed ledger is an open distributed database, with no central authority, where the data itself is distributed among the participants.

Agreement in dynamic, Byzantine systems. Bitcoin's distributed ledger implementation is relatively simple to explain in the framework of state machine replication. Conceptually it builds on randomized consensus (something that had already been carefully studied in traditional approaches, as noted in the sidebar "Circumventing Consensus

Impossibility"), through the following ingenious technique to implement it. Whenever several processes (not necessarily known a priori, hence the name of "dynamic system") want to concurrently append a block, they participate in a lottery. Each process selects a random number (by solving cryptographic puzzles) between 0 and some large integer K, and the one that gets a number smaller than $k << K$, wins, and has the right to append its desired block.

The implementation details of the lottery (by a procedure called *proof of work*) are not important for this article; what is important here is that with high probability only one wins (and selected at random). However, from time to time, more than one process wins, and a *fork* happens, with more than one block being appended at the end of the blockchain. Only one branch eventually pervades (in Bitcoin this is achieved by always appending to the longest branch). This introduces a new interesting idea into the paradigm of mastering concurrency through sequential thinking: a trade-off between faster state machine replication, and temporary loss of consistency. In other words, the x operations at the very end of the blockchain, for some constant x (which depends on the assumptions about the environment) cannot yet be considered committed.

The Limits of the Approach

It is intuitively clear, and it has been formally proved, that linearizability or even serializability may be costly. Recent papers in the context of shared memory programming, argue that it is often possible to improve performance of concurrent data structures by relaxing their semantics.[9] In the context of distributed systems, eventual consistency is widely deployed to achieve high availability by guaranteeing that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value (despite its name is not technically a consistency condition.[3]). In the case of distributed ledgers, we have seen the benefit that can be gained by relaxing the sequential approach to mastering concurrency: branches at the end of the blockchain (such as Bitcoin) temporarily violate a consistent view of the ledger. Still, blockchains suffer from a performance bottleneck due to the requirement of ordering all transactions in a single list, which has prompted the exploration of partially ordered ledgers, based on directed acyclic graphs such as those of Tangle or Hedera Hashgraph.

The CAP Theorem formalizes a fundamental limitation of the approach of mastering concurrency through sequential reasoning—at most, two of the following three properties are achievable: consistency, availability, partition tolerance.[17] This may give an intuition of why distributed ledgers implementations have temporary forks. An alternative is a cost in availability and postpone the property that every non-failing participant returns a response for all operations in a reasonable amount of time. We have already seen in the ABD algorithm that the system continues to function and upholds its consistency guarantees, provided that only a minority of processes may fail.

Finally, another fundamental limitation to the approach of mastering concurrency through sequential reasoning is that not all concurrent problems of interest have sequential specifications. Many examples are discussed in Castañeda et al.,[10] where a generalization of linearizability to arbitrary concurrent specifications is described.

Acknowledgment. The authors acknowledge support of UNAM-PAPIT IN106520, INRIA-Mexico Associate Team, CNRS-Mexico UMI.

