SLR 206 – Fondements des Algorithmes Répartis

Randomized Consensus and Other Agreement Protocols

Luciano Freitas   Petr Kuznetsov

January 25th 2023

So far...

- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.

So far...

- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.
- Consensus is impossible to solve deterministically in an asynchronous message passing system if at least one process may crash.

Next

- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.
- Consensus is impossible to solve deterministically in an asynchronous message passing system if at least one process may crash.

These constructions have *Consensus number 1*

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
⋮	⋮
$2n-2$	n-register assignment
⋮	⋮
∞	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

Figure: Maurice Herlihy – Wait-free synchronization.

Next

- Consensus is impossible to solve deterministically in an asynchronous message passing system if at least one process may crash.

Consensus can be deterministically solved in *partial-synchronous* and *synchronous* networks.

- Asynchronous – No bound on message delays.
- Partial-synchronous – There exists a delay, but it is unknown.
- Synchronous – There exists a known delay.

Next

- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.
- Consensus is impossible to solve deterministically in an asynchronous message passing system if at least one process may crash.

In this lecture: There are other agreement protocols.

- Approximate Agreement
- Commit-adopt

Next

- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.

In this lecture: Consensus can be deterministically solved in an *obstruction-free* run.

- Lock-free – If processes try infinitely many times to take steps, they eventually finish.
- Obstruction-free – If a process takes steps in isolation, then it finishes in finitely many steps.
- Wait-free – Every process finishes in finitely many steps.

Next

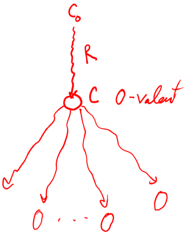
- Consensus is impossible to solve deterministically in a wait-free manner using read-write registers if at least one process may crash.
- Consensus is impossible to solve deterministically in an asynchronous message passing system if at least one process may crash.

In this lecture: Consensus can be solved by randomized protocols with probabilistic guarantees.

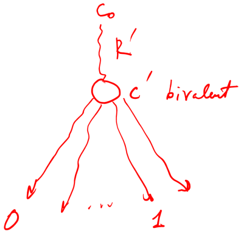
Table of Contents

Table of Contents

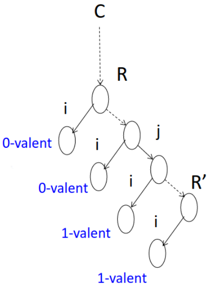
Review – FLP



C0
R
C
0-valent
0 ... 0
0



C0
R'
C'
bivalent
0
1



C
R
i
j
0-valent
i
0-valent
i
R'
1-valent
i
1-valent

The iterated memory model

Algorithm 1 Two processes iterated memory – Code for process i

Infinite array of n atomic registers $R_k[1\ldots]$

$k \leftarrow 0$

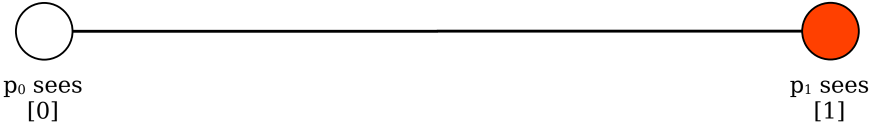
$v_i \leftarrow$ input

while decision not reached **do**

  $k \leftarrow k + 1$

  $R_k[i]$.write(v_i)

  $v_i \leftarrow [v_i, R_k[i-1]$.read()$]$

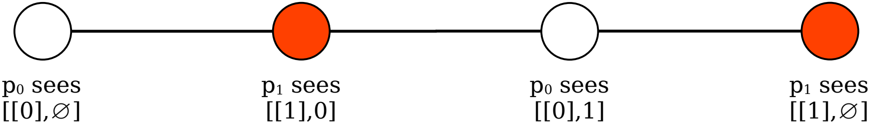


The iterated memory model

Algorithm 2 Two processes iterated memory – Code for process i

Infinite array of n atomic registers $R_k[1\ldots]$

$k \leftarrow 0$
$v_i \leftarrow$ input
while decision not reached **do**
    $k \leftarrow k+1$ $//k=1$
    $R_k[i]$.write(v_i)
    $v_i \leftarrow [v_i, R_k[i-1]$.read$()]$



