Distributed Universal Constructions
a guided tour

Michel RAYNAL
Institut Universitaire de France
Academia Europaea
IRISA, Université de Rennes, France
Polytechnic University (PolyU), Hong Kong

Content

• Short historical perspective and a point of view
• From sequential computing to distributed computing
• Distributed universal constructions
• Conclusion

Never forget ....

"No, you weren't downloaded. Your were born."
From the very beginning (?) mankind is looking for UNIVERSALITY!

One upon a time...

Plimpton tablet 322
(1800 BC)
15 lines
Pythagorean triplets
($a^2 + b^2 = c^2$)
Base sexagesimal base

Algorithms seem to be born with writing...
(only receipts at this time, no formalization, no proofs)

A little bit later...

A great step ahead!
Axioms: Euclid ($\sim$ 300 BC)

“Ruler + compass” constructions
“Ruler + compass” define the set of allowed operations
We have: algorithms + proofs

a few historical references

- Neugebauer O. E., *The exact sciences in Antiquity*

- Kramer S. N., *History begins at Sumer: thirty-nine firsts in man’s recorded history*
  University of Pennsylvania Press, 416 pages (1956)

- Donald Knuth
  *Ancient Babylonian Algorithms*
Example: Bisecting an angle with compass + ruler

- **Proof:** consists in showing that the triangles ABD and ACD are equal

---

**BTW: what about trisecting an angle?**

- Is it possible to **trisect an angle with a compass + ruler?**
- One of the hardest pb for Ancient Greeks (squaring the circle)
- Plus the fact that \( \pi \) is a transcendent number (F. von Lindemann 1882)
- Hence **ruler+ compass operations are not universal** for geometric constructions!

---

**Still a little bit later...**

**M. Ibn Musa Al Khawarizmi**

780, Khiva - 850, Bagdad

Contributed to algebra ... but gave its name to algorithms!

---

**A few references**

- *Kitabu al-mukhtasar fi hisabi al-jabr wa’l-muqabala*
- *Kitabu al-jami’ wa ’t-tafriq bi-hisabi ’l-Hind* (the book of addition and subtraction from Indian calculus)
The science of operations
Loking for universality!

- Founding result:
  - FSA $\subset$ Pushdown Automata $\subset$ Turing Machines
  - Machines to process SYMBOLS

- Church-Turing Thesis: universal machines
- Universality of data representation: séquences de bits (books, images, files, etc.)

About informatics (1)

• Main resources:
  ✷ up to mid of XX-th century: matter/energy
  ✷ from mid of XX-th century: information
  ✷ as matter/energy: information can be collected, consumed, transformed, stored, carried, etc.
  ✷ differently from matter/energy: it does not burn, it can be copied at “zero cost”

• Looking for universality (just repeating...)

About informatics (2)

• Produces a “new” way of thinking (algorithmics-based)
• From putting the world into equations to putting the world into algorithms
• Informatics is the language of science!

PART 2

From sequential computing to distributed computing

The basic unit of sequential computing

\[ \text{in} \xrightarrow{f()} \text{out} = f(\text{in}) \]

• The notion of a function
• Sequential algorithm
• The notion of computability (Turing machine)
• The notion of impossibility (e.g., halting problem)
• The fundamental hierarchy
  \[ \text{FSA} \subset \text{Pushdown Automata} \subset \text{Turing Machines} \]
• Church-Turing’s Thesis
The case of parallel computing

- We look inside the box implementing \( f() \)
  - mono-processor
  - multiprocessor: to be more efficient
- The problem could be solved by a sequential algorithm, but can be solved more efficiently with several computing entities
- Parallel computing is an “extension” of sequential computing looking for efficiency
- This has a long story and introduced new techniques and concepts (e.g., task graphs, scheduling, etc.)

What is distributed computing?

