Tight bounds for anonymous adopt-commit objects

James Aspnes[1] Faith Ellen[2]

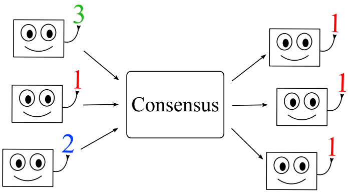
[1]Yale

[2]Toronto

June 6th, 2011

Motivation

What we really care about is shared-memory consensus:



- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody's input.
- **Agreement:** All output values are equal.

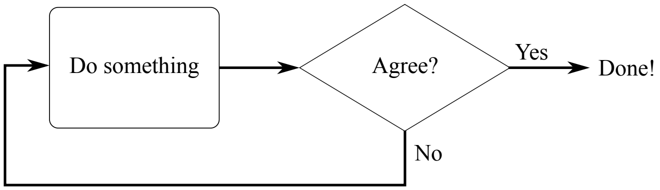
Model

Usual asynchronous shared-memory model:

- n concurrent processes.
- Communication by reading and writing atomic registers.
- Asynchronous, with timing controlled by an **adversary scheduler**.
- **Wait-free**: each process finishes in a finite number of steps.

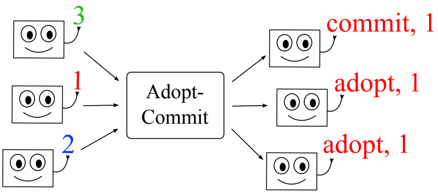
We will be considering **anonymous** algorithms in which all processes run the same code.

Implementing consensus



- Typical implementation: use some randomized process that produces agreement with some probability, and commit to a return value when we detect agreement.
- But how to detect agreement?

Adopt-commit objects

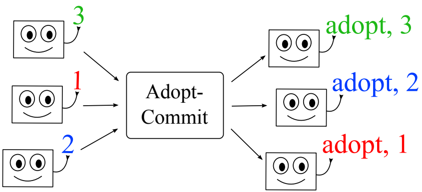


(Gafni, PODC 1998; Mostefaoui *et al.*, SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody's input.
- ~~**Agreement:** All output values are equal.~~
- **Coherence:** All output values are equal *if* some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.

Adopt-commit objects

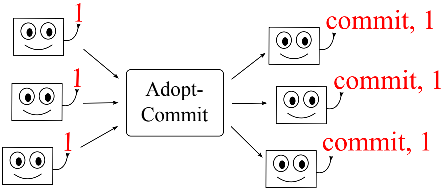


(Gafni, PODC 1998; Mostefaoui *et al.*, SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody's input.
- ~~**Agreement:** All output values are equal.~~
- **Coherence:** All output values are equal *if* some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.

Adopt-commit objects

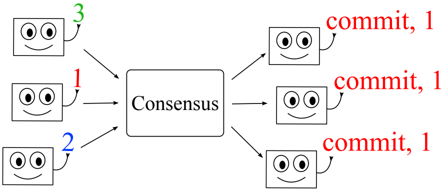


(Gafni, PODC 1998; Mostefaoui *et al.*, SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody's input.
- ~~**Agreement:** All output values are equal.~~
- **Coherence:** All output values are equal *if* some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.

Adopt-commit objects

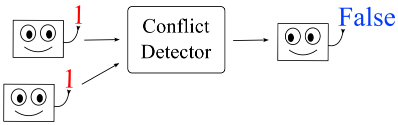


(Gafni, PODC 1998; Mostefaoui *et al.*, SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody's input.
- ~~**Agreement:** All output values are equal.~~
- **Coherence:** All output values are equal *if* some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.

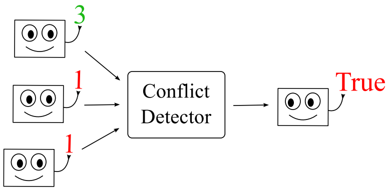
Conflict detectors



We show that adopt-commit is equivalent (up to small constants) to a **conflict detector**:

- Two operations: `write` and `read`.
- The `read` operation returns **true** if distinct values have previously been written, otherwise **false**.

