Adopt-commit
Universal Constructions
BG-simulation

Julien Stainer

Concurrent Algorithms
Distributed Programming Laboratory

Table of Contents

Adopt-commit is a one-shot distributed object.

- It offers the operation PROPOSE(*value*),

Adopt-commit is a one-shot distributed object.

- It offers the operation $\textsc{propose}(\mathit{value})$,
- returns $\langle \mathit{tag}, \mathit{value} \rangle$, $\mathit{tag} \in \{\mathit{commit}, \mathit{adopt}\}$.

Adopt-commit is a one-shot distributed object.

- It offers the operation $\textsc{propose}(\mathit{value})$,
- returns $\langle \mathit{tag}, \mathit{value} \rangle$, $\mathit{tag} \in \{\mathit{commit}, \mathit{adopt}\}$.

Validity Any returned value has been proposed.
If a process p_i invokes propose(v) and returns before any other process p_j invokes propose(v') with $v' \neq v$, then only $\langle \mathit{commit}, v \rangle$ is returned.

Adopt-commit is a one-shot distributed object.

- It offers the operation PROPOSE($value$),
- returns $\langle tag, value \rangle$, $tag \in \{commit, adopt\}$.

Validity Any returned value has been proposed.
If a process p_i invokes propose(v) and returns before any other process p_j invokes propose(v') with $v' \neq v$, then only $\langle commit, v \rangle$ is returned.

Agreement If a process returns $\langle commit, v \rangle$, the only pairs that can be returned are $\langle commit, v \rangle$ and $\langle adopt, v \rangle$.

Adopt-commit is a one-shot distributed object.

- It offers the operation PROPOSE($value$),
- returns $\langle tag, value \rangle$, $tag \in \{commit, adopt\}$.

Validity	Any returned value has been proposed. If a process p_i invokes propose(v) and returns before any other process p_j invokes propose(v') with $v' \neq v$, then only $\langle commit, v \rangle$ is returned.
Agreement	If a process returns $\langle commit, v \rangle$, the only pairs that can be returned are $\langle commit, v \rangle$ and $\langle adopt, v \rangle$.
Termination	An invocation of PROPOSE() by a correct process terminates.

Adopt-commit Implementation

- n processes

Adopt-commit Implementation

- n processes
- Two arrays of n SWMR atomic registers $A[j]$ and $B[j]$, $0 \leq j \leq n-1$

- n processes
- Two arrays of n SWMR atomic registers $A[j]$ and $B[j]$, $0 \leq j \leq n-1$
- All registers initialized to the special value $\perp$

Adopt-commit Implementation

- n processes
- Two arrays of n SWMR atomic registers $A[j]$ and $B[j]$, $0 \leq j \leq n-1$
- All registers initialized to the special value $\perp$
- Two local sets a_i and b_i

Adopt-commit Implementation

```
 1: operation AC.PROPOSE(v_i)
 2:     A[i] ← v_i; a_i ← ∅
 3:     for j from 0 to n − 1 do
 4:         tmp_i ← A[j]
 5:         if tmp_i ≠ ⊥ then a_i ← a_i ∪ {tmp_i} end if
 6:     end for
 7:     if a_i = {v} then B[i] ← ⟨one, v⟩ else B[i] ← ⟨more, v_i⟩ end if
 8:     b_i ← ∅
 9:     for j from 0 to n − 1 do
10:         tmp_i ← B[j]
11:         if tmp_i ≠ ⊥ then b_i ← b_i ∪ {tmp_i} end if
12:     end for
13:     if b_i = {⟨one, v⟩} then
14:         return(⟨commit, v⟩)
15:     else if ∃⟨one, v⟩ ∈ b_i then
16:         return(⟨adopt, v⟩)
17:     else
18:         return(⟨adopt, v_i⟩)
19:     end if
20: end operation
```

- Validity and termination are straightforward.

Elements of Proof

- Validity and termination are straightforward.
- Agreement: at most one value can appear with the tag *one*.

- Implements consensus.

Adopt-commit-based Consensus

- Implements consensus.
- Implementation stripped in sequence of asynchronous rounds.

- Implements consensus.
- Implementation stripped in sequence of asynchronous rounds.
- Based on an infinite array of Adopt-commit objects $AC[r]$, $r \geq 0$.