The iterated memory model

Algorithm 3 Two processes iterated memory – Code for process i

Infinite array of n atomic registers $R_k[1\ldots]$

$k \leftarrow 0$

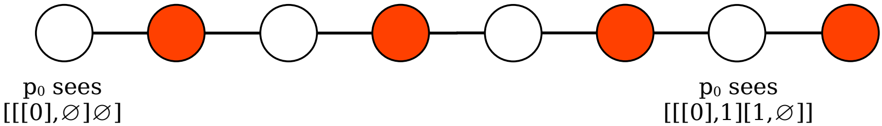
$v_i \leftarrow$ input

while decision not reached **do**

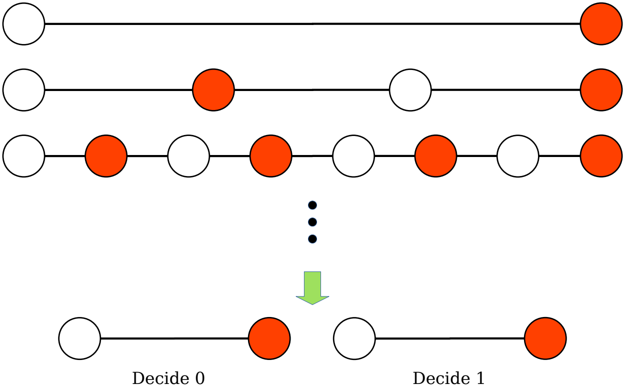
  $k \leftarrow k + 1$ $//k = 2$

  $R_k[i]$.write(v_i)

  $v_i \leftarrow [v_i, R_k[i-1]$.read$()]$



The topological argument for FLP



The topological argument for FLP

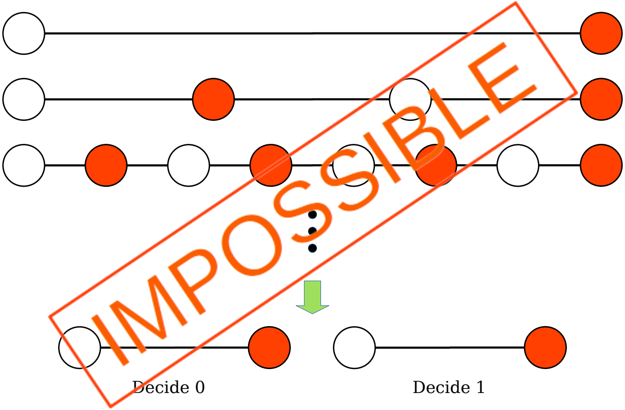


Table of Contents

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations?

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Bad examples	Good examples

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Bad examples	Good examples
input: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 0.01$ **output**: $[1\ 1\ 0.99\ 1\ 1]$	

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Bad examples	Good examples
input: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 0.01$ **output**: $[1\ 1\ 0.99\ 1\ 1]$	**input**: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 10000$ **output**: $[1\ 1\ 1\ 1\ 1]$

Approximate Agreement

An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Bad examples	Good examples
input: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 0.01$ **output**: $[1\ 1\ 0.99\ 1\ 1]$	**input**: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 10000$ **output**: $[1\ 1\ 1\ 1\ 1]$
input: $v = [0\ 2\ 3\ 9\ 7\ 3], \epsilon = 0.5$ **output**: $[9.3\ 9\ 8.9\ 8.95\ 1]$	

Approximate Agreement

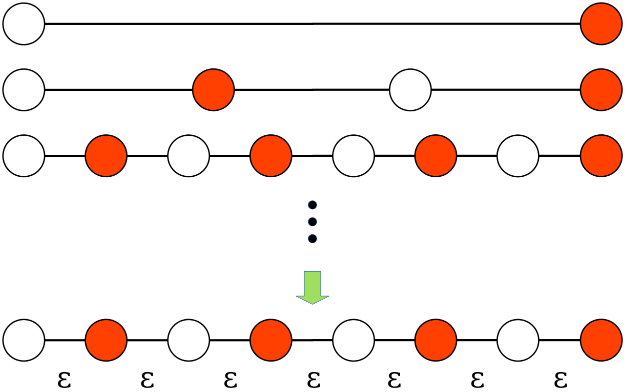
An approximate agreement protocol provides the following properties:

- Validity – Every output value is in the range of input values.
- Agreement – The output value of different processes are at most ϵ apart.
- Liveness – Eventually ever correct process decides.