Conflict detectors



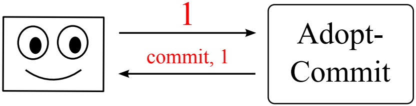
We show that adopt-commit is equivalent (up to small constants) to a **conflict detector**:

- Two operations: `write` and `read`.
- The `read` operation returns **true** if distinct values have previously been written, otherwise **false**.

Conflict detector from adopt-commit

procedure `write(`v`)`
begin
  if `adoptCommit`$(v) \neq$ (commit, v) **then**
    conflict $\leftarrow$ **true**
  end
end

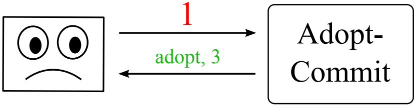
procedure `read()`
begin
  return conflict
end



Conflict detector from adopt-commit

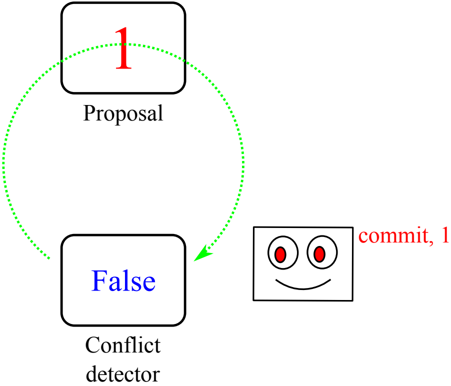
procedure `write(`v`)`
begin
  if `adoptCommit`$(v) \neq (\text{commit}, v)$ **then**
    conflict $\leftarrow$ **true**
  end
end

procedure `read()`
begin
  return conflict
end



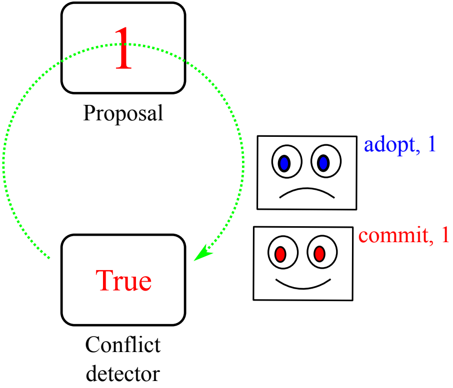
Adopt-commit from conflict detector

procedure `adoptCommit`(v)
begin
  conflict.`write`(v)
  $u \leftarrow$ proposal
  if $u = \perp$ **then**
    proposal $\leftarrow v$
  else
    $v \leftarrow u$
  end
  if conflict.`read`() $=$ **false**
  then
    return (commit, v)
  else
    return (adopt, v)
  end
end



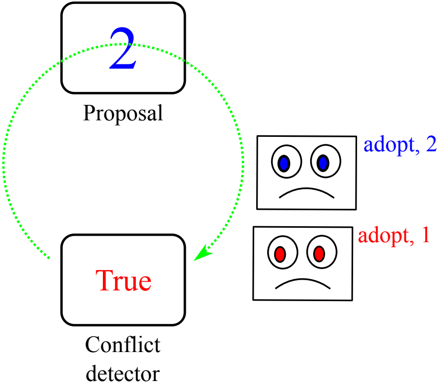
Adopt-commit from conflict detector

procedure `adoptCommit(`v`)`
begin
  conflict.`write(`v`)`
  $u \leftarrow$ proposal
  if $u = \perp$ **then**
    proposal $\leftarrow v$
  else
    $v \leftarrow u$
  end
  if conflict.`read()` $=$ **false**
  then
    return (commit, v)
  else
    return (adopt, v)
  end
end



Adopt-commit from conflict detector

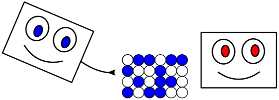
procedure `adoptCommit`(v)
begin
  conflict.`write`(v)
  $u \leftarrow$ proposal
  if $u = \perp$ **then**
    proposal $\leftarrow v$
  else
    $v \leftarrow u$
  end
  if conflict.`read`() $=$ **false**
  then
    return (commit, v)
  else
    return (adopt, v)
  end
end



Conflict detector using subsets

(Aspnes, PODC 2010)

- Assign unique write quorum W_v of k out of $2k$ registers to each value v, where $k = \Theta(\log m)$ satisfies $\binom{2k}{k} \geq m$.
- Write v by writing all registers in W_v.
- Check for $v' \neq v$ by reading all registers in $\overline{W}_v$.
- I always see you if you finish writing $W_{v'}$.