- Implements consensus.
- Implementation stripped in sequence of asynchronous rounds.
- Based on an infinite array of Adopt-commit objects $AC[r]$, $r \geq 0$.
- A shared MWMR register DEC initialized to $\perp$.

```
1:  operation CONS.PROPOSE(v_i)
2:      est_i ← v_i; r_i ← 0
3:      while DEC = ⊥ do
4:          if leader_i = i then
5:              r_i ← r_i + 1
6:              ⟨tag_i, val_i⟩ ← AC[r_i].PROPOSE(est_i)
7:              if tag_i = commit then
8:                  DEC ← val_i
9:              else
10:                 est_i ← val_i
11:             end if
12:         end if
13:     end while
14:     return DEC
15: end operation
```

Validity

- Straightforward.

Validity

- Straightforward.

Validity
- Straightforward.

Agreement
- No two processes decide differently at the same round.

Validity

- Straightforward.

Agreement

- No two processes decide differently at the same round.
- After the first round r at which a process decides a value v, the estimates of all processes in the following rounds $r' > r$ are all v.

Validity

- Straightforward.

Agreement

- No two processes decide differently at the same round.
- After the first round r at which a process decides a value v, the estimates of all processes in the following rounds $r' > r$ are all v.

Elements of Proof

Validity
- Straightforward.

Agreement
- No two processes decide differently at the same round.
- After the first round r at which a process decides a value v, the estimates of all processes in the following rounds $r' > r$ are all v.

Termination
- If eventually one and only one correct process verifies $leader_i = i$ then any correct process eventually decides.

Elements of Proof

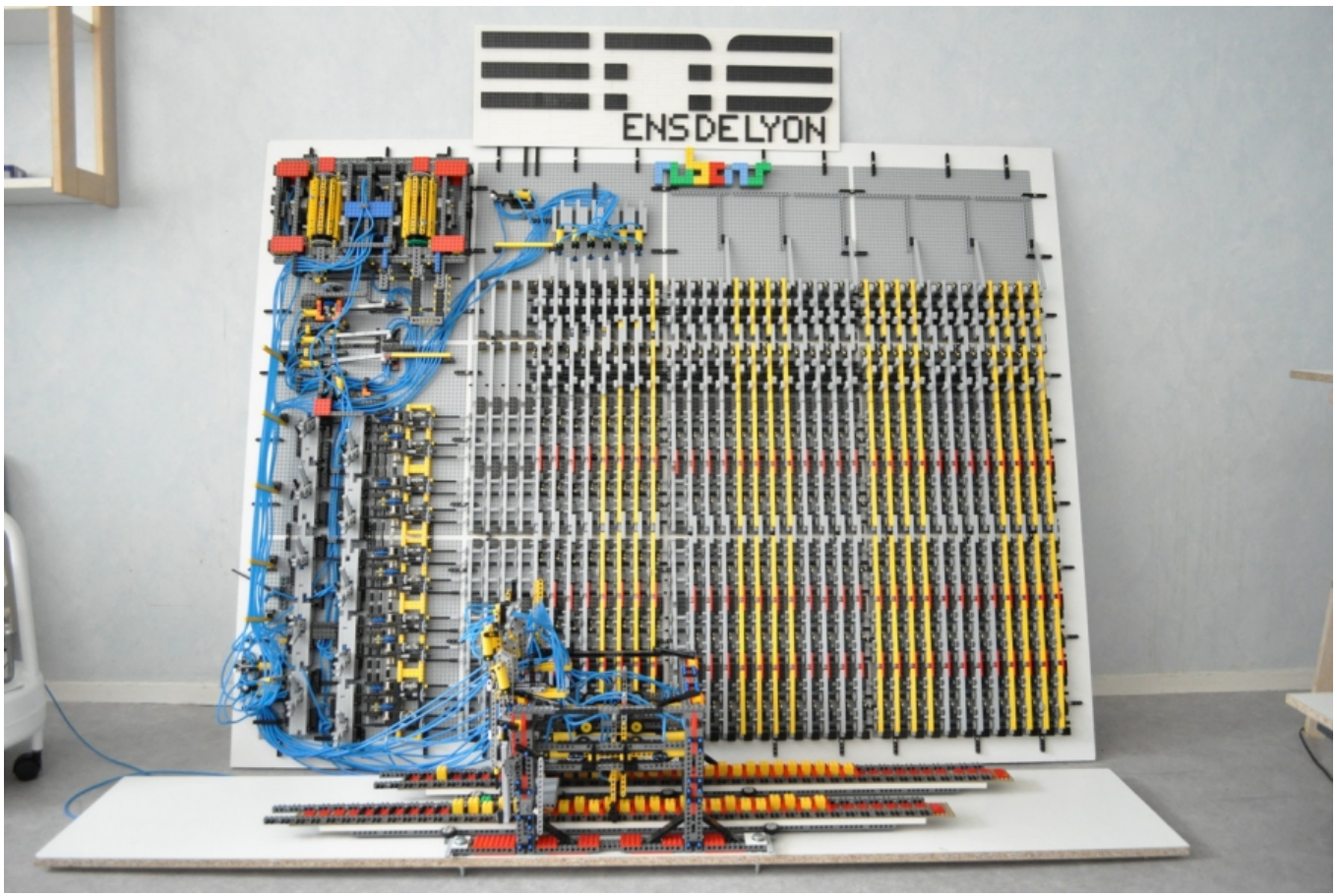
Validity
- Straightforward.