DC arises when one has to solve a problem in terms of entities (processes, agents, sensors, peers, actors, nodes, processors, ...) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved

DC is about Mastering UNCERTAINTY

The basic unit of distributed computing

- The notion of a (distributed) task
  - A task \( T \) is a triple \((\mathcal{I}, \mathcal{O}, \Delta)\)
    - \( \mathcal{I} \): set of input vectors (of size \( n \))
    - \( \mathcal{O} \): set of output vectors (of size \( n \))
    - \( \Delta \): relation from \( \mathcal{I} \) into \( \mathcal{O} \): \( \forall I \in \mathcal{I}: \Delta(I) \subseteq \mathcal{O} \)
  - \( I[i] \): private input of \( p_i \)
  - \( O[i] \): private output of \( p_i \)
  - \( \forall I \in \mathcal{I} \):
    - \( \Delta(I) \) defines the set of legal output vectors that can be decided from the input vector \( I \)
  - The notion of a task: from an input vector to an output
  - The inputs are DISTRIBUTED (this is not under the control of the algorithm designer)
  - Failures belong to the model (in nearly all cases)

T() is a relation
Examples of tasks

- Binary consensus
  - $I = \{\text{all vectors of 0 and 1}\}$
  - $O = \{\{0,\ldots,0\}, \{1,\ldots,1\}\}$
  - Let $X_0 = \{0,\ldots,0\}$ and $X_0 = \{1,\ldots,1\}$
    - $\Delta(X_0) = \{0,\ldots,0\}$ and $\Delta(X_1) = \{1,\ldots,1\}$
    - $\Delta(\text{any vector except } X_0, X_1) = O$

- $k$-set agreement, Renaming, Weak symmetry breaking
- $k$-Simultaneous consensus, etc.

Solving a task

A distributed algorithm $A$ is a set of $n$ local automata (Turing machines) that cooperate through specific communication objects (e.g., message-passing network, shared memory, etc.)

An algorithm $A$ solves a task $T$ if in any run

- $\exists I \in \mathcal{I}$ such that each $p_i$ starts with (proposes) $in_i = I[i]$
- $\exists O \in \Delta(I)$ such that $O[j] = out_j$ for each process $p_j$ that that computes (decides) an output $out_j$

Distributed computing: birth certificates

  - Partial order on events
  - Scalar clocks
  - State machine replication
  - Impossibility result in asynch. crash-prone systems
  - Notion of valence (captures non-determinism)

A famous quote ... and its formalization

- “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable” (L. Lamport)

Reminder: DC is about Mastering UNCERTAINTY!
To summarize

- **Real-time**: masters On-time computing
- **Parallelism**: provides Efficiency
- **Distributed computing**: masters Uncertainty

(We are -more or less- implicitly using a lot of heuristics!)

Fundamental issue: cope with the non-determinism created by the environment (asynchrony, failures)

---

**PART 3**

Universal constructions in crash-prone shared memory systems

---

**Content**

- Concurrent objects, failures, asynchrony, progress
- What is a universal construction?
- Basic asynchronous read/write model
- Warm-up: a simple LL/SC-based universal construction
- Extensions: disjoint parallelism, abortable objects
- From memory operations to agreement objects
- Consensus object and consensus hierarchy
- Universal construction “1 among $k$” and “$\ell$ among $k$”

---

**Companion paper**

Distributed Universal Constructions: a Guided Tour by Michel Raynal
"In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine.

In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants."


### Computation model (base wait-free model)

- **Process and failure model:**
  - A set of $n$ asynchronous processes $p_1, \ldots, p_n$
  - "Asynchronous" means each process proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes.
  - Up to $t < n - 1$ processes may crash
  - A process that crashes: faulty, otherwise: non-faulty

- **Communication model:**
  - The processes communicate with atomic read/write registers (memory locations)
  - "Atomicity" (or Linearizability) means that the read and write primitive operations on a register appear as if they have been executed one after the other

- **Notation:** $\text{CARW}_n[\emptyset]$

### Linearizability (atomicity) and non-determinism

![Diagram showing linearizability (atomicity) and non-determinism]

Possibly different linearizations, but all respect physical order on operations

- **Message-passing model:**
  - complete point-to-point network
  - no bound on transfer delays (but finite)
  - reliable (no loss, creation, duplication, alteration)

- In the presence of up to $t$ failures:
  - Crash: the read/write model can be simulated on top the message-passing model only iff $t < n/2$
  - Byzantine: the read/write model can be simulated on top the message-passing model only iff $t < n/3$
Concurrent objects

- **Concurrent object**: object that can be accessed (possibly simultaneously) by several processes
- Here: defined by
  - a sequential specification
  - on total operations
- Remark: not all objects have a seq. specification
- Fundamental problem of shared memory distributed programming: 
  implement high level concurrent objects, where “high level” means that the object provides the processes with an abstraction level higher than the atomic hardware-provided instructions

