An introduction to Mobile Robotics Seen through the eyes of distributed computing

Eric Goubault and Bernardo Hummes

June 15, 2022

Part I: An introduction to Distributed Task solvability

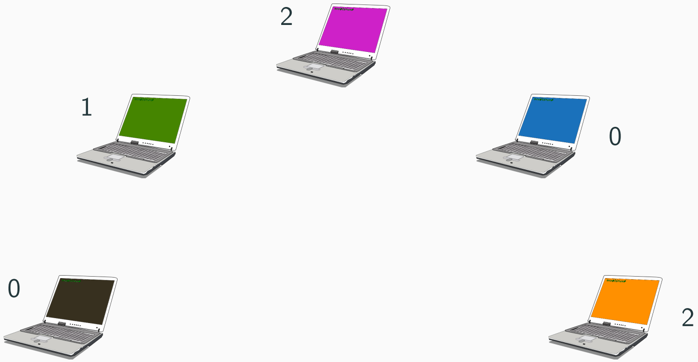
(slightly biased towards geometric methods though)

Based on slides from Jérémy Ledent

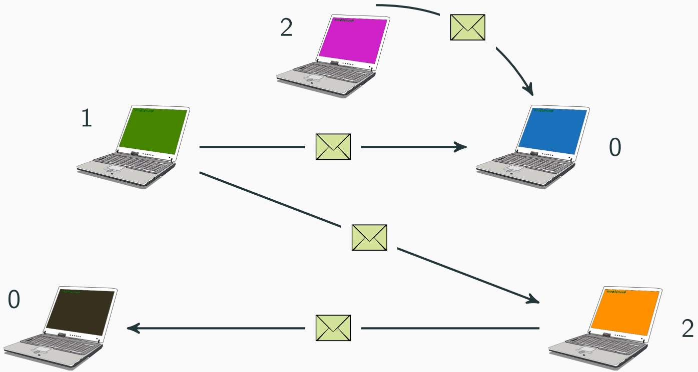
Introduction

The distributed computing setting

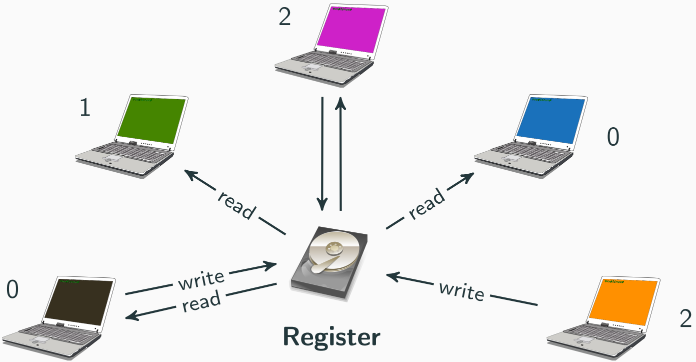
The distributed computing setting



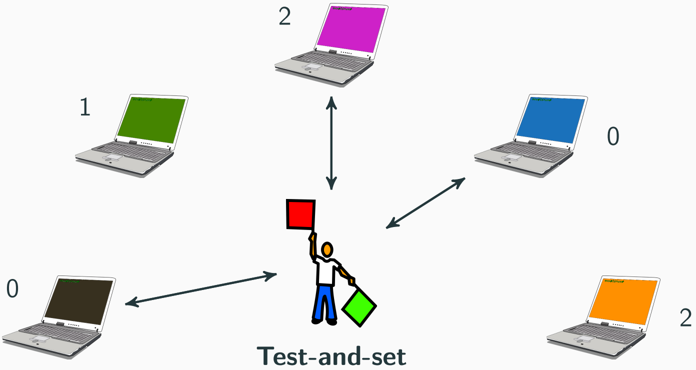
The distributed computing setting



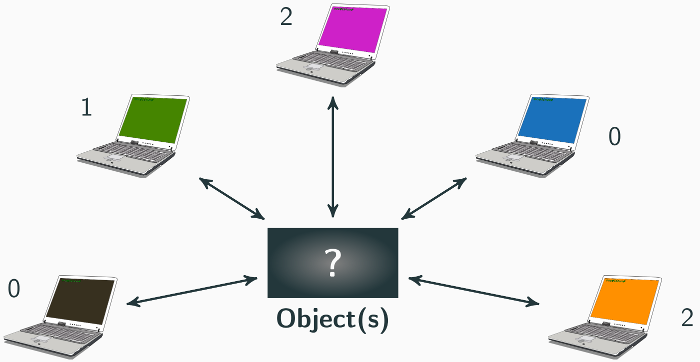
The distributed computing setting



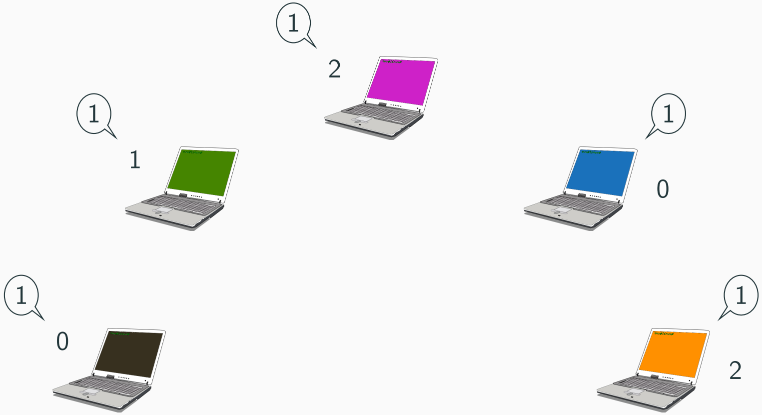
The distributed computing setting



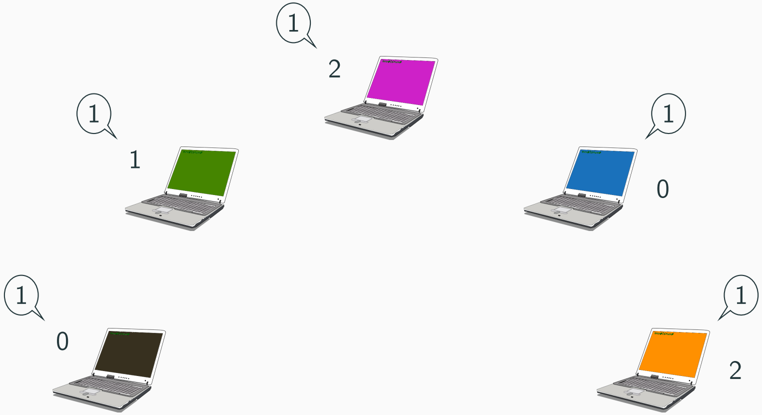
The distributed computing setting



The distributed computing setting



The distributed computing setting



Task specification: $(0,1,2,0,2) \rightarrow (1,1,1,1,1)$ ✓ or ✗ ?

Question: Can we solve the task T using the objects $X_1,\ldots,X_n$?

Asynchronous computability (a.k.a. Fault-tolerant distributed computing)

Question: Can we solve the task T using the objects $X_1,\ldots,X_n$?

- Compare the strength of objects.

Asynchronous computability (a.k.a. Fault-tolerant distributed computing)

Question: Can we solve the task T using the objects $X_1, \ldots, X_n$?

- Compare the strength of objects.
- Compare the difficulty of solving tasks.

Asynchronous computability (a.k.a. Fault-tolerant distributed computing)