Agreement
- No two processes decide differently at the same round.
- After the first round r at which a process decides a value v, the estimates of all processes in the following rounds $r' > r$ are all v.

Termination
- If eventually one and only one correct process verifies $leader_i = i$ then any correct process eventually decides.
- If all processes verify $leader_i = i$ forever, the algorithm is only *obstruction-free*.

Table of Contents

Universality of the Turing Machine



Consensus is *universal*

Any object O following a sequential specification can be implemented, in a wait-free and linearizable manner, from atomic registers and consensus objects.[1]

[1] Maurice Herlihy: *Wait-Free Synchronization.* ACM TOPLAS (1991)

Consensus is *universal*

Any object O following a sequential specification can be implemented, in a wait-free and linearizable manner, from atomic registers and consensus objects.[1]

If we know how to solve consensus in our system, we can implement a highly available Turing machine.

[1] Maurice Herlihy: *Wait-Free Synchronization.* ACM TOPLAS (1991)

What kind of universality can we achieve
without consensus?

A weaker agreement: k-set agreement.

Interface Offers a $\textsc{propose}(v)$ operation that returns a value.

A weaker agreement: *k*-set agreement.

Interface Offers a PROPOSE(v) operation that returns a value.

Validity Decided values are proposed values.

A weaker agreement: *k*-set agreement.

Interface Offers a PROPOSE(v) operation that returns a value.

Validity Decided values are proposed values.

Termination Any invocation of PROPOSE by a correct process terminates.

A weaker agreement: *k*-set agreement.

Interface Offers a PROPOSE(v) operation that returns a value.

Validity Decided values are proposed values.

Termination Any invocation of PROPOSE by a correct process terminates.

Agreement No more than k different values are decided in the system.

Another generalization of consensus: k-simultaneous consensus.

Interface Offers a $\textsc{propose}(v_1, \ldots, v_k)$ operation that returns a pair $(\mathit{index}, \mathit{value})$, $\mathit{index} \in \{1, \ldots, k\}$.

Another generalization of consensus: *k*-simultaneous consensus.

Interface Offers a PROPOSE($v_1, \ldots, v_k$) operation that returns a pair (*index*, *value*), *index* $\in \{1, \ldots, k\}$.

Validity If a PROPOSE operation returns (i, v), then a process invoked PROPOSE($v_1, \ldots, v_k$) with $v_i = v$.

Another generalization of consensus: *k*-simultaneous consensus.

Interface Offers a PROPOSE($v_1, \ldots, v_k$) operation that returns a pair ($index, value$), $index \in \{1, \ldots, k\}$.

Validity If a PROPOSE operation returns (i, v), then a process invoked PROPOSE($v_1, \ldots, v_k$) with $v_i = v$.

Termination Any invocation of PROPOSE by a correct process terminates.

Another generalization of consensus: *k*-simultaneous consensus.

Interface Offers a PROPOSE$(v_1, \ldots, v_k)$ operation that returns a pair $(index, value)$, $index \in \{1, \ldots, k\}$.

Validity If a PROPOSE operation returns (i, v), then a process invoked PROPOSE$(v_1, \ldots, v_k)$ with $v_i = v$.

Termination Any invocation of PROPOSE by a correct process terminates.