On Progress conditions

- Failure-free model
  - * Deadlock-freedom
  - * Starvation-freedom
- Wait-free model
  - * Locks (mutex) cannot be used!
  - * three progress conditions
    - * Wait-freedom
    - * Non-blocking
    - * Obstruction-freedom

Wait-freedom

- Any operation (on the object that is built) issued by a process that does not crash terminates (whatever the behavior of the other processes)
- The strongest progress condition


Non-blocking aka Lock-freedom

- At least one process can always progress (all its object operations terminate)
- Generalized: $k$-lock-freedom which states that at least $k$ processes can always make progress
- $n$-lock-freedom = wait-freedom

Obstruction-freedom

- A process that does not crash terminates its operation if all the other processes hold still long enough
- \(k\)-obstruction-freedom states that, if a set of at most \(k\) processes run alone for a sufficiently long period of time, they will terminate their operations
- Differently from wait-freedom and non-blocking, the definition of obstruction-freedom depends on concurrency pattern


Universal construction

- Let PC be a progress condition
- A PC-compliant universal construction is an algorithm that, given the sequential specification of an object \(O\) (or a sequential implementation of it), provides a concurrent implementation of \(O\) satisfying \(PC\) in the presence of up \((n-1)\) process crashes

What can be done in pure read/write systems

- Let us consider \(\text{CARW}_n[\emptyset]\)

- OB-compliant universal construction: easy
- WF-compliant universal construction: impossible


- to implement WF-compliant universal constructions \(\text{CARW}_n[\emptyset]\) must be enriched with hardware operations providing (strong enough) additional computational power
- in the following: WF-compliant universal constructions

Enriching the basic read-write model with LL/SC

- Notation \(\text{CARW}[\text{LL/SC}]\)
- The atomic operations \text{LL} and \text{SC}

- Let \(X\) a memory location and \(p_i\) the invoking process

  - \(X.\text{LL}()\) returns the current value of \(X\)
  - \(X.\text{SC}(v)\) is a conditional write, returns a Boolean
    - let \(p_i\) be the process that issues \(X.\text{SC}(v)\). This writes succeeds (the value \(v\) is written into \(X\) and true is returned) iff \(X\) has not been written by an other process since the last reading of \(X\) by \(p_i\) (\(X.\text{LL}()\))
  - Weak variants exist on some architectures such as Alpha AXP (\text{ldl}/\text{stl}_c), IBM PowerPC (\text{lwarx}/\text{stwcx})
The pair Load Linked/Store Conditional

An algorithmic definition

Assume a boolean array \( \text{valid}_X[1..n] \) init to \([false, \ldots, false]\)

\[
\text{operation } X.\text{LL()} \text{ issued by } p_i \text{ is } \\
\text{valid}_X[i] \leftarrow true; \text{return}(X).
\]

\[
\text{operation } X.\text{SC}(v) \text{ issued by } p_i \text{ is } \\
\text{if } \neg\text{valid}_X[i] \text{ then return}(false) \\
\text{else } X \leftarrow v; \\
\quad \forall j: \text{valid}_X[j] \leftarrow false; \\
\text{return}(true)
\]

end if.

LL/SC in action

Notion of a speculative execution

\( x_i \leftarrow X.\text{LL}(); \)% \( x_i \) : local copy of \( X \)%

Statements (involving accesses to local memory and possibly accesses to the shared memory)
computing a new value for \( X \);
% this is the speculative execution %
if \( X.\text{SC}(v) \) then statement associated with success
else statement associated with failure
end if.

A simple universal construction


- Here we consider a simplified version with increasing sequence numbers

- Shared memory representation
  - a non-atomic collect object \( BOARD \) of size \( n \)
  - an array of \( n \) atomic memory locations \( STATE \)
The collect object

- Array BOARD[1..n] with one entry per process
- provides each $p_i$ with two operations: update() and collect()
- $BOARD.update(v)$ by process $p_i$: assigns $v$ to $BOARD[i]$
- $BOARD.collect()$: asynchronous scan of the array returning, for each entry $j$, the value read from $BOARD[j]$
- collect() is not atomic (⇐ asynchronous scan)
- $BOARD[i]$ contains a pair $⟨op, sn⟩$ where $op$ is the last operation on $O$ issued by $p_i$ and $sn$ is its seq number