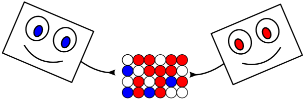


Cost: $\Theta(\log m)$ individual work and $\Theta(\log m)$ space.
Can we do better?

Conflict detector using subsets

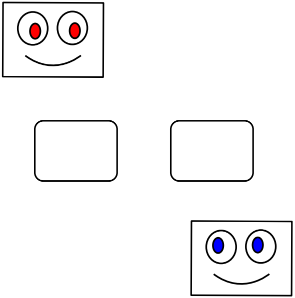
(Aspnes, PODC 2010)

- Assign unique write quorum W_v of k out of $2k$ registers to each value v, where $k = \Theta(\log m)$ satisfies $\binom{2k}{k} \geq m$.
- Write v by writing all registers in W_v.
- Check for $v' \neq v$ by reading all registers in $\overline{W}_v$.
- I always see you if you finish writing $W_{v'}$.



Cost: $\Theta(\log m)$ individual work and $\Theta(\log m)$ space.
Can we do better?

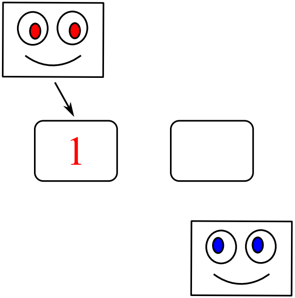
Conflict detector using permutations



With 2 values:

- Processes with 1 write r_1 then read r_2.
- Processes with 2 write r_2 then read r_1
- With a conflict, whoever writes last sees the other value.

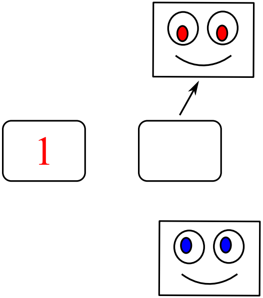
Conflict detector using permutations



With 2 values:

- Processes with 1 write r_1 then read r_2.
- Processes with 2 write r_2 then read r_1
- With a conflict, whoever writes last sees the other value.

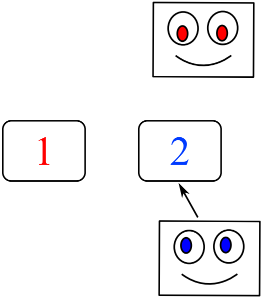
Conflict detector using permutations



With 2 values:

- Processes with 1 write r_1 then read r_2.
- Processes with 2 write r_2 then read r_1
- With a conflict, whoever writes last sees the other value.

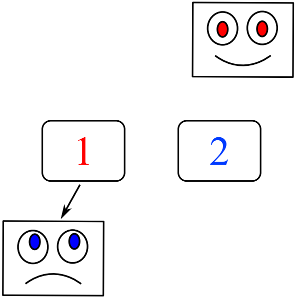
Conflict detector using permutations



With 2 values:

- Processes with 1 write r_1 then read r_2.
- Processes with 2 write r_2 then read r_1
- With a conflict, whoever writes last sees the other value.

Conflict detector using permutations



With 2 values:

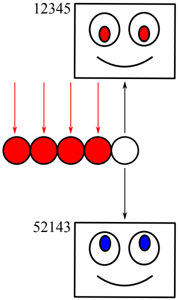
- Processes with 1 write r_1 then read r_2.
- Processes with 2 write r_2 then read r_1
- With a conflict, whoever writes last sees the other value.

Conflict detector using permutations

With m values:

- Use k registers with $k! \geq m$.
- Each value v gets a distinct permutation π_v.
- Processes execute the following code:

for i **in** π_v **do**
    $r \leftarrow r_i$
    if $r = \perp$ **then**
        $r_i \leftarrow v$
    else if $r \neq v$ **then**
        conflict $\leftarrow$ **true**
    end
end



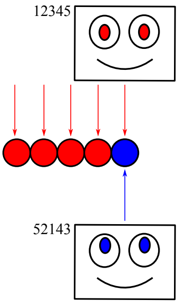
- Any distinct permutations invert some pair $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log \log m)$.