Agreement If two PROPOSE operations return (i, v) and (i', v') with $i = i'$, then $v = v'$.

- Both 1-set agreement and 1-simultaneous consensus are equivalent to consensus.

- Both 1-set agreement and 1-simultaneous consensus are equivalent to consensus.
- k-set agreement and k simultaneous consensus are equivalent in asynchronous shared memory systems in presence of an arbitrary number of crashes.

- Both 1-set agreement and 1-simultaneous consensus are equivalent to consensus.
- k-set agreement and k simultaneous consensus are equivalent in asynchronous shared memory systems in presence of an arbitrary number of crashes.
- k-set agreement cannot be implemented in asynchronous shared memory systems prone to $t \geq k$ crashes.

Generalized Universality

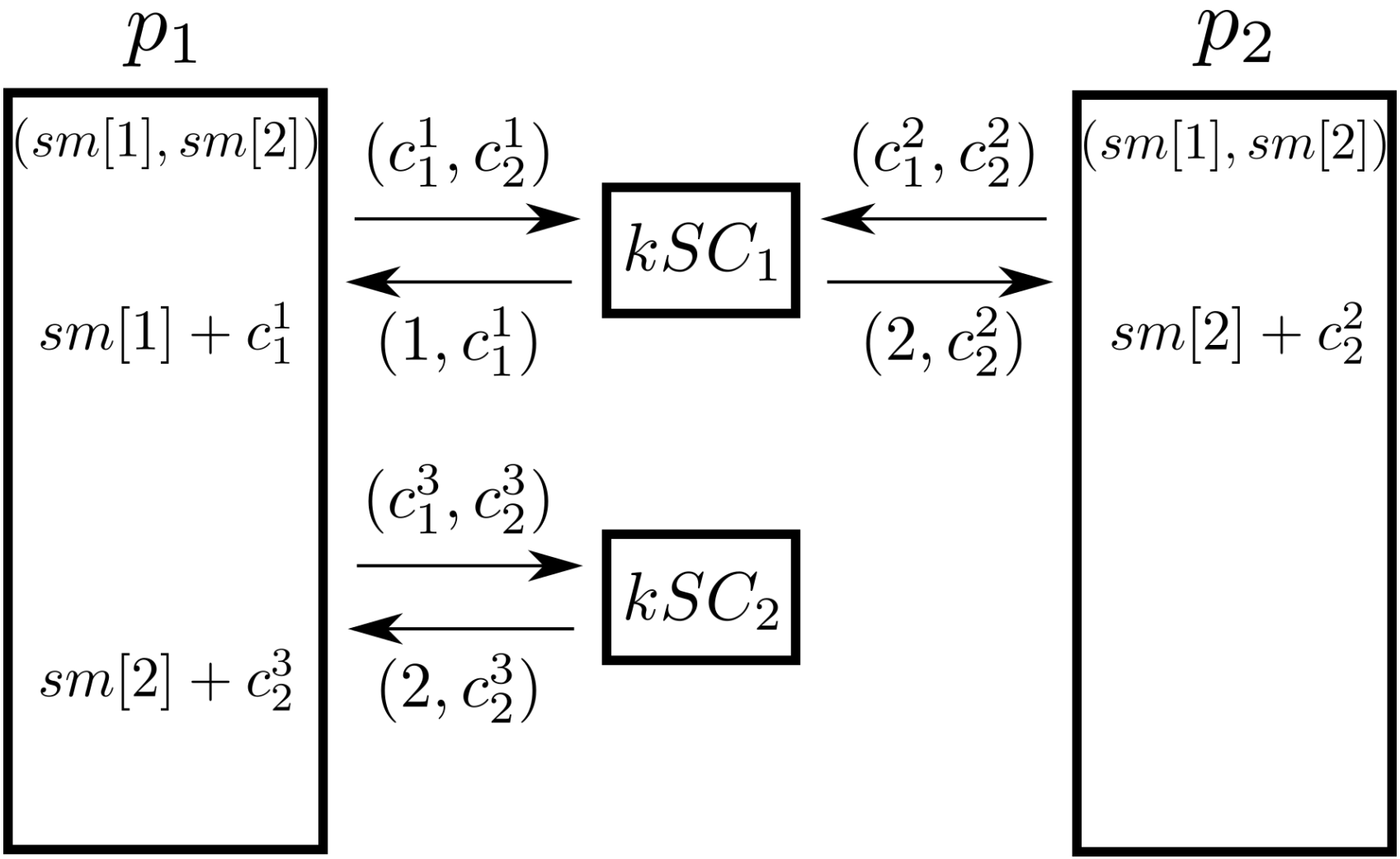
From k-simultaneous consensus objects and registers, it is possible to implement k shared objects of which at least one is highly available[2].