References

1. Alpern, B. and Schneider, F.B. Defining liveness. *Information Processing Letters 21*, 4 (1985), 181–18.
2. Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message-passing systems. *JACM 42*, 1 (1995), 121–132.
3. Attiya, H., Ellen, F. and Morrison, A., Limitations of highly-available eventually-consistent data stores. *IEEE Trans. Parallel Distributed Systems 28*, 1 (2017), 141–155.
4. Attiya, H. and Welch, J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, (2nd Edition), Wiley, 2004.
5. Ben-Or, M. Another advantage of free choice: completely asynchronous agreement protocols. In *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, (1983), 27–30.
6. Brinch, H.P. (Ed.). *The Origin of Concurrent Programming.* Springer, (2002).
7. Cachin, C. State machine replication with Byzantine faults. *Replication.* Springer LNCS 5959, (2011), 169–184.
8. Chandra, T.D., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. *JACM 43*, 4 (1996), 685–722.
9. Calciu, I., Sen, S., Balakrishnan, M., and Aguilera, M. How to implement any concurrent data structure for modern servers. *ACM Operating Systems Rev. 51*, 1 (2017), 24–32.
10. Castañeda, A., Rajsbaum, S., and Raynal, M. Unifying concurrent objects and distributed tasks: Interval-linearizability. *JACM 65*, 6 (2018), Article 45.
11. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM 8*, 8 (Sept. 1965), 569.
12. Dijkstra, E.W. Cooperating sequential processes. *Programming Languages.* Academic Press, 1968, 43–112.
13. Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Informatica 1*, 1 (1971), 115–138.
14. Dolev, D., Dwork, C., and Stockmeyer, L. On the minimal synchronism needed for distributed consensus. *JACM 34*, 1 (1987), 77–97.
15. Fatourou, P. and Kallimanis, N.D. Highly-efficient wait-free synchronization. *Theory of Computing Systems 55* (2014), 475–520.
16. Fischer, M.J., Lynch, N.A., and Paterson, M.S. Impossibility of distributed consensus with one faulty process. *JACM 32*, 2 (1985), 374–382.
17. Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.
18. Hadzilacos, V. and Toueg, S. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425. Cornell Univ. (1994)
19. Herlihy, M.P. Wait-free synchronization. *ACM Trans. on Prog. Languages and Systems 13*, 1 (1991), 124–149.
20. Herlihy, M.P. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th ACM Int'l Symp. Computer Architecture.* ACM Press, 1993, 289–300.
21. Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann, ISBN 978-0-12-370591-4 (2008).
22. Herlihy, M.P. and Wing, J.M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Programming Languages and Systems 12*, 2 (1990), 463–492.
23. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (1974), 453–455.
24. Lamport, L. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov. 1977), 806–811.
25. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (Sept. 1978), 558–565.
26. Lamport, L. On interprocess communication, Part I: Basic formalism. *Distributed Computing 1*, 2 (1986), 77–85.
27. Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Computer Systems 5*, 1 (1987), 1–11.
28. Lamport, L. The part-time parliament. *ACM Trans. Computer Systems 16*, 2 (1998), 133–169.
29. Mostéfaoui, Rajsbaum, S. and Raynal, M. Conditions on input vectors for concensus solvability in asynchronous distributed systems. *JACM 50*, 6 (2003, 922–954.
30. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Unpublished manuscript* (2008).
31. Pease, M., Shostak, R. and Lamport, L. Reaching agreement in the presence of faults. *JACM 27* (1980), 228–234.
32. Peterson, G.L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (1981), 115–116.
33. Peterson, G.L. Concurrent reading while writing. *ACM Trans. on Prog. Languages and Systems* (1983), 5:46-5:55.
34. Rabin, M. Randomized Byzantine generals. *Proc. 24th IEEE Symp. Foundations of Computer Science.* IEEE Computer Society Press, 1983, 116–124.
35. Raynal, M. *Concurrent Programming: Algorithms, Principles and Foundations.* Springer, 2013, ISBN 978-3-642-32026-2.
36. Raynal, M. *Distributed Algorithms for Message-Passing Systems.* Springer, 2013, ISBN 978-3-642-38122-5.
37. Raynal, M. *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach.* Springer, 2018, ISBN 978-3-319-94140-0.
38. Stearns, R.C., Lewis, P.M. and Rosenkrantz, D.J. Concurrency control for database systems. In *Proc. 16th Conf. Found. Comp. Sci.*, (1976), 19–32.
39. Schneider, F.B. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys 22*, 4 (1990), 299–319.
40. Shavit, N. and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

Sergio Rajsbaum (rajsbaum@im.unam.mx) is a professor at the Instituto de Matematicas at the Universidad Nacional Autónoma de Mexico inn Mexico City, México.

Michel Raynal (raynal@irisa.fr) is a professor at IRISA, University of Rennes, France, and Polytechnic University in Hong Kong.