Limitations? The problem must be "continuous" and admit disagreement.

Bad examples	Good examples
input: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 0.01$ **output**: [1 1 0.99 1 1]	**input**: $v = [1\ 1\ 1\ 1\ 1\ 1], \epsilon = 10000$ **output**: [1 1 1 1 1]
input: $v = [0\ 2\ 3\ 9\ 7\ 3], \epsilon = 0.5$ **output**: [9.3 9 8.9 8.95 1]	**input**: $v = [0\ 2\ 3\ 9\ 7\ 3], \epsilon = 0.5$ **output**: [9 9 8.9 8.95 1]

The topology of AA



Solving AA

Algorithm 4 Two process wait-free ϵ-AA

Given two atomic registers $R[0], R[1]$

$v_i \leftarrow$ input

while true **do**

  // R[i] is originally $\perp$

  $R[i]$.write(v_i)

  $v_j \leftarrow R[1-i]$.read(v_i)

  if $v_j = \perp \vee |v_j - v_i| \leq \epsilon$ **then** Decide v_i

  else $v_i \leftarrow \frac{v_i + v_j}{2}$

Solving AA

Algorithm 5 Two process wait-free ϵ-AA

1: Given two atomic registers $R[0], R[1]$
2: $v_i \leftarrow$ input
3: **while** true **do**
4: // R[i] is originally $\perp$
5: $R[i].\mathrm{write}(v_i)$
6: $v_j \leftarrow R[1-i].\mathrm{read}(v_i)$
7: **if** $v_j = \perp \vee |v_j - v_i| \leq \epsilon$ **then** Decide v_i
8: **else** $v_i \leftarrow \frac{v_i+v_j}{2}$

Solving AA

Algorithm 6 Two process wait-free ϵ-AA

Given two atomic registers $R[0], R[1]$

$v_i \leftarrow$ input

while true **do**

  // R[i] is originally $\perp$

  $R[i]$.write(v_i)

  $v_j \leftarrow R[1-i]$.read(v_i)

  if $v_j = \perp \vee |v_j - v_i| \leq \epsilon$ **then** Decide v_i

  else $v_i \leftarrow \frac{v_i + v_j}{2}$

Solving AA

Algorithm 7 Two process wait-free ϵ-AA

```
Given two atomic registers R[0], R[1]
v_i ← input
while true do
    // R[i] is originally ⊥
    R[i].write(v_i)
    v_j ← R[1 − i].read(v_i)
    if v_j = ⊥ ∨ |v_j − v_i| ≤ ε then Decide v_i
    else v_i ← (v_i + v_j) / 2
```

Solving AA

Algorithm 8 Two process wait-free ϵ-AA

Given two atomic registers $R[0], R[1]$

$v_i \leftarrow$ input

while true **do**

    // R[i] is originally $\perp$

    $R[i]$.write(v_i)

    $v_j \leftarrow R[1-i]$.read(v_i)

    if $v_j = \perp \vee |v_j - v_i| \leq \epsilon$ **then** Decide v_i

    else $v_i \leftarrow \frac{v_i+v_j}{2}$

Solution Analysis

- Validity – The decided values are either v_i or some iteration of $\frac{v_i + v_j}{2}$.

Solution Analysis

- Validity – The decided values are either v_i or some iteration of $\frac{v_i+v_j}{2}$.
- Agreement – If a process decides by reading $\perp$, because processes write before they read, process $1-i$ will for certain begin looping. If both processes loop, they will converge to the same value.

Solution Analysis

- Validity – The decided values are either v_i or some iteration of $\frac{v_i + v_j}{2}$.
- Agreement – If a process decides by reading $\perp$, because processes write before they read, process $1 - i$ will for certain begin looping. If both processes loop, they will converge to the same value.
- Liveness – The algorithm is wait-free, a process does not need the other to take steps in order to make progress and always terminates.

The algorithm terminates in at most

Solution Analysis

- Validity – The decided values are either v_i or some iteration of $\frac{v_i+v_j}{2}$.
- Agreement – If a process decides by reading $\perp$, because processes write before they read, process $1-i$ will for certain begin looping. If both processes loop, they will converge to the same value.
- Liveness – The algorithm is wait-free, a process does not need the other to take steps in order to make progress and always terminates.