STATE: the representation of the object $O$

STATE is a memory location made up of three fields

- $STATE.value$: current value of $O$
- $STATE.sn[1..n]$: array of seq numbers (init. $[0, \cdots, 0]$)
  \[ STATE.sn[i] = \text{seq number of } p_i \text{'s last invocation on } O \]
- $STATE.res[1..n]$: array of result values (init. $[⊥, \cdots, ⊥]$)
  \[ STATE.res[i] = \text{result of the last operation issued by } p_i \]

Local variable $sn_i$ at every process $p_i$ (init $1$)

The sequential specification of the object $O$

- Defined by a transition function $δ()$
- inputs:
  \[ ⋆ s: \text{the current state of } O \]
  \[ ⋆ op(in): \text{invocation of the operation } op(in) \text{ on } O \]
- $δ(s, op(in))$ outputs a pair $⟨s', r⟩$ such that
  \[ ⋆ s' \text{ is the state of } O \text{ after the execution of } op(in) \text{ on } s, \]
  \[ ⋆ \text{and } r \text{ is the result of } op(in) \]

Construction: operation invocation

when $p_i$ invokes $op(in)$ do
\[ BOARD.update(⟨op(in), sn_i⟩); \]
\[ sn_i ← sn_i + 1; \]
apply();
\[ \text{let } r = STATE.res[i]; \text{ return}(r). \]
Procedure apply() (1)

internal procedure apply() is
  ls ← STATE.LL();
  pairs ← BOARD.collect();
  for ℓ ∈ {1, 2, · · ·, n} do
    if (pairs[ℓ].sn = ls.sn[ℓ] + 1) then
      ⟨new_state, r⟩ ← δ(ls.value, pairs[ℓ].op);
      ls.res[ℓ] ← r; ls.sn[ℓ] ← pairs[ℓ].sn
    end if
  end for
  STATE.SC(ls)

Properties

• An operation cannot be executed more than once
• If a process does crash during its invocation, it terminates its operation (seq. asynchronous code)
• But is the result returned for the operation correct?

An execution

Final algorithm for apply()

internal procedure apply() is
  repeat twice
    ls ← STATE.LL();
    pairs ← BOARD.collect();
    for ℓ ∈ {1, 2, · · ·, n} do
      if (pairs[ℓ].sn = ls.sn[ℓ] + 1) then
        ⟨new_state, r⟩ ← δ(ls.value, pairs[ℓ].op);
        ls.res[ℓ] ← r; ls.sn[ℓ] ← pairs[ℓ].sn
      end if
    end for
    STATE.SC(ls)
  end repeat twice.

Cost: ≤ 2n (seq.) shared memory accesses
Linearization points of the operations

- Let SC[1], SC[2], ..., SC[x], ... be the sequence of the successful invocations of \textit{STATE.SC()}
- As \textit{STATE.SC()} is atomic, this sequence is well-defined
- Starting from SC[1], each SC[x] applies at least one operation on the object \(O\)
- The operations applied to \(O\) by each SC[x] are totally ordered
- Let seq[x] be the corresponding sequence
- The sequence of operations applied to \(O\) is then seq[1], seq[2], ..., seq[x], etc.

Exercise: build an atomic collect object

- Consider an atomic object \(X\) with two operations
  \begin{itemize}
  \item \(X.add(v)\) adds \(v\) to \(X\)
  \item \(X.read()\) returns the value of \(X\)
  \end{itemize}
- \(D\) = value domain of the entries of the collect object
- \(d\) = number of bits needed to represent a value of \(D\)
- \(X\) = atomic register of \(nd\) bits (\(n\) chunks of \(d\) bits)

The operations of the atomic collect objects

\[v' = \text{previous value written by } p_i, \text{ init 0}\]

operation \textit{update}(\(v\)) by \(p_i\) is
\[
\langle b_d, \ldots, b_1 \rangle \leftarrow \text{binary encoding of } (v - v');
\]
\[
\text{val} \leftarrow \langle 0, \ldots, 0, b_d, \ldots, b_1, 0, \ldots, 0 \rangle
\]
\[
\text{with } \langle b_d, \ldots, b_1 \rangle \text{ in position }[id...(i-1)d+1];
\]
\[
X.add(val); \ v' \leftarrow val;
\]
return.