Question: Can we solve the task T using the objects $X_1,\ldots,X_n$?

- Compare the strength of objects.
- Compare the difficulty of solving tasks.
- Usually, study objects in which failures can occur.

Asynchronous computability (a.k.a. Fault-tolerant distributed computing)

Question: Can we solve the task T using the objects $X_1, \ldots, X_n$?

- Compare the strength of objects.
- Compare the difficulty of solving tasks.
- Usually, study objects in which failures can occur.

Main focus today: Proving impossibility results.

⚠ Usually *not* a matter of computing power:
Dealing with incomplete or unreliable information.

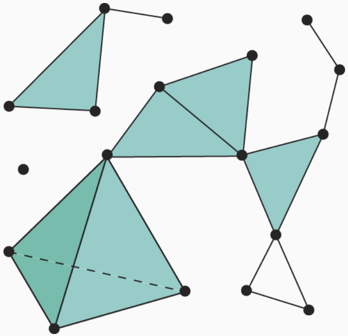
Combinatorial Topology

Simplicial complexes

Definition

A **simplicial complex** is a pair $\langle V, S \rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)

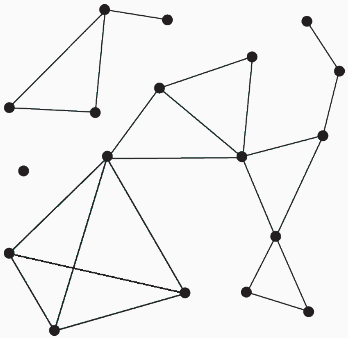


Simplicial complexes

Definition

A **simplicial complex** is a pair $\langle V, S\rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)

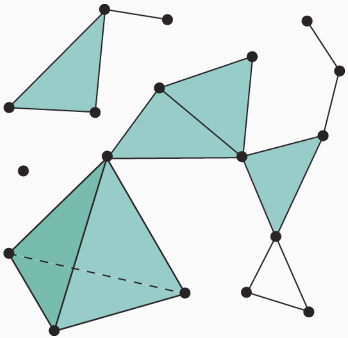


Simplicial complexes

Definition

A **simplicial complex** is a pair $\langle V, S\rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)



Simplicial complexes

Definition

A **simplicial complex** is a pair $\langle V, S \rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)

Definition

- If $X, Y \in S$ are simplices such that $Y \subseteq X$, then Y is a face of X.