The algorithm terminates in at most $O(\log(\frac{|v_1-v_0|}{\epsilon}))$ steps.

AA with n processes

Simply extending the case for $n = 2$ by using snapshot objects results in a $O((n + \log n)\log(\frac{|v_1 - v_0|}{\epsilon}))$ complexity algorithm. In "Are wait-free algorithms fast?" by Attiya, Lynch and Shavit, the authors provide a wait-free solution with complexity $O(\log n)$.

In message passing:

1. Broadcast v
2. Wait for n-f values
3. Compute mean of received values
4. Repeat

The problem is more interesting when there are Byzantine failures.

Table of Contents

Commit Adopt Definition

In commit adopt, processes propose an input value from V and output a couple $(c, v) \in \{false, true\} \times V$. If a process outputs either $(false, v)$ or $(true, v)$, we say that it adopts v. If a process outputs $(true, v)$, then we say that it commits to v.

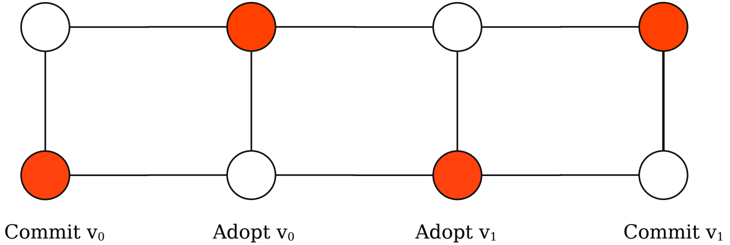
- Validity – Every adopted value is an input value.
- Termination – Every correct process eventually adopts a value.
- CA-Agreement
  - If a process commits a value v, then every correct process adopts v.
  - If every process inputs the same value, then every correct process commits to a value.

Graded Commit Adopt Definition

In graded commit adopt, processes propose an input value from V and output a couple $(c, v) \in \{0, 1, 2\} \times V$. If a process outputs either $(1, v)$ or $(2, v)$, we say that it adopts v. If a process outputs $(2, v)$, then we say that it commits to v, finally, if a process outputs $(0, v)$ we say that it sticks with its input.

- Validity – Every output value is an input value.
- Termination – Every correct process eventually outputs a value.
- CA-Agreement
  - If a process commits a value v, then every correct process adopts v.
  - If every process inputs the same value, then every correct process commits to a value.

Commit Adopt Topology



Graded Commit Adopt Algorithm

Algorithm 9 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are $(true, v_i)$ **then** Decide $(2, v_i)$
else if $\exists(true, v') \in U$ **then** Decide $(1, v')$
else Decide$(0, v_i)$

Graded Commit Adopt Algorithm

Algorithm 10 Graded Commit Adopt (GCA)

```
A[1..n] n atomic registers
B[1..n] n atomic registers
v_i ← v    // v is the original input
A[i].write(v_i)
U ← read A[1..n]
if All non-⊥ values in U are v_i then B[i].write((true, v_i))
else B[i].write((false, v_i))
U ← read B[1..n]
if All non-⊥ values in U are (true, v_i) then Decide (2, v_i)
else if ∃(true, v′) ∈ U then Decide (1, v′)
else Decide(0, v_i)
```

Graded Commit Adopt Algorithm

Algorithm 11 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists$($true$, v') $\in U$ **then** Decide (1, v')
else Decide(0, v_i)

Graded Commit Adopt Algorithm

Algorithm 12 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are $(true, v_i)$ **then** Decide $(2, v_i)$
else if $\exists(true, v') \in U$ **then** Decide $(1, v')$
else Decide$(0, v_i)$

Graded Commit Adopt Algorithm

Algorithm 13 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists(true, v') \in U$ **then** Decide (1, v')
else Decide(0, v_i)

Graded Commit Adopt Algorithm

Algorithm 14 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers

$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))

$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists(true, v') \in U$ **then** Decide (1, v')
else Decide(0, v_i)

Graded Commit Adopt Algorithm

Algorithm 15 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists$($true$, v') $\in U$ **then** Decide (1, v')
else Decide(0, v_i)

Graded Commit Adopt Algorithm

Algorithm 16 Graded Commit Adopt (GCA)