operation \textit{collect()} is
\[
v \leftarrow X.read();
\]
\[
\text{decompose } v \text{ according to the } n\text{-chunk encoding; return (corresponding array } r[1..n]).
\]

Exercise: replace add() by xor()
The case of large objects

A large object is an object whose internal state cannot be copied in one atomic step (machine instruction)

- A large object is fragmented into blocks
- Pointers linking blocks: speculative execution with pointers manipulated with LL/SC
- Long array fragmented into blocks: implemented with LargeLL and LargeSC operations (built from LL/SC-based algorithm)

Extension 1: disjoint-access parallelism (1)

- A universal construction is disjoint-access parallel if two processes that access distinct parts of an object $O$ do not access common base objects or common memory location which constitute $O$’s internal representation
- As an example, let us consider a queue $Q$. When $|Q| \geq 3$, a disjoint-access parallelism implementation allows a process executing $\text{enqueue}(v)$ and a process executing $\text{dequeue}()$ to progress without interfering
- Is it possible to design a disjoint-access parallelism WF-compliant universal construction?

Example

```
enqueue(v)
```

```
dequeue()
```

Extension 1: disjoint-access parallelism (2)

- General impossibility result: Disjoint-access parallelism and wait-freedom are mutually exclusive when designing a universal construction
- Specific possibility result: Possible for the object class containing all the objects $O$ for which, in any sequential execution, each operation accesses a bounded number of base objects used to represent $O$.

  This class includes bounded trees, stacks and queues whose internal representations are list-based
Extension 2: abortable objects, definition

An **abortable object** is defined by a sequential specification and such that:

- When executed in a concurrency-free context, an operation takes effect, i.e., modifies the state of the object and returns a result as defined by its sequential specification.
- When executed in a concurrency context, an operation either takes effect and returns a result as defined by its sequential specification, or returns the default value \( \bot \) (abort).

An operation returning \( \bot \) has no effect on the state of the object.

The operations of an abortable object always terminate.

**k-abortable objects**

- An **operation** is allowed to abort only if it is concurrent with operations issued by \( k \) distinct processes and none of them returns \( \bot \) (abort).
- This means that the \( k \) operations that entail the abort of another operation must succeed.
- \( n \)-abortability is \( \bot \)-free wait-freedom.
- A (non-trivial) WF-compliant universal construction for \( k \)-abortable objects exists in \( CARW_n[LL/SC] \).


**WF-compliant universal const. for Abort. Objects**

- Successful speculative execution returns a value.
- Unsuccessful speculative execution returns \( \bot \) (occurs only in a concurrency pattern).

```plaintext
when \( p_i \) invokes \( op(in) \) do
  \( ls \leftarrow STATE.LL() \);
  \( \langle new\_state, r \rangle \leftarrow \delta(ls, op(in)) \);
  \( done \leftarrow STATE.SC(new\_state) \);
  if \( done \) then \( \text{return}(r) \) else \( \text{return}(\bot) \) end if.
```

Universal constructions

From operations on memory locations to agreement objects.
Hardware-provided uniform operations

- The previous universal constructions are based on hardware-provided atomic operations such as LL/SC.
- These hardware-provided atomic operations are uniform in the sense they can be applied to any memory location.
- Memory locations are not “objects” in the classical sense (e.g. a `push()` operation on a stack is meaningless on a set).

A few important questions

- Can we design WF-compliant universal constructions with hardware atomic operations such as Test&Set or Fetch&Add?
- Are all hardware atomic operations “equal” wrt WF-compliant universal constructions?
- Is it possible to generalize the concept of a universal construction to the coordinated construction of several objects with different progress conditions?

A fundamental object: Consensus

- A single operation denoted `propose()` that
  - a process can invoke only once
  - has an input parameter (proposed value) and a result (decided value)
- Consensus is defined by the following three properties:
  - **Validity.** A decided value is a proposed value
  - **Agreement.** No two processes decide different values
  - **Termination.** If a correct process invokes `propose()`, it decides

A simple consensus-based WF-compliant UC (1)

- Inspired from the state machine replication paradigm
- Each process $p_i$ manages
  - a local copy of the object $O$: $state_i$
  - an array $sn_i[1..n]$
  
  $sn_i[j] = \text{sequence number of the last operation on } O \text{ issued by } p_j, \text{ locally applied to } state_i$
A simple consensus-based WF-compliant UC (2)