Simplicial complexes

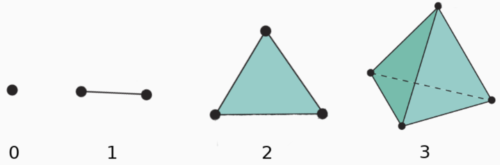
Definition

A **simplicial complex** is a pair $\langle V, S \rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)

Definition

- If $X, Y \in S$ are simplices such that $Y \subseteq X$, then Y is a face of X.
- The dimension of a simplex $X \in S$ is $\text{card}(X) - 1$.



Simplicial complexes

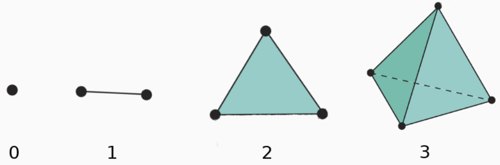
Definition

A **simplicial complex** is a pair $\langle V, S \rangle$ where

- V is a set of vertices
- S is a downward-closed family of subsets of V called simplices. (i.e., $X \in S$ and $Y \subseteq X$ implies $Y \in S$)

Definition

- If $X, Y \in S$ are simplices such that $Y \subseteq X$, then Y is a face of X.
- The dimension of a simplex $X \in S$ is $\text{card}(X) - 1$.



- A simplicial complex is pure if all maximal simplices are of the same dimension.

Simplicial topology

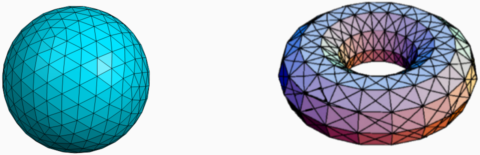
Definition

A **simplicial map** f from $\mathscr{C} = \langle V, S \rangle$ to $\mathscr{C}' = \langle V', S' \rangle$ is a function $f : V \to V'$ such that for all $X \in S$, $f(X) \in S'$.

Simplicial topology

Definition

A **simplicial map** f from $\mathscr{C} = \langle V, S \rangle$ to $\mathscr{C}' = \langle V', S' \rangle$ is a function $f : V \to V'$ such that for all $X \in S$, $f(X) \in S'$.

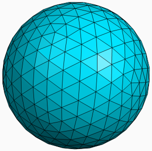


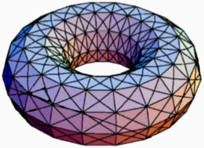
Simplicial complexes	$\simeq$	Topological spaces
Simplicial maps	$\simeq$	Continuous functions

Simplicial topology

Definition

A **simplicial map** f from $\mathscr{C} = \langle V, S \rangle$ to $\mathscr{C}' = \langle V', S' \rangle$ is a function $f : V \to V'$ such that for all $X \in S$, $f(X) \in S'$.





Simplicial complexes	$\simeq$	Topological spaces
Simplicial maps	$\simeq$	Continuous functions

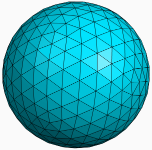
Simplicial complexes are a good setting for algebraic topology:

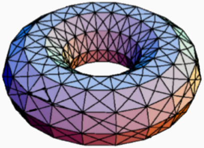
- Simplicial approximation theorem

Simplicial topology

Definition

A **simplicial map** f from $\mathscr{C} = \langle V, S \rangle$ to $\mathscr{C}' = \langle V', S' \rangle$ is a function $f : V \to V'$ such that for all $X \in S$, $f(X) \in S'$.





Simplicial complexes	$\simeq$	Topological spaces
Simplicial maps	$\simeq$	Continuous functions

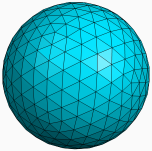
Simplicial complexes are a good setting for algebraic topology:

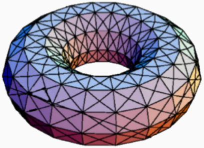
- Simplicial approximation theorem
- Sperner's Lemma, Index Lemma

Simplicial topology

Definition

A **simplicial map** f from $\mathscr{C} = \langle V, S \rangle$ to $\mathscr{C}' = \langle V', S' \rangle$ is a function $f : V \to V'$ such that for all $X \in S$, $f(X) \in S'$.





Simplicial complexes	$\simeq$	Topological spaces
Simplicial maps	$\simeq$	Continuous functions

Simplicial complexes are a good setting for algebraic topology:

- Simplicial approximation theorem
- Sperner's Lemma, Index Lemma
- Homology

Asynchronous Computability via Combinatorial Topology

Chromatic simplicial complexes

We fix a *finite* set P of colors/processes.