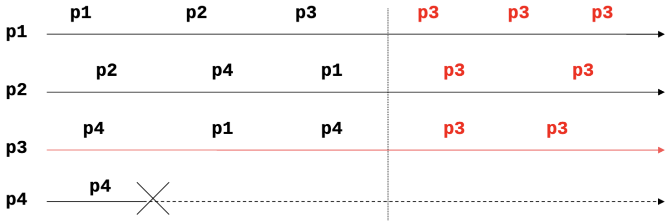
$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists(true, v') \in U$ **then** Decide $(1, v')$
else Decide(0, v_i)

Algorithm 17 Graded Commit Adopt (GCA)

$A[1..n]$ n atomic registers
$B[1..n]$ n atomic registers
$v_i \leftarrow v$ // v is the original input
$A[i]$.write(v_i)
$U \leftarrow$ read $A[1..n]$
if All non-$\perp$ values in U are v_i **then** $B[i]$.write(($true$, v_i))
else $B[i]$.write(($false$, v_i))
$U \leftarrow$ read $B[1..n]$
if All non-$\perp$ values in U are ($true$, v_i) **then** Decide (2, v_i)
else if $\exists(true, v') \in U$ **then** Decide (1, v')
else Decide(0, v_i)

Ω failure detector

- Eventual Leader Detector.
- Outputs at every process a process identifier.
- Eventually, the same correct process is output at every correct process.



It can be implemented in partial-synchronous systems.

Consensus

Algorithm 18 GCA + Ω = Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
    $k \leftarrow k + 1$
    if $v' \leftarrow D$.read() $\neq \perp$ **then** Decide v'
    $(c, v_i) \leftarrow GCA_k$.propose$(v_i)$
    if $c = 0 \wedge R[\Omega]$.read() $\neq \perp$ **then** $v_i \leftarrow R[\Omega]$.read()
        // $c = 1$ the process adopts the value from GCA
    else if $c = 2$ **then** D.write(v_i)

Consensus

Algorithm 19 GCA $+ \Omega =$ Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
  $k \leftarrow k + 1$
  if $v' \leftarrow D.\text{read}() \neq \perp$ **then** Decide v'
  $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
  if $c = 0 \wedge R[\Omega].\text{read}() \neq \perp$ **then** $v_i \leftarrow R[\Omega].\text{read}()$
    // $c = 1$ the process adopts the value from GCA
  else if $c = 2$ **then** $D.\text{write}(v_i)$

Consensus

Algorithm 20 GCA $+ \Omega =$ Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
    $k \leftarrow k + 1$
    if $v' \leftarrow D$.read() $\neq \perp$ **then** Decide v'
    $(c, v_i) \leftarrow GCA_k$.propose$(v_i)$
    if $c = 0 \wedge R[\Omega]$.read() $\neq \perp$ **then** $v_i \leftarrow R[\Omega]$.read()
        // $c = 1$ the process adopts the value from GCA
    else if $c = 2$ **then** D.write(v_i)

Consensus

Algorithm 21 GCA + Ω = Consensus

```
D a regular register
R[1..n] n regular registers
GCA_1, GCA_2, ... a series of graded commit adopt instances
Ω failure detector

v_i ← v   // v is the original input
k ← 0
R[i] ← v
while decision not reached do
    k ← k + 1
    if v' ← D.read() ≠ ⊥ then Decide v'
    (c, v_i) ← GCA_k.propose(v_i)
    if c = 0 ∧ R[Ω].read() ≠ ⊥ then v_i ← R[Ω].read()
        // c = 1 the process adopts the value from GCA
    else if c = 2 then D.write(v_i)
```

Consensus

Algorithm 22 GCA + Ω = Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
  $k \leftarrow k + 1$
  if $v' \leftarrow D.\text{read}() \neq \perp$ **then** Decide v'
  $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
  if $c = 0 \wedge R[\Omega].\text{read}() \neq \perp$ **then** $v_i \leftarrow R[\Omega].\text{read}()$
    // $c = 1$ the process adopts the value from GCA
  else if $c = 2$ **then** $D.\text{write}(v_i)$

Consensus

Algorithm 23 GCA + Ω = Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
    $k \leftarrow k + 1$
    if $v' \leftarrow D.\text{read}() \neq \bot$ **then** Decide v'
    $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
    if $c = 0 \wedge R[\Omega].\text{read}() \neq \bot$ **then** $v_i \leftarrow R[\Omega].\text{read}()$
        // $c = 1$ the process adopts the value from GCA
    else if $c = 2$ **then** $D.\text{write}(v_i)$