Shared memory

- An array $BOARD[1..n]$ of SWMR atomic registers
  - $BOARD[j] = \langle BOARD[j].op, BOARD[j].sn \rangle$
  - $BOARD[j].op$ = last operation issued by $p_j$
  - $BOARD[j].sn$ = its sequence number
- $BOARD[j]$: initialized to $\langle \bot, 0 \rangle$
- An unbounded array $CONS[1..]$ of consensus objects


A simple consensus-based UC

\[\text{when } p_i \text{ invokes } op(in) \text{ do}\]
\[\text{done}_i \leftarrow false;\]
\[BOARD[i] \leftarrow \langle op(in), sn_i[i] + 1 \rangle;\]
\[\text{wait } (\text{done}_i);\]
\[\text{return } (res_i).\]

Strutural view of the Universal construction

Underlying local task $T(1)$

\[
\begin{align*}
\text{while } (\text{true}) \text{ do} & \\
\text{prop}_i & \leftarrow \epsilon; \quad \% \text{ empty list } \% \\
\text{for } j \in \{1, \ldots, n\} \text{ do} & \\
\text{if } (BOARD[j].sn > sn_i[j]) \text{ then} & \\
\text{append } (BOARD[j].op, j) \text{ to } prop_i & \\
\text{end} & \\
\text{end for}; & \\
\text{if } (prop_i \neq \epsilon) \text{ then see NEXT SLIDE end if} & \\
\text{end while}. & \\
\end{align*}
\]
Underlying local task $T$ (2)

\[
k_i \leftarrow k_i + 1;
list_i \leftarrow CONS[k_i].\text{propose}(prop_i);
\text{for } r = 1 \text{ to } |list_i| \text{ do}
\quad \langle state_i, res_i \rangle \leftarrow \delta(state_i, list_i[r].op);
\quad \text{let } j = list_i[r].\text{proc}; sn_i[j] \leftarrow sn_i[j] + 1;
\quad \text{if } (i = j) \text{ then } done_i \leftarrow \text{true} \text{ end if}
\text{end for}.
\]

Simple sequence of consensus instances to agree on the same sequence of operations applied to the object $O$

Bounded WF vs Unbounded WF

- **Bounded-wait-freedom:** the number of steps (accesses to the shared memory) executed before an operation terminates is bounded
- **Unbounded-wait-freedom:** the number of steps (accesses to the shared memory) executed before an operation terminates is finite (not bounded)
- This construction ensures that the operations issued by the processes are wait-free, but does not guarantee that they are bounded-wait-free (processes have to catch up)
- There are bounded WF universal constructions

A bounded WF universal construction

The object representation is in the shared memory

- A list of objects modifications + a helping mechanism
- Next pointers: consensus objects allowing the processes to agree on the sequence of operations applied to the object

Consensus number

- Let us consider an object of type $T$ (defined by a sequential specification)
- The consensus number of an object of type $T$ is the greatest integer $n$ such that it is possible to implement a consensus object in a system of $n$ processes, with any number of atomic read/write registers and objects of type $T$
- The consensus number is $+\infty$ if there is no largest $n$

The consensus hierarchy

- The consensus number of read/write registers is 1

  It follows that all objects that can be built from read/write registers only (i.e., in \( CARW_n[\emptyset] \) without enrichment with additional operations) have consensus number 1

- The consensus number of hardware operations such as Test&Set, Fetch&Add, Swap, and a few others, is 2

- Let a \( k \)-window read/write register be a register that stores only the sequence of the last \( k \) values which have been written, and whose read operation returns this sequence of at most \( k \) values. The consensus number of a \( k \)-window is \( k \)

- Finally, the consensus number of Compare&Swap, LL/SC, and a few others, is \( +\infty \)

Universality of consensus

- Consensus objects are universal in the sense they allow to WF-implement any object defined by a sequential specification in \( CARW_n[\emptyset] \)

- Any hardware-provided operation \( h_{op} \) whose consensus number is \( n \) is universal in \( CARW_n[\emptyset] \)

  This means that any object defined by a sequential specification can WF-implemented in \( CARW_n[h_{op}] \)

The problem

- The previous hierarchy considers consensus built from read/write registers and objects of a given type \( T \) only

- What can be done with when several hardware operations which access the same memory locations are given?