Definition

A **chromatic simplicial complex** is given by $\langle V, S, \chi \rangle$ where

- $\langle V, S \rangle$ is a simplicial complex
- $\chi : V \to P$ assigns colors to vertices, such that every simplex $X \in S$ has vertices of different colors $(\forall u, v \in X.\, \chi(u) \neq \chi(v))$

Chromatic simplicial complexes

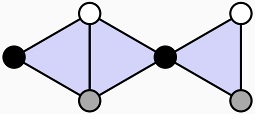
We fix a *finite* set P of colors/processes.

Definition

A **chromatic simplicial complex** is given by $\langle V, S, \chi \rangle$ where

- $\langle V, S \rangle$ is a simplicial complex
- $\chi : V \to P$ assigns colors to vertices, such that every simplex $X \in S$ has vertices of different colors ($\forall u, v \in X.\, \chi(u) \neq \chi(v)$)

Example: a pure chromatic simplicial complex of dimension 2:



Chromatic simplicial complexes

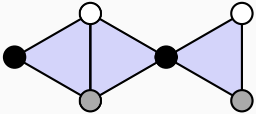
We fix a *finite* set P of colors/processes.[1]

Definition

A **chromatic simplicial complex** is given by $\langle V, S, \chi \rangle$ where

- $\langle V, S \rangle$ is a simplicial complex
- $\chi : V \to P$ assigns colors to vertices, such that every simplex $X \in S$ has vertices of different colors ($\forall u, v \in X. \chi(u) \neq \chi(v)$)

Example: a pure chromatic simplicial complex of dimension 2:



[1] All the pictures will have 3 processes in order to remain 2-dimensional; this is of course not a technical requirement.

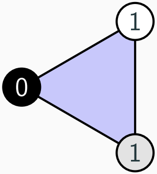
Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

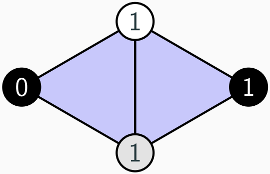
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

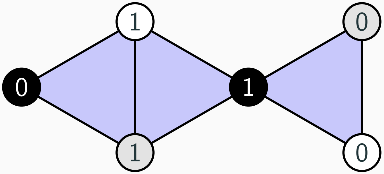
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

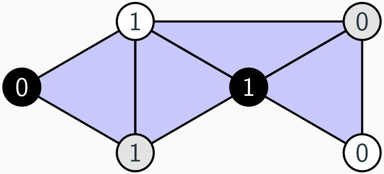
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

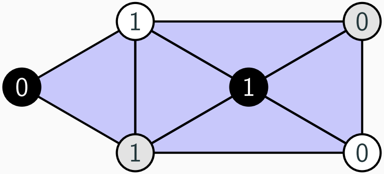
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

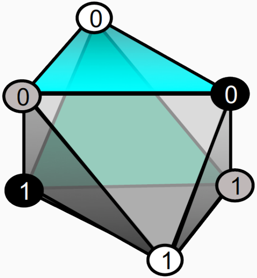
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

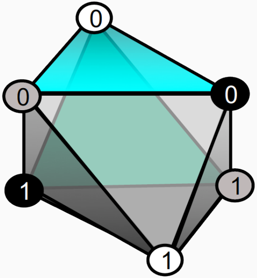
In the picture below, the three process names are represented as the colors black, grey, white:



Example: binary input complex for 3 processes

- Every process has input value either 0 or 1.
- Every process knows its value, but not the other values.

In the picture below, the three process names are represented as the colors black, grey, white:



Remark: for $n+1$ processes, we get a combinatorial n-sphere.

The immediate snapshot object

```
immediate_snapshot : int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

The immediate snapshot object

```
immediate_snapshot : int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

The immediate snapshot object

```
immediate_snapshot : int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

	2	

The immediate snapshot object

```
immediate_snapshot :  int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

	2	3

The immediate snapshot object

```
immediate_snapshot : int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

	2	3

R's view:

	2	3

The immediate snapshot object

```
immediate_snapshot :  int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

	2	3

Q's view:

	2	3

R's view:

	2	3

The immediate snapshot object

```
immediate_snapshot : int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

1	2	3

Q's view:

	2	3

R's view:

	2	3

The immediate snapshot object