[2] Gafni E. and Guerraoui R., *Generalizing universality.* CONCUR (2011)

From Standard Universal Construction...

1: **while** true **do**
2: $c \leftarrow commands.next()$
3: $CONS \leftarrow consensus.next()$
4: $c' \leftarrow CONS.\text{PROPOSE}(c)$
5: $sm.perform(c')$
6: **end while**

1: **while** true **do**
2: **for** j from 1 to k **do**
3: $c[j] \leftarrow commands[j].next()$
4: **end for**
5: $kSC \leftarrow k\text{-}sim\text{-}cons.next()$
6: $(i, dc) \leftarrow kSC.\text{PROPOSE}(c[1], \ldots, c[k])$
7: $sm[i].perform(dc)$
8: **end while**


p_1
p_2
$(sm[1], sm[2])$
$(sm[1], sm[2])$
(c_1^1, c_2^1)
(c_1^2, c_2^2)
kSC_1
$sm[1] + c_1^1$
$(1, c_1^1)$
$(2, c_2^2)$
$sm[2] + c_2^2$
(c_1^3, c_2^3)
kSC_2
$sm[2] + c_2^3$
$(2, c_2^3)$

- Each process needs to keep track of the operations applied on the different machines.

Problem with Safety

- Each process needs to keep track of the operations applied on the different machines.
- They need to communicate the commands they apply and to retrieve the commands of the other processes.

Problem with Safety

- Each process needs to keep track of the operations applied on the different machines.
- They need to communicate the commands they apply and to retrieve the commands of the other processes.
- Adopt-commit objects may help...

```
 1: while true do
 2:     for j from 1 to k do
 3:         if c[j] = ⊥ then c[j] ← commands[j].next() end if
 4:     end for
 5:     kSC ← k-sim-cons.next()
 6:     (i, dc) ← kSC.PROPOSE(c[1], ..., c[k])
 7:     for j from 1 to k do
 8:         AC[j] ← adopt-commit[j].next()
 9:         if i = j then
10:             ⟨tag[j], ac_com[j]⟩ ← AC[j].PROPOSE(dc)
11:         else
12:             ⟨tag[j], ac_com[j]⟩ ← AC[j].PROPOSE(c[j])
13:         end if
14:         if tag[j] = commit then
15:             sm[j].perform(ac_com[j]); c[j] ← ⊥
16:         else
17:             c[j] ← ac_com[j]
18:         end if
19:     end for
20: end while
```

p_1 $(1, c_1^1) \leftarrow kSC.\text{PROPOSE}(c_1^1, c_2^1)$

p_1 $(1, c_1^1) \leftarrow kSC.\text{PROPOSE}(c_1^1, c_2^1)$

p_2 $(2, c_2^2) \leftarrow kSC.\text{PROPOSE}(c_1^2, c_2^2)$

Problem with Liveness

$$
\begin{aligned}
p_1 \quad & (1, c_1^1) \leftarrow kSC.\textsc{propose}(c_1^1, c_2^1) \\
p_2 \quad & (2, c_2^2) \leftarrow kSC.\textsc{propose}(c_1^2, c_2^2) \\
p_1 || p_2 \quad & AC[1].\textsc{propose}(c_1^1) || AC[1].\textsc{propose}(c_1^2)
\end{aligned}
$$

Problem with Liveness

$$
\begin{aligned}
p_1 \quad & (1, c_1^1) \leftarrow kSC.\textsc{propose}(c_1^1, c_2^1) \\
p_2 \quad & (2, c_2^2) \leftarrow kSC.\textsc{propose}(c_1^2, c_2^2) \\
p_1||p_2 \quad & AC[1].\textsc{propose}(c_1^1)||AC[1].\textsc{propose}(c_1^2) \\
p_1||p_2 \quad & AC[2].\textsc{propose}(c_2^1)||AC[2].\textsc{propose}(c_2^2)
\end{aligned}
$$

$$
\begin{array}{rl}
p_1 & (1, c_1^1) \leftarrow kSC.\text{PROPOSE}(c_1^1, c_2^1) \\
p_2 & (2, c_2^2) \leftarrow kSC.\text{PROPOSE}(c_1^2, c_2^2) \\
p_1||p_2 & AC[1].\text{PROPOSE}(c_1^1)||AC[1].\text{PROPOSE}(c_1^2) \\
p_1||p_2 & AC[2].\text{PROPOSE}(c_2^1)||AC[2].\text{PROPOSE}(c_2^2)
\end{array}
$$

The four adopt-commit can return $\langle adopt, - \rangle$...
It can be repeated forever without any progress.