Universal constructions

Consensus from several operations on memory locations

Illustration

- System model $\mathcal{CARW}_n[\text{Test}&\text{Set}, \text{Fetch}&\text{Add2}]
  - Test&Set returns the value in the memory location, and sets it to 1 if it contained 0
  - Fetch&Add2 returns the value in the memory location and increases it by 2 (preserves parity: invariant)

- Test&Set and Fetch&Add2 have consensus number 2
- Which power has $\mathcal{CARW}_n[\text{Test}&\text{Set}, \text{Fetch}&\text{Add2}]$?

Binary consensus object for any $n$

A single memory location $X$, initialized to 0

operation propose($v$) is
  if ($v = 0$)
    then $x \leftarrow X.\text{fetch}&\text{add2}();$
      if ($x$ is odd) then return(1) else return(0) end if
  else $x \leftarrow X.\text{test}&\text{set}();$
    if ($x$ is odd) $\lor$ ($x = 0$)
      then return(1) else return(0)
  end if.

- Decision is sealed by the first atomic operation executed
- If the first operation executed is
  - fetch&add2(): $X$ becomes and remains even forever (decision 0)
  - test&set(): $X$ becomes and remains odd forever (decision 1)

Power number of an object type $T$

- Definition:
  The power number of an object type $T$ (PN($T$)) is the largest integer $k$ such that it is possible to implement a $k$-obstruction-free consensus object for any number of processes, using any number of atomic read/write registers, and any number of objects of type $T$ (the registers and the objects of type $T$ being wait-free)
  If there is no such largest $k$, PN($T$) = $+\infty$
- We have CN($T$) = PN($T$)
- Establish a strong relation linking wait-freedom and $k$-obstruction-freedom (progress conditions)

Universal constructions

“1 among $k$” and “$\ell$ among $k$”

Aim

- Consider $k$ objects (state machines, seq. specification)
- Design a WF-compliant universal construction such that
  - at least one object progresses forever
  - at least $\ell$ objects progress forever


Yet another agreement object: $k$-set agreement

$k$-SA is consensus where up to $k$ values can be decided

- Validity. A decided value is a proposed value
- Agreement. At most $k$ different values are decided
- Termination. If a correct process invokes propose(), it decides a value


$k$-set agreement vs $k$-SC

- In read/write systems: They are equivalent
- In message-passing systems: $k$-SC is strictly stronger than $k$-set agreement
Guerraoui-Gafni’s question

- Their question: **Is 1 a special value?** (wrt \( k \in [2..n] \))
- **\( k \)-set agreement:**
  - Allows up to \( k \) different values to be decided
  - 1-set agreement is consensus
- What they do:
  - They consider the implementation of \( k \) objects (each defined by a seq. specification) instead of only one, and “replace” consensus by \((k\)-simultaneous consensus \((= k\)-set agreement)\) objects
  - They provide a non-blocking universal construction in which at least one object progresses forever

Underlying basic object: **adopt-commit (1)**

- One-shot object
- A single operation denoted `propose()`, which
  - takes a value \( v \) as input parameter
  - and returns a pair \( \langle tag, v' \rangle \)


Underlying basic object: **adopt-commit (2)**

- **Validity:**
  - **Result domain:** Any returned pair \( \langle tag, v \rangle \) is such that (a) \( v \) has been proposed by a process and (b) \( tag \in \{commit, adopt\} \)
  - **No-conflicting values:** If a process \( p_i \) invokes `propose(v)` and returns before any other process \( p_j \) has invoked `propose(v')` with \( v' \neq v \), then only the pair \( \langle commit, v \rangle \) can be returned
- **Agreement:** If a process returns \( \langle commit, v \rangle \), only the pairs \( \langle commit, v \rangle \) or \( \langle adopt, v \rangle \) can be returned
- **Termination:**
  - The invocation of `propose()` by a correct process always terminates

  Can be implemented in \( CARW_n[\emptyset] \)

The heart of GG11 universal construction

- \( oper_i[\ m] = \text{next op on object } m \in [1..k] \text{ by } p_i \)
- One adopt-commit per round and object \( m \in [1..k] \)
  1. \( \langle obj, op \rangle \leftarrow KSC[r_i].propose(oper_i[1..k]) \);
  2. \( (tag_i[\ obj], ac\_op_i[\ obj]) \leftarrow AC[r_i][\ obj].propose(op) ; \)
  3. for each \( m \in \{1, ..., k \} \setminus \{obj\} \) do
     \( (tag_i[m], ac\_op_i[m]) \leftarrow AC[r_i][m].propose(oper_i[m]) \)
     end for
Why it works