```
immediate_snapshot :  int -> int array
```

Fix a number n of processes.
We suppose given a shared array A of size n.
Only process P_i can write in $A[i]$, but everyone can read it.

When P_i calls `immediate_snapshot(x)`:

- It writes its input value x in its own cell $A[i]$.
- Then atomically takes a snapshot of the whole array.

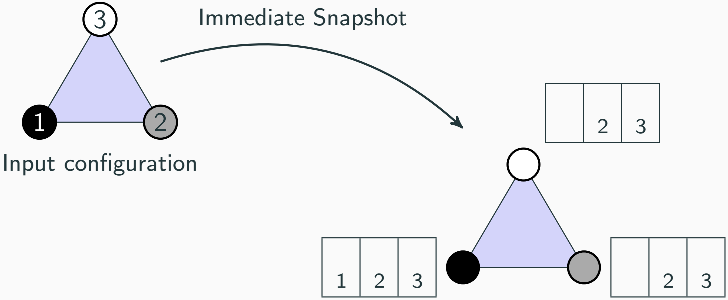
Example: for 3 processes P, Q, R with inputs 1,2,3.

$A =$

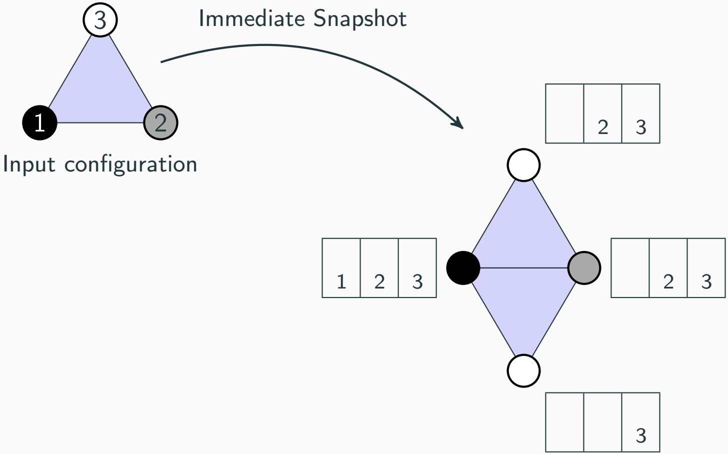
1	2	3

P's view:	1	2	3
Q's view:		2	3
R's view:		2	3

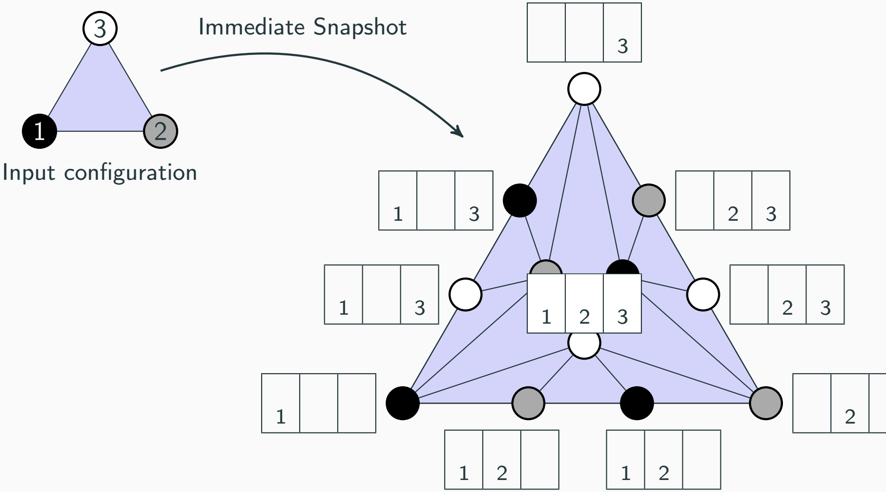
Protocol complex for immediate snapshot



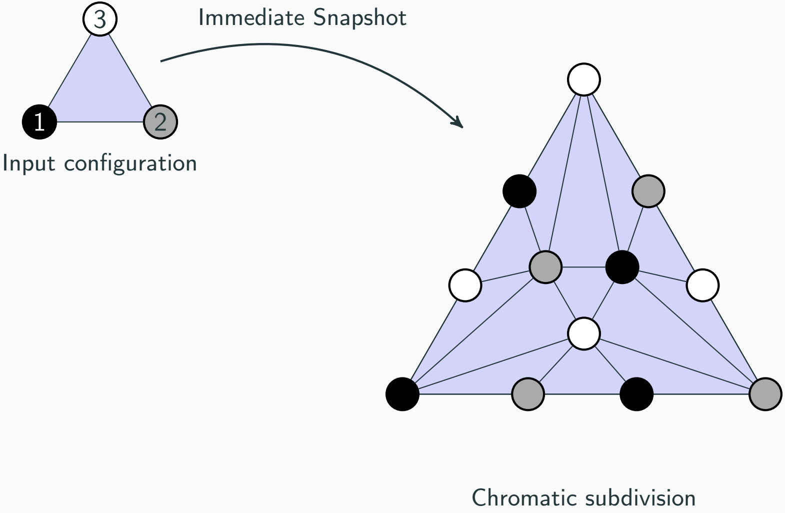
Protocol complex for immediate snapshot

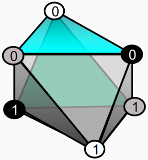


Protocol complex for immediate snapshot



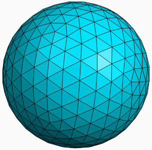
Protocol complex for immediate snapshot





Input complex

Immediate snapshot



Protocol complex

Key property: the topology is preserved.

The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathrm{in}_i \in \{0,1\}$.
After communicating, it decides an output $\mathrm{d}_i \in \{0,1\}$.

The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathrm{in}_i \in \{0,1\}$.
After communicating, it decides an output $\mathrm{d}_i \in \{0,1\}$.

Specification:

- **Agreement:** $\mathrm{d}_i = \mathrm{d_j}$ for all i, j.
- **Validity:** $\mathrm{d}_i \in \{\mathrm{in}_i \mid 1 \leq i \leq n\}$ for all i.

The (binary) consensus task

There is a fixed number n of processes.
Each process P_i has a binary input $\mathrm{in}_i \in \{0,1\}$.
After communicating, it decides an output $\mathrm{d}_i \in \{0,1\}$.

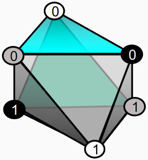
Specification:

- **Agreement:** $\mathrm{d}_i = \mathrm{d_j}$ for all i, j.
- **Validity:** $\mathrm{d}_i \in \{\mathrm{in}_i \mid 1 \leq i \leq n\}$ for all i.

Examples: for 3 processes,

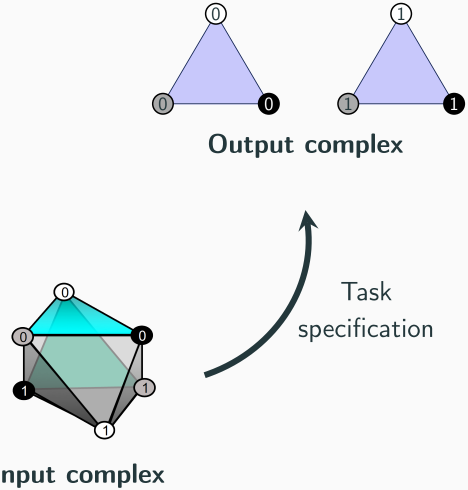
- if the inputs are $(0,0,0)$, the outputs must be $(0,0,0)$.
- if the inputs are $(1,0,1)$, the outputs can be $(0,0,0)$ or $(1,1,1)$.

Topological characterization of task solvability

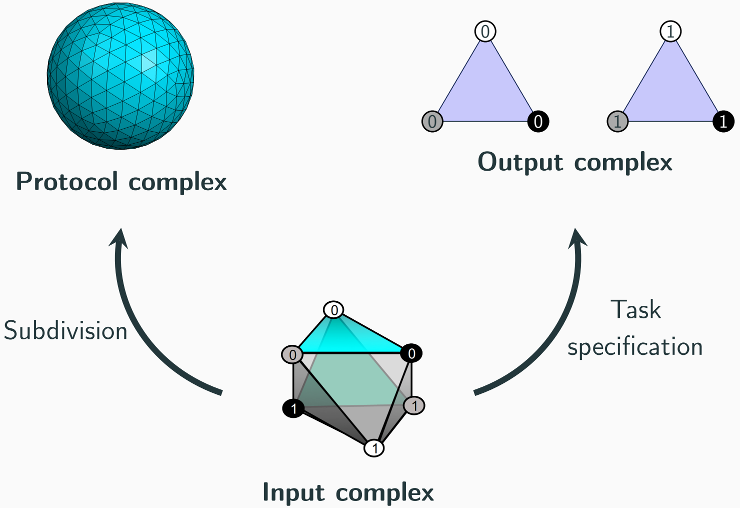


Input complex

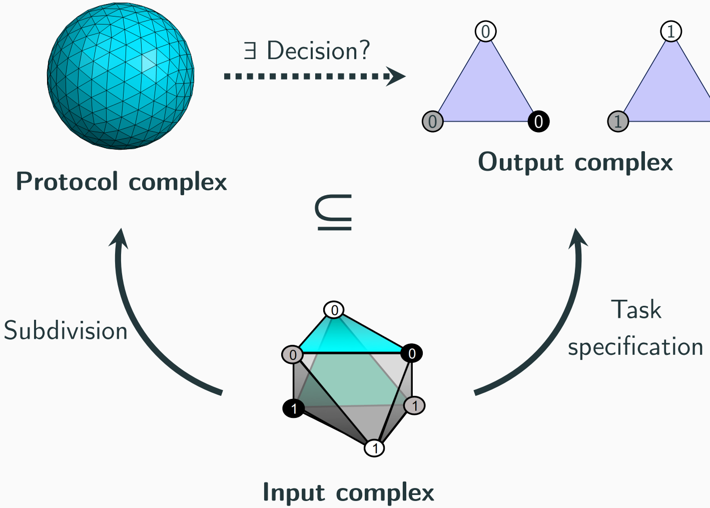
Topological characterization of task solvability



Topological characterization of task solvability



Topological characterization of task solvability



Asynchronous Computability Theorem (ACT)