Adopt-commit guarantees a *commit* if a PROPOSE terminates before any other value is proposed.

Adopt-commit guarantees a *commit* if a PROPOSE terminates before any other value is proposed.

The k-simultaneous consensus does not return more than one command per machine.

Adopt-commit guarantees a *commit* if a PROPOSE terminates before any other value is proposed.

The k-simultaneous consensus does not return more than one command per machine.

Exploit Success First

Let's launch the processes first on the machines returned by the k-simultaneous consensus.

Exploit Success First

```
 1: while true do
 2:     for j from 1 to k do
 3:         if c[j] = ⊥ then c[j] ← commands[j].next() end if
 4:     end for
 5:     kSC ← k-sim-cons.next()
 6:     ⟨i, dc⟩ ← kSC.PROPOSE(c[1], ..., c[k])
 7:     AC[i] ← adopt-commit[i].next()
 8:     ⟨tag[i], ac_com[i]⟩ ← AC[i].PROPOSE(dc)
 9:     for j from 1 to k, j ≠ i do
10:         AC[j] ← adopt-commit[j].next()
11:         ⟨tag[j], ac_com[j]⟩ ← AC[j].PROPOSE(c[j])
12:         if tag[j] = commit then
13:             sm[j].perform(ac_com[j]); c[j] ← ⊥
14:         else
15:             c[j] ← ac_com[j]
16:         end if
17:     end for
18: end while
```

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.
- p_y necessarily (a) got a pair $\langle i', dc' \rangle$ with $i \neq i'$ from the k-simultaneous consensus,

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.
- p_y necessarily (a) got a pair $\langle i', dc' \rangle$ with $i \neq i'$ from the k-simultaneous consensus,
- and (b) already finished executing $AC[i'].\text{PROPOSE}$.

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.
- p_y necessarily (a) got a pair $\langle i', dc' \rangle$ with $i \neq i'$ from the k-simultaneous consensus,
- and (b) already finished executing $AC[i'].\text{PROPOSE}$.
- If p_y didn't commit, then another process concurrently accessed $AC[i']$.

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.
- p_y necessarily (a) got a pair $\langle i', dc' \rangle$ with $i \neq i'$ from the k-simultaneous consensus,
- and (b) already finished executing $AC[i'].\text{PROPOSE}$.
- If p_y didn't commit, then another process concurrently accessed $AC[i']$.
- But it cannot be p_x!

Exploit Success First

- If a process p_x gets $\langle i, dc \rangle$ from the k-simultaneous consensus and does not commit dc to $AC[i]$,
- then another process p_y concurrently proposed another value to $AC[i]$.
- p_y necessarily (a) got a pair $\langle i', dc' \rangle$ with $i \neq i'$ from the k-simultaneous consensus,
- and (b) already finished executing $AC[i'].\text{PROPOSE}$.
- If p_y didn't commit, then another process concurrently accessed $AC[i']$.
- But it cannot be p_x!
- Now there is at least a commit per round.

Still some Safety Problems

We now commit at each round and each process at least adopts each of the committed values. But commands can be skipped.

p_1 commits and apply a command c on machine m

We now commit at each round and each process at least adopts each of the committed values. But commands can be skipped.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

We now commit at each round and each process at least adopts each of the committed values. But commands can be skipped.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_1 proposes c' for machine m

Still some Safety Problems

We now commit at each round and each process at least adopts each of the committed values. But commands can be skipped.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_1 proposes c' for machine m

p_1 commits c' for machine m

We now commit at each round and each process at least adopts each of the committed values. But commands can be skipped.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_1 proposes c' for machine m

p_1 commits c' for machine m

p_2 commits/adopt c' for m?