Consensus

Algorithm 24 GCA + Ω = Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
    $k \leftarrow k + 1$
    if $v' \leftarrow D.\text{read}() \neq \bot$ **then** Decide v'
    $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
    if $c = 0 \wedge R[\Omega].\text{read}() \neq \bot$ **then** $v_i \leftarrow R[\Omega].\text{read}()$
        // $c = 1$ the process adopts the value from GCA
    else if $c = 2$ **then** $D.\text{write}(v_i)$

Consensus

Algorithm 25 GCA $+ \Omega =$ Consensus

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
Ω failure detector

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
  $k \leftarrow k + 1$
  if $v' \leftarrow D.\text{read}() \neq \perp$ **then** Decide v'
  $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
  if $c = 0 \wedge R[\Omega].\text{read}() \neq \perp$ **then** $v_i \leftarrow R[\Omega].\text{read}()$
    // $c = 1$ the process adopts the value from GCA
  else if $c = 2$ **then** $D.\text{write}(v_i)$

Table of Contents

Modified Ben-Or Consensus Protocol in Shared Memory

Algorithm 26 Ben-Or in shared memory

D a regular register
$R[1..n]$ n regular registers
$GCA_1, GCA_2, \ldots$ a series of graded commit adopt instances
$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
$R[i] \leftarrow v$
while decision not reached **do**
    $k \leftarrow k + 1$
    if $v' \leftarrow D.\text{read}() \neq \perp$ **then** Decide v'
        $(c, v_i) \leftarrow GCA_k.\text{propose}(v_i)$
    if $c = 0$ **then** $v_i \leftarrow$ random bit
        // $c = 1$ the process adopts the value from GCA
    else if $c = 2$ **then** $D.\text{write}(v_i)$

Ben-Or Consensus

Algorithm 27 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
  $k \leftarrow k + 1$
  Send $\langle m_1, k, v_i \rangle$ to all
  Wait for $n - f$ messages $\langle m_1, k, * \rangle$
  if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
  else send $\langle m_2, k, \perp \rangle$
  Wait for $n - f$ messages $\langle m_2, k, * \rangle$
  // $n - f = f + 1$
  if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
  else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
  else $v_i \leftarrow$ Random bit

Algorithm 28 Ben-Or in message passing

```
v_i ← v    // v is the original input
k ← 0
while decision not reached do
    k ← k + 1
    Send ⟨m_1, k, v_i⟩ to all
    Wait for n − f messages ⟨m_1, k, ∗⟩
    if All msgs are ⟨m_1, k, v⟩ then  send ⟨m_2, k, v⟩
    else  send ⟨m_2, k, ⊥⟩

    Wait for n − f messages ⟨m_2, k, ∗⟩
    // n − f = f + 1
    if All msgs are ⟨m_2, k, v⟩ where v ≠ ⊥ then  send ⟨DECIDE, k, v⟩
    else if ∃⟨m_2, k, v⟩ where v ≠ ⊥ then  v_i ← v
    else  v_i ← Random bit
```

Ben-Or Consensus

Algorithm 29 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
  $k \leftarrow k + 1$
  Send $\langle m_1, k, v_i \rangle$ to all
  Wait for $n - f$ messages $\langle m_1, k, * \rangle$
  if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
  else send $\langle m_2, k, \perp \rangle$

  Wait for $n - f$ messages $\langle m_2, k, * \rangle$
  // $n - f = f + 1$
  if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
  else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
  else $v_i \leftarrow$ Random bit

Ben-Or Consensus

Algorithm 30 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
    $k \leftarrow k + 1$
    Send $\langle m_1, k, v_i \rangle$ to all
    Wait for $n - f$ messages $\langle m_1, k, * \rangle$
    if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
    else send $\langle m_2, k, \perp \rangle$

    Wait for $n - f$ messages $\langle m_2, k, * \rangle$
    // $n - f = f + 1$
    if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
    else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
    else $v_i \leftarrow$ Random bit

Ben-Or Consensus

Algorithm 31 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
  $k \leftarrow k + 1$
  Send $\langle m_1, k, v_i \rangle$ to all
  Wait for $n - f$ messages $\langle m_1, k, * \rangle$
  if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
  else send $\langle m_2, k, \perp \rangle$
  Wait for $n - f$ messages $\langle m_2, k, * \rangle$
  // $n - f = f + 1$
  if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
  else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
  else $v_i \leftarrow$ Random bit