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a wait-free protocol using read/write registers if and only if there exists a simplicial map from the protocol complex into the output complex that satisfies the task specification.

We have reduced a computational question ("Is the task solvable?") to a topological one ("Is there a simplicial map?").

Algebraic topology excels at answering such questions!

- Simplicial maps preserve k-connectedness.
- Compute algebraic invariants of spaces.

Some results in the field

Asynchronous Computability Theorem (ACT)

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write registers** if and only if there is a decision map from the protocol complex into the output complex that satisfies the task specification.

For instance: Update/scan wait-free protocols are:

- $(n-1)$-connected (no hole in any dimension)
- whatever number of communication rounds

Applications

- k-set agreement: generalisation of consensus; processes must terminate with at most k different values, taken from the initial values
- we cannot even solve k-consensus ($k \geq 1$) on such a machine!
- Approximate agreement: end up with "close enough" decisions: Possible!

Asynchronous Computability Theorem (ACT)

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write registers** if and only if there is a decision map from the protocol complex into the output complex that satisfies the task specification.

What if:

- we replace "wait-free" by "t-resilient"?

Asynchronous Computability Theorem (ACT)

Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write registers** if and only if there is a decision map from the protocol complex into the output complex that satisfies the task specification.

What if:

- we replace "wait-free" by "t-resilient"?

  ⟶ *Asynchronous Computability Theorems for t-resilient systems*, Saraph, Herlihy, Gafni (DISC 2016).

Asynchronous Computability Theorem (ACT)

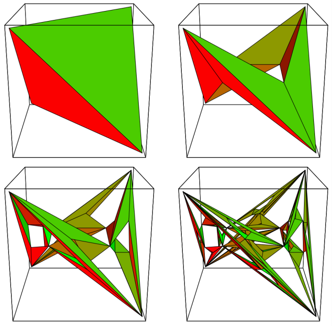
Theorem (Herlihy and Shavit, 1999)

A task is solvable by a **wait-free** protocol using **read/write registers** if and only if there is a decision map from the protocol complex into the output complex that satisfies the task specification.

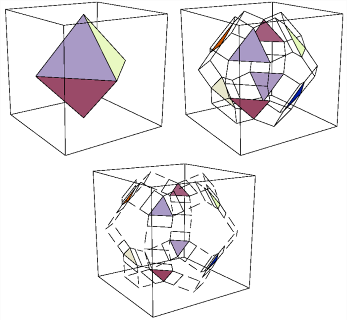
What if:

- we replace "wait-free" by "t-resilient"?
  ⟶ *Asynchronous Computability Theorems for t-resilient systems*, Saraph, Herlihy, Gafni (DISC 2016).