We now commit at each round, but maybe twice the same command on the same machine in consecutive rounds.

p_1 commits and apply a command c on machine m

We now commit at each round, but maybe twice the same command on the same machine in consecutive rounds.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

Still some Safety Problems

We now commit at each round, but maybe twice the same command on the same machine in consecutive rounds.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_2 proposes c for machine m

Still some Safety Problems

We now commit at each round, but maybe twice the same command on the same machine in consecutive rounds.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_2 proposes c for machine m

p_2 commits c for machine m

Still some Safety Problems

We now commit at each round, but maybe twice the same command on the same machine in consecutive rounds.

p_1 commits and apply a command c on machine m

p_2 adopts c for machine m

p_2 proposes c for machine m

p_2 commits c for machine m

p_1 commits/adopt c for m?

We can solve the problems of skipped and doubled commands by:

- Piggy-backing the previous command in the currently proposed one.

[3] Michel Raynal, Julien Stainer, Gadi Taubenfeld: *Distributed Universality*. OPODIS (2014)

We can solve the problems of skipped and doubled commands by:

- Piggy-backing the previous command in the currently proposed one.
- When a command is committed, the local history is checked to verify

[3] Michel Raynal, Julien Stainer, Gadi Taubenfeld: *Distributed Universality*. OPODIS (2014)

We can solve the problems of skipped and doubled commands by:

- Piggy-backing the previous command in the currently proposed one.
- When a command is committed, the local history is checked to verify
    (a) if the committed command has not already been applied;

[3] Michel Raynal, Julien Stainer, Gadi Taubenfeld: *Distributed Universality*. OPODIS (2014)

We can solve the problems of skipped and doubled commands by:

- Piggy-backing the previous command in the currently proposed one.
- When a command is committed, the local history is checked to verify
  - (a) if the committed command has not already been applied;
  - (b) if the previous command has already been applied, if not, apply both commands.

[3] Michel Raynal, Julien Stainer, Gadi Taubenfeld: *Distributed Universality*. OPODIS (2014)

We can solve the problems of skipped and doubled commands by:

- Piggy-backing the previous command in the currently proposed one.
- When a command is committed, the local history is checked to verify
  - (a) if the committed command has not already been applied;
  - (b) if the previous command has already been applied, if not, apply both commands.

- Histories can also be exchanged directly through the shared memory.[3]

[3] Michel Raynal, Julien Stainer, Gadi Taubenfeld: *Distributed Universality*. OPODIS (2014)

From k-simultaneous consensus objects and atomic register, it is possible to simulate k state-machines such that at least one always progresses.

- The current algorithm is non-blocking, some processes may never apply any command to the machines.

- The current algorithm is non-blocking, some processes may never apply any command to the machines.
- It can become wait-free by the use of helping:

- The current algorithm is non-blocking, some processes may never apply any command to the machines.
- It can become wait-free by the use of helping:
  - Processes write in shared memory the commands they plan to execute on the machines.

- The current algorithm is non-blocking, some processes may never apply any command to the machines.
- It can become wait-free by the use of helping:
  - Processes write in shared memory the commands they plan to execute on the machines.
  - While deciding the next command to apply on a machine m, processes check the number of commands nc that have been applied to m.

- The current algorithm is non-blocking, some processes may never apply any command to the machines.
- It can become wait-free by the use of helping:
  - Processes write in shared memory the commands they plan to execute on the machines.
  - While deciding the next command to apply on a machine m, processes check the number of commands nc that have been applied to m.
  - If process p_x with $x = nc \mod n$ has written a command that has not been executed, then other processes propose it as $(nc + 1)$-th command to execute on m.

- When there is no contention (e.g. a process is far ahead), the use of the k-simultaneous consensus object can be avoided, at the cost of more adopt-commit objects.

- When there is no contention (e.g. a process is far ahead), the use of the k-simultaneous consensus object can be avoided, at the cost of more adopt-commit objects.
- To guarantee that several machines progress, the k-simultaneous consensus objects can be replaced by more powerful objects.

Table of Contents

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement At most one value is decided.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement At most one value is decided.

Validity A decided value is a proposed value.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement At most one value is decided.

Validity A decided value is a proposed value.

In a crash-free system, safe-agreement objects implement consensus.