At least one object operation is committed at every round

\[ \langle \text{adopt, -} \rangle \leftarrow AC[r][\text{obj1}].\text{propose()} \]

\[ \langle \text{obj1, -} \rangle \leftarrow KSC[r].\text{propose()} \]

\[ AC[r][\text{obj2}].\text{propose()} \]

\[ AC[r][\text{obj1}].\text{propose()} \]

\[ AC[r][\text{objx}].\text{propose()} \]

\[ AC[r][\text{obj2}].\text{propose()} \]

Summarizing GG11 Universal construction

- At least one process progresses forever: non-blocking
- At least one object progresses forever
- Hence, \( k \)-set agreement allow a coordinated NB-compliant universal construction of \( k \) objects (state machines), such that at least one object progresses forever

Beyond GG11 Universal construction!

- Design a coordinated WF-compliant universal construction of \( k \) objects (state machines), such that at least \( \ell \in [1..k] \) objects progress forever


RTS16 universal construction at a glance

- Introduces \( (k, \ell) \)-consensus objects \( (k, \ell \) constant)
- Considering \( k \) objects, it introduces a \( (k, \ell) \)-universal construction
  * in which \( \ell \) \( (1 \leq \ell \leq k) \) objects progress forever
  * in which the progress condition is wait-freedom
  * that is contention-aware (only read/write registers are used in the absence of contention)
  * that is generous wrt to the obstruction-freedom progress condition
- Shows that \( (k, \ell) \)-consensus objects are necessary and sufficient for such a \( (k, \ell) \)-universal construction
Remarks

- Contention awareness:
  \[ \text{Cost(Compare\&Swap)} \approx 1000 \times \text{Cost (read/write)} \]
- Generosity: “dual” of indulgence

\( (k, \ell) \)-simultaneous consensus (1)

- One-shot object
- A single operation denoted \texttt{propose()}, which
  \* takes a vector of size \( k \) as input parameter,
  \* and returns \( \ell \) pairs \( \langle x_1, v_1 \rangle, \ldots, \langle x_\ell, v_\ell \rangle \)
  (where all \( x_j \) are different)

Underlying basic objects: \( (k, \ell) \)-SC (2)

- \textbf{Validity}: A pair \( (x, v) \) returned by a process \( v \) has been proposed by a process in the \( x \)-th entry of its input vector
- \textbf{Agreement}: If a process returns \( \langle x, v \rangle \) and another process returns \( \langle y, v' \rangle \), then \( x = y \Rightarrow (v = v') \)
- \textbf{Termination}: An invocation of \texttt{propose()} by a correct process always terminates

The \( (k, \ell) \)-universal construction (1)

- First a non-blocking \( (k, 1) \)-universal construction is built
  \* It relies on copies of the views (histories) of each object by each process
  \* The consistency of these views is ensured thanks to \( (k, 1) \)-simultaneous consensus objects
  \* Each view is a full object history (seq. of operations)
  \* This facilitates the statement and the proof universal construction
  \* The full objects history can be eliminated, and replaced by registers containing the state of each object
The \((k, \ell)\)-universal construction (2)

- Then, one step after the other, the algorithm is enriched
  - to satisfy contention-awareness
  - to ensure wait-freedom of each object operation
- Finally the \((k, 1)\)-simultaneous consensus objects are replaced by \((k, \ell)\)-simultaneous consensus objects to obtain a wait-free, contention aware, \((k, \ell)\)-universal construction

Remarks

- When \(k = \ell = 1\), the universal construction obtained is the first contention-aware \((1, 1)\)-universal construction
- More generally, when \(\ell = 1\), the resulting construction is the first contention-aware \((k, 1)\)-universal construction

Conclusion

- Quest for distributed universal constructions is at the heart of distributed computability
- Understand distributed computability is mainly concerned by mastering uncertainty (non-determinism) created by the environment (mainly asynchrony, failures, and concurrency)
- This quest is far from being finished...
- Still remain to have a deeper understanding of the relations between shared memory systems, message-passing communication abstractions, and agreement objects