Conflict detector using permutations

With m values:

- Use k registers with $k! \geq m$.
- Each value v gets a distinct permutation π_v.
- Processes execute the following code:

  for i **in** π_v **do**
  $\quad r \leftarrow r_i$
  $\quad$ **if** $r = \perp$ **then**
  $\quad\quad r_i \leftarrow v$
  $\quad$ **else if** $r \neq v$ **then**
  $\quad\quad$ conflict $\leftarrow$ **true**
  $\quad$ **end**
  end



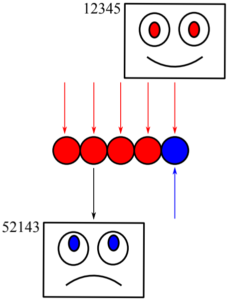
- Any distinct permutations invert some pair $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log\log m)$.

Conflict detector using permutations

With m values:

- Use k registers with $k! \geq m$.
- Each value v gets a distinct permutation π_v.
- Processes execute the following code:

for i **in** π_v **do**
  $r \leftarrow r_i$
  if $r = \perp$ **then**
    $r_i \leftarrow v$
  else if $r \neq v$ **then**
    conflict $\leftarrow$ **true**
  end
end



- Any distinct permutations invert some pair
  $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log\log m)$.

Upper bound result

We have reduced the cost of an m-valued adopt-commit from

$$\Theta(\log m)$$

to

$$\Theta(\log m / \log\log m).$$

This is not especially exciting on its own, but we also have a matching lower bound.

Lower bound: deterministic version

Theorem: Any anonymous deterministic conflict detector has an input that causes a process to take $\Omega(\log m / \log\log m)$ steps in a solo execution

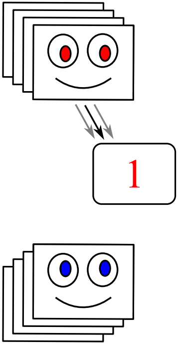
Proof outline:

1. For each input v, consider set of registers accessed in resulting solo execution E_v.
2. Define a permutation π_v of this set based on order of accesses.
3. If π_v and $\pi_{v'}$ agree on order of registers accessed in both E_v and $E_{v'}$, then there exists an execution where $v \neq v'$ conflict is not detected.
4. Avoiding this requires longest π_v to have at least $\Omega(\log m / \log\log m)$ elements.

Using clones to hide writes

We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

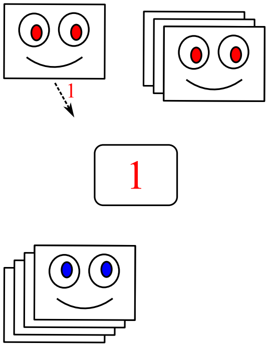
- Most clones do the same thing at the same time (they're anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution E_v into clone execution E_v^*.



Using clones to hide writes

We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

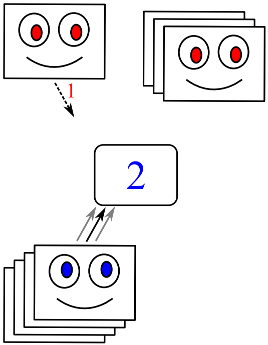
- Most clones do the same thing at the same time (they're anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution E_v into clone execution E_v^*.



Using clones to hide writes

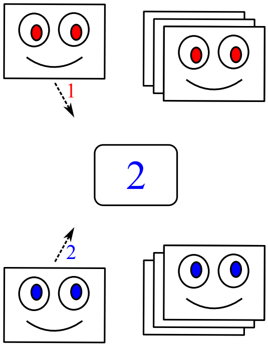
We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they're anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution E_v into clone execution E_v^*.