- we use other objects instead of read/write registers?
  ⟶ Many results with atomic operations (test&set, fetch&add etc.), (semi-) synchronous broadcast etc.

Protocol complexes for other objects

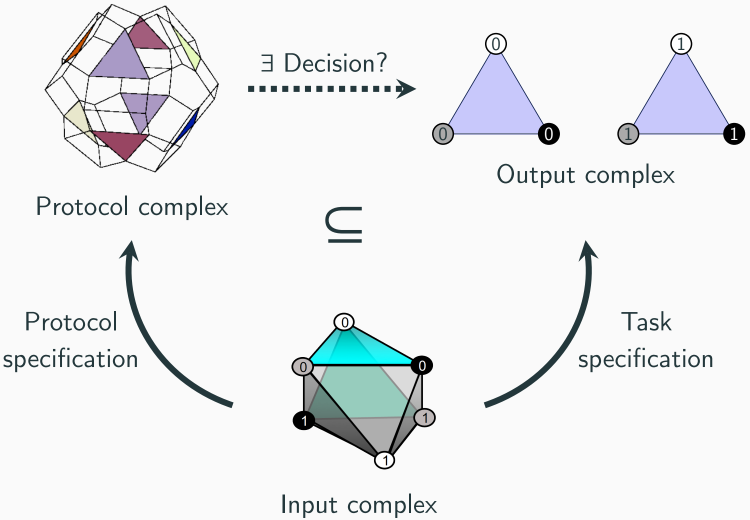


For test-and-set protocols
Herlihy, Rajsbaum, PODC'94

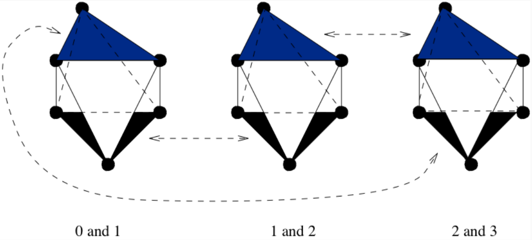


For synchronous message-passing
Herlihy, Rajsbaum, Tuttle, 2001

Example: task solvability for 3 procs, one round, 1-resilient synchronous broadcast



Ex: output complex, 3 processes, for the 2-set agreement



3 sphere, glued together, minus the simplex formed of the 3 values: connected but not 1-connected! (i.e. simply-connected) - compare with the output complex for consensus: 2 disconnected triangles!

Consequence

The 1-round protocol complex is connected, not simply connected:
$\Longrightarrow$ Impossible to solve consensus in 1 round, in a 1-resilient manner, with synchronous broadcast
$\Longrightarrow$ Possible to solve 2-set agreement in 1 round, in a 1-resilient manner, with synchronous broadcast

More generally speaking

In every synchronous broadcast protocol

- With $(n-2)$ steps (for n processes, synchronous message passing, one fault at most, i.e. 1-resilient), the sub-complex with all values equal to zero, and the one with all values equal to one, are connected
- Corollary : there is no algorithm, for this architecture, to solve consensus in (less than) $n-2$ rounds of communication (for at most one round)

Easy...

More generally

For r rounds of communication, and at most k faults in the synchronous model (message passing)

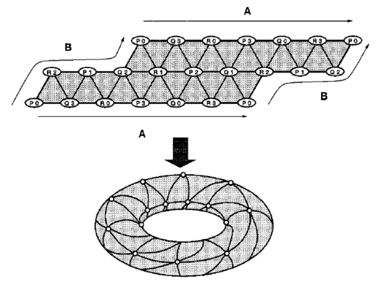
- The (sub-) protocol complex corresponding to an input, homeomorphic to the sphere in dimension $n-1$ (binary input values) $P(S^{n-1})$ is $(n-rk-2)$-connected
- This implies in particular that we have a lower bound of $n-1$ rounds for consensus, with $k=1$ (at most one crash)

Is it that simple?

Renaming

It is known to be implementable on an asynchronous system with message passing, in the presence of faults:

The $(n+1, K)$-renaming starts with $n+1$ processes which all have a name in $0, \ldots, N$. They must terminate with a name in in $0, \ldots, K$ with $n \leq K < N$.



Is it that simple?

Renaming

It is known to be implementable on an asynchronous system with message passing, in the presence of faults:

- (Attiya et al. JACM 1990) : wait-free solution for $K \geq 2n+1$, and none when $K \leq n+2$