1: **init** $REG[0, \ldots, n-1] \leftarrow [\langle \perp, 0 \rangle]$
2: **operation** PROPOSE(v)
3: $REG[i] \leftarrow \langle v, 1 \rangle$
4: $snap_i \leftarrow REG.snapshot()$
5: **if** $\exists x \ : \ snap_i[x].level = 2$ **then**
6: $REG[i] \leftarrow \langle v, 0 \rangle$
7: **else**
8: $REG[i] \leftarrow \langle v, 2 \rangle$
9: **end if**
10: **end operation**
11: **operation** DECIDE( )
12: **repeat**
13: $snap_i \leftarrow REG.snapshot()$
14: **until** $\forall x \ : \ snap_i[x].level \neq 1$
15: $x \leftarrow \min \{y \mid snap_i[y] = 2\}$
16: return $snap_i[x].value$
17: **end operation**

The BG-Simulation allows to wait-free simulate a larger system while preserving the number of crashes.

- $t+1$ simulators $q_0, \ldots, q_t$ among which up to t may crash.

The BG-Simulation allows to wait-free simulate a larger system while preserving the number of crashes.

- $t+1$ simulators $q_0, \ldots, q_t$ among which up to t may crash.
- n simulated processes $p_0, \ldots, p_{n-1}$ communicating by writes and snapshots.

The BG-Simulation allows to wait-free simulate a larger system while preserving the number of crashes.

- $t+1$ simulators $q_0, \ldots, q_t$ among which up to t may crash.
- n simulated processes $p_0, \ldots, p_{n-1}$ communicating by writes and snapshots.

- Each simulator simulates in parallel each of the simulated processes.

The BG-Simulation allows to wait-free simulate a larger system while preserving the number of crashes.

- $t+1$ simulators $q_0, \ldots, q_t$ among which up to t may crash.
- n simulated processes $p_0, \ldots, p_{n-1}$ communicating by writes and snapshots.

- Each simulator simulates in parallel each of the simulated processes.
- They use the shared memory available to the simulators to simulate writes and snapshots of the simulated processes.

The BG-Simulation allows to wait-free simulate a larger system while preserving the number of crashes.

- $t+1$ simulators $q_0, \ldots, q_t$ among which up to t may crash.
- n simulated processes $p_0, \ldots, p_{n-1}$ communicating by writes and snapshots.

- Each simulator simulates in parallel each of the simulated processes.
- They use the shared memory available to the simulators to simulate writes and snapshots of the simulated processes.

To preserve coherence, simulators have to agree on the snapshots taken by the simulated processes.

1: **init** $r_i \leftarrow 1$
2: **while** p_i not decided **do**
3: simulate its r_i-th write on behalf of p_i
4: simulate its r_i-th snapshot on behalf of p_i
5: propose this snapshot to the r_i-th safe-agreement object associated to p_i
6: decide on a snapshot from this safe-agreement object
7: compute the new state of p_i
8: $r_i \leftarrow r_i + 1$
9: **end while**

- Simulators agree on the state of simulated processes

- Simulators agree on the state of simulated processes

- But the crash of a simulator can block more than one simulated process.

1: **init** $r_i \leftarrow 1$
2: **while** p_i not decided **do**
3: simulate its r_i-th write on behalf of p_i
4: simulate its r_i-th snapshot on behalf of p_i
5: enter mutex
6: propose this snapshot to the r_i-th safe-agreement object associated to p_i
7: exit mutex
8: decide on a snapshot from this safe-agreement object
9: compute the new state of p_i
10: $r_i \leftarrow r_i + 1$
11: **end while**

- Thanks to the mutex, a simulator never participates to more than one safe-agreement propose operation.

- Thanks to the mutex, a simulator never participates to more than one safe-agreement propose operation.
- The crash of a simulator consequently blocks at most one simulated process.

- Consensus is impossible in a system of 2 processes with 1 crash

  $\Longrightarrow$ consensus is impossible in a system of 100 processes with 1 crash.

- Consensus is impossible in a system of 2 processes with 1 crash
  $\Longrightarrow$ consensus is impossible in a system of 100 processes with 1 crash.

- k-set agreement is impossible in a system of $k+1$ processes with k crashes
  $\Longrightarrow$ k-set agreement is impossible in a system of 100 processes with k crashes.

What matters in a system is not the number of processes but the maximum number of crashes.

- Adopt-commit and adopt-commit-based consensus
- Universal construction from k-simultaneous consensus objects and registers
- BG-simulation