Section 6
Fault-Tolerant Consensus

Consensus

Description of the Problem

✓ Each process starts with an individual input from a particular value set V. Processes may fail by crashing.

✓All non-faulty processes are required to produce outputs from the value set V, subject to simple agreement and validity.

Correcteness Conditions

Agreement: No two processes decide on different values.
Validity: If all processes start with the same initial value $v \in V$, then v is the only decision value.
Termination: All non-faulty processes eventually decide.

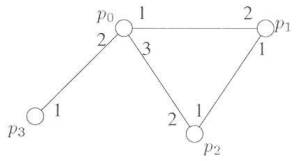
Motivation

- ❑ Processes in a database system may need to agree whether a transaction should commit or abort.
- ❑ Processes in a communication system may need to agree on whether or not a message has been received.
- ❑ Processes in a control system may need to agree on whether or not a particular other process is faulty.

Synchronous Shared Memory System

- Is there an algorithm that solves consensus in a synchronous shared-memory system?

Modeling Message Passing Systems



- n: number of processes (p_0, …, p_{n-1})
- Each process is modeled as a (possibly infinite) state machine.
- The state of process p_i contains 2r special components, where r is the number of edges that are incident to p_i:
  - $outbuf_i[l]$, $1 \le l \le r$, holds messages that p_i has sent to its neighbor over its lth incident channel but that have not yet been delivered to the neighbor;
  - $inbuf_i[l]$, $1 \le l \le r$, holds messages that have been delivered to p_i on its lth incident channel but that p_i has not yet processed with an internal computation step.
- The state e.g., of p_0 consists of p_0's local variables, and of six arrays:
- $inbuf_0[1]$, …, $inbuf_0[3]$: messages that have been sent to p_0, and p_0 has not yet processed.
- $outbuf_0[1]$, …, $outbuf_0[3]$: messages that have been sent by p_0 to each of the processes p_1, p_2, p_3, respectively, and which have not yet been delivered to p_1, p_2, p_3.

Modeling Message Passing Systems

- The state of a process p_i, excluding the $outbuf_i[l]$ components, comprises the accessible state of p_i.
- Each process has a state at which all inbuf arrays are empty.
- In each step executed by p_0, p_0 processes all messages stored in its inbuf arrays, the state of p_0 changes and at most one message is sent to each of its neighboring processes.

Modeling Message Passing Systems

Events in Message-Passing Systems

- **Delivery event, del(k,j,m)**: delivery of message m from process p_k to process p_j; just before the event occurs, m must be an element of $outbuf_k[l]$, where l is p_k's label for channel $\{p_k,p_j\}$. The event causes m to be deleted from $outbuf_k[l]$ and be inserted to $inbuf_j[l']$, where l' is p_j's label for channer $\{p_k,p_j\}$:
  - A message is delivered only if it is in transit and the only change is to move he message from the sender's outgoing buffer to the recepient's incoming buffer.
- **Computational event by p_j, comp(j)**: computation step of process pj in which pj's transition function is applied to its current accessible state; p_j changes state according to its transition function operating on p_j's accessible state and the set of messages specified by p_j's transition function are added to the $outbuf_j$ variables.

Modeling Message Passing Systems

Admissible execution

- Each process has an infinite number of computation events and every message sent is eventually delivered.

Message complexity

- Maximum number of messages that are sent in any execution.

A Simple Algorithm for Synchronous Message-Passing Systems

code for processor p_i, $0 \leq i \leq n-1$.

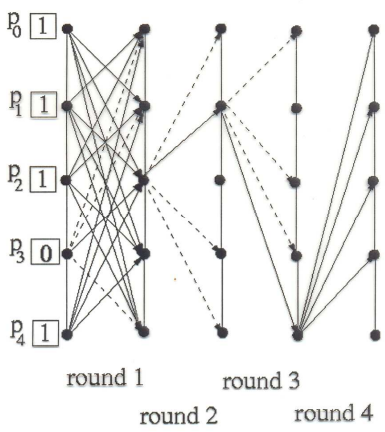
Initially $V = \{x\}$ // V contains p_i's input

round k, $1 \leq k \leq f+1$:

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all processors
2: receive S_j from p_j, $0 \leq j \leq n-1$, $j \neq i$
3: $V := V \cup \bigcup_{j=0}^{n-1} S_j$
4: if $k = f+1$ then $y := \min(V)$ // decide

- Each process maintains a set of the values it knows to exist in the system; initially, this set contains only its own input.
- At the first round, each process broadcasts its own input to all processes.
- For the subsequent f rounds, each process takes the following actions:
  - updates its set by joining it with the sets received from other processes, and
  - broadcasts any new additions to the set to all processes.
- After f+1 rounds, the process decides on the smallest value in its set.

A Simple Algorithm for Synchronous Message-Passing Systems