- By using entirely geometric techniques: it was shown that there is there is no renaming when $K \leq 2n$ (Herlihy and Shavit STOC 1993)

Is it that simple?

Renaming

It is known to be implementable on an asynchronous system with message passing, in the presence of faults:

A mistake in the proof has been found in 2008 (Rajsbaum and Castaneda, PoDC). In fact, this is still computable when $K = 2n$ and $n+1$ is not a power of a prime number!
Example : computable for $(K,n) = (10,5),(14,7)\dots$ but not for $(K,n) = (4,2),(6,3),(8,4)\dots$

Is it that simple? (2)

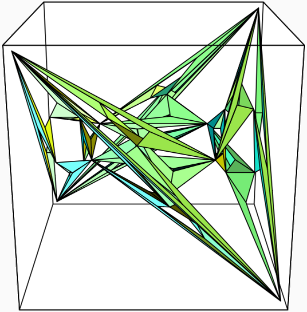
Real multiprocessors use much more refined synchronisation primitives

- test&set
- fetch&add
- compare&swap
- queues...

Exemple : Test&Set

Wait-free protocols with Test&Set

- are all $(n-3)$-connecterd
- are more expressive than scan/update protocols (for instance, we can solve the consensus with 2 processes)
- but we still cannot solve the consensus problem, in the presence of faults, for 3 processes or more



Some references

Other primitives, other models (asynchronous, synchronous, semi-synchronous etc.)...other results

- "Distributed Algorithms", N. Lynch
- "The art of multiprocessor programming", M. Herlihy, N. Shavit
- "Distributed Computing through Combinatorial Algebraic Topology", M. Herlihy, D. Kozlov, S. Rajsbaum

And also "Directed Topology and Concurrency", L. Fajstrup, E. Haucourt, E. Goubault, S. Mimram, M. Raussen

Conclusion

Conclusion

A deep connection between topology and distributed computing.

- Useful to prove impossibility results.
- Applies to a large range of computational models.

Conclusion

A deep connection between topology and distributed computing.

- Useful to prove impossibility results.
- Applies to a large range of computational models.

What I did not talk about:

- Full description of the proofs of impossibility results (next time!)
- Connection with epistemic logic (very good for proving algorithms correct, next time!).

Conclusion

A deep connection between topology and distributed computing.

- Useful to prove impossibility results.
- Applies to a large range of computational models.

What I did not talk about:

- Full description of the proofs of impossibility results (next time!)
- Connection with epistemic logic (very good for proving algorithms correct, next time!).
- Connection with swarm robotics: Bernardo, now!.