Ben-Or Consensus

Algorithm 32 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
    $k \leftarrow k + 1$
    Send $\langle m_1, k, v_i \rangle$ to all
    Wait for $n - f$ messages $\langle m_1, k, * \rangle$
    if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
    else send $\langle m_2, k, \perp \rangle$

    Wait for $n - f$ messages $\langle m_2, k, * \rangle$
    // $n - f = f + 1$
    if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
    else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
    else $v_i \leftarrow$ Random bit

Ben-Or Consensus

Algorithm 33 Ben-Or in message passing

```
v_i ← v    // v is the original input
k ← 0
while decision not reached do
    k ← k + 1
    Send ⟨m_1, k, v_i⟩ to all
    Wait for n − f messages ⟨m_1, k, ∗⟩
    if All msgs are ⟨m_1, k, v⟩ then  send ⟨m_2, k, v⟩
    else  send ⟨m_2, k, ⊥⟩

    Wait for n − f messages ⟨m_2, k, ∗⟩
    // n − f = f + 1
    if All msgs are ⟨m_2, k, v⟩ where v ≠ ⊥ then  send ⟨DECIDE, k, v⟩
    else if ∃⟨m_2, k, v⟩ where v ≠ ⊥ then  v_i ← v
    else  v_i ← Random bit
```

Ben-Or Consensus

Algorithm 34 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
    $k \leftarrow k + 1$
    Send $\langle m_1, k, v_i \rangle$ to all
    Wait for $n - f$ messages $\langle m_1, k, * \rangle$
    if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
    else send $\langle m_2, k, \perp \rangle$

    Wait for $n - f$ messages $\langle m_2, k, * \rangle$
    // $n - f = f + 1$
    if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
    else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
    else $v_i \leftarrow$ Random bit

Ben-Or Consensus

Algorithm 35 Ben-Or in message passing

$v_i \leftarrow v$ // v is the original input
$k \leftarrow 0$
while decision not reached **do**
  $k \leftarrow k + 1$
  Send $\langle m_1, k, v_i \rangle$ to all
  Wait for $n - f$ messages $\langle m_1, k, * \rangle$
  if All msgs are $\langle m_1, k, v \rangle$ **then** send $\langle m_2, k, v \rangle$
  else send $\langle m_2, k, \perp \rangle$

  Wait for $n - f$ messages $\langle m_2, k, * \rangle$
  // $n - f = f + 1$
  if All msgs are $\langle m_2, k, v \rangle$ where $v \neq \perp$ **then** send $\langle DECIDE, k, v \rangle$
  else if $\exists \langle m_2, k, v \rangle$ where $v \neq \perp$ **then** $v_i \leftarrow v$
  else $v_i \leftarrow$ Random bit

Complexity

The expected number of rounds is $\mathbb{E}[R] = \sum_{r=1}^{\infty}(r \times P(\text{Decide at round r}))$.

Complexity

The expected number of rounds is $\mathbb{E}[R] = \sum_{r=1}^{\infty}(r \times P(\text{Decide at round r}))$.

Let p be the probability that all processes get the same value.

Complexity

The expected number of rounds is $\mathbb{E}[R] = \sum_{r=1}^{\infty}(r \times P(\text{Decide at round r}))$.

Let p be the probability that all processes get the same value.

$P(\text{Decide at round r}) = (1-p)^{r-1}p$.

Complexity

The expected number of rounds is $\mathbb{E}[R] = \sum_{r=1}^{\infty}(r \times P(\text{Decide at round r}))$.

Let p be the probability that all processes get the same value.

$P(\text{Decide at round r}) = (1-p)^{r-1}p$.

$\mathbb{E}[R] = \sum_{r=1}^{\infty}(r(1-p)^{r-1}p) = \frac{1}{p}$.

Complexity

The expected number of rounds is $\mathbb{E}[R] = \sum_{r=1}^{\infty}(r \times P(\text{Decide at round r}))$.

Let p be the probability that all processes get the same value.

$P(\text{Decide at round r}) = (1-p)^{r-1}p$.

$\mathbb{E}[R] = \sum_{r=1}^{\infty}(r(1-p)^{r-1}p) = \frac{1}{p}$.