We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

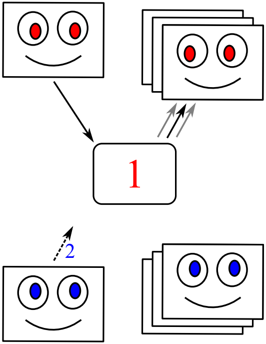
- Most clones do the same thing at the same time (they're anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution E_v into clone execution E_v^*.



Using clones to hide writes

We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they're anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution E_v into clone execution E_v^*.



First-write/last-read permutation

$$
\begin{array}{cccccccccc}
E_v & = & \mathbf{W1} & \mathrm{R2} & \mathrm{W1} & \mathrm{R3} & \mathbf{W2} & \mathrm{R1} & \mathbf{R3} \\
 & & \downarrow & & & & \downarrow & & \downarrow \\
\pi_v & = & 1 & & & & 2 & & 3
\end{array}
$$

- For each register r, pick the
  - First write to r if there is one, or
  - Last read from r otherwise.
- Let π_v list the registers in order of these operations.

Interleaved execution

E_v^*		**W1**		R2		W1 R3		**W2** R1 **R3**					
$E_{v'}^*$	R2		**W1**		R1		**R2**			**W4** (W1) R1			

- Interleave E_v^* and $E_{v'}^*$ according to $\pi_v \cup \pi_{v'}$ to make chosen operations on the same registers adjacent.
- Put last-reads before first-writes.
- Use delayed clones to rewrite registers before later reads.

Why the interleaving works

Restricting the view to a single register:

- If I *don't* write to r, my last read of r comes before your first write:

E_v^*		R2		**W2**
$E_{v'}^*$	R2		**R2**	

- If I *do* write to r, your first write happens at the same time as mine, so we can use cloned operations to mask it (and any subsequent writes):

E_v^*	**W1**			W1	R1		
$E_{v'}^*$		**W1**	R1			(W1)	R1

$\Rightarrow$ Conflict detector doesn't work unless π_v and $\pi_{v'}$ are inconsistent for all $v \neq v'$.

Combinatorial lemma

Claim: Any family of pairwise-inconsistent partial permutations $\{\pi_v\}$ satisfies

$$\sum_v \frac{1}{|\pi_v|!} \leq 1.$$

Proof:

1. Pick a random ordering of all registers.
2. Let A_v be the event that π_v is increasing in this ordering.
3. $\Pr[A_v] = \frac{1}{|\pi_v|!}$.
4. Observe that if π_v and $\pi_{v'}$ are inconsistent, $A_v \cap A_{v'} = \emptyset$.
5. $\Rightarrow \sum \Pr[A_v] = \Pr\left[\bigcup A_v\right] \leq 1$.

Corollary: Pigeonhole argument gives $\frac{1}{|\pi_v|!} \leq \frac{1}{m}$ for some v, which gives $\max_v |\pi_v| = \Omega(\log m / \log\log m)$.

Lower bound: randomized version

For a randomized conflict detector:

1. Define E_v to be shortest solo execution that occurs with nonzero probability for input v.
2. Repeat same analysis as for deterministic executions.
3. If we can interleave E_v^* and $E_{v'}^*$, there is a (small) nonzero probability that every clone flips its coins the right way, violating the spec.

So lower bound applies with probability 1 to solo executions of randomized algorithms as well.

Dependence on n

Let n be the number of processes.

- Interleaving consumes $O(1)$ clones per step.
- $\Rightarrow$ lower bound can't exceed $\Omega(n)$.
- Can also get $O(n)$ upper bound.
- So real bound is:

$$\Theta\left(\min\left(\frac{\log m}{\log\log m}, n\right)\right)$$

Same lower bound applies for anonymous m-valued consensus.

Open problem

Does $\Theta\left(\min\left(\frac{\log m}{\log\log m}, n\right)\right)$ bound hold without anonymity?

Progress so far (not in proceedings version):

- Lower bound:

$$\Omega\left(\min\left(\frac{\log m}{\log\log m}, \frac{\sqrt{\log n}}{\log\log n}\right)\right)$$

  for deterministic implementations.

- Upper bound:

$$O\left(\min\left(\frac{\log m}{\log\log m}, \log n\right)\right).$$