The probability of a decision being reached at every round as a result of all flips matching is: $p = 2^{-n}$. Hence, the number of expected rounds is $O(2^n)$.

Shared Memory Common Coin

Algorithm 36 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers

$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
  $count \leftarrow count + 1$
  $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
  $R[i].\text{write}((count, sum))$
  if $count \mod n = 0$ **then**
    $U \leftarrow \text{Read } R[1..n]$
    if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
      $done \leftarrow true$
      Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 37 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers
$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
    $count \leftarrow count + 1$
    $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
    $R[i].\text{write}((count, sum))$
    if $count \mod n = 0$ **then**
        $U \leftarrow \text{Read } R[1..n]$
        if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
            $done \leftarrow true$
            Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 38 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers
$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
  $count \leftarrow count + 1$
  $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
  $R[i].\text{write}((count, sum))$
  if $count \mod n = 0$ **then**
    $U \leftarrow \text{Read } R[1..n]$
    if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
      $done \leftarrow true$
      Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 39 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers

$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
  $count \leftarrow count + 1$
  $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
  $R[i].\text{write}((count, sum))$
  if $count \mod n = 0$ **then**
    $U \leftarrow \text{Read } R[1..n]$
    if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
      $done \leftarrow true$
      Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 40 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers

$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
    $count \leftarrow count + 1$
    $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
    $R[i]\text{.write}((count, sum))$
    if $count \mod n = 0$ **then**
        $U \leftarrow \text{Read } R[1..n]$
        if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
            $done \leftarrow true$
            Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 41 Bracha-Rachman common coin

done – shared atomic register
$R[1..n]$ – n atomic registers

$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
    $count \leftarrow count + 1$
    $sum \leftarrow sum +$ flip()   // The flip result is -1 or +1
    $R[i]$.write($(count, sum)$)
    if $count \mod n = 0$ **then**
        $U \leftarrow$ Read $R[1..n]$
        if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
            $done \leftarrow true$
            Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin

Algorithm 42 Bracha-Rachman common coin

$done$ – shared atomic register
$R[1..n]$ – n atomic registers
$count \leftarrow 0$
$sum \leftarrow 0$
while not done **do**
    $count \leftarrow count + 1$
    $sum \leftarrow sum + \text{flip}()$ // The flip result is -1 or +1
    $R[i].\text{write}((count, sum))$
    if $count \mod n = 0$ **then**
        $U \leftarrow \text{Read } R[1..n]$
        if $\sum_{i=1}^{n} U[i].count \geq n^2$ **then**
            $done \leftarrow true$
            Decide $sgn(\sum_{i=1}^{n} U[i].sum)$

Shared Memory Common Coin – Analysis

The summation of n^2 can be approximated to a normal random function.

The summation of n^2 can be approximated to a normal random function.

The probability that $sum \geq cn$ can be lowerbounded by a constant.

Shared Memory Common Coin – Analysis

The summation of n^2 can be approximated to a normal random function.

The probability that $sum \geq cn$ can be lowerbounded by a constant.

The probability that $count$ exceeds n^2 by more than $O(n)$ can be bounded by a constant.

Shared Memory Common Coin – Analysis

The summation of n^2 can be approximated to a normal random function.

The probability that $sum \geq cn$ can be lowerbounded by a constant.

The probability that $count$ exceeds n^2 by more than $O(n)$ can be bounded by a constant.

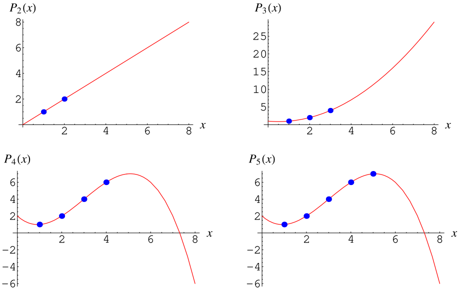
The probability that processes agree is therefore constant.

Message Passing Common Coin

Digital Signatures:

- Unforgeability – Processes with not enough keys cannot produce a signature.
- Uniqueness – Processes with enough keys always generate the same signature for the same message.

How to achieve it? – Polynomial interpolation.



These coins have constant success probability $\rightarrow$ randomized consensus takes a constant amount of times in expectation.

Good Luck with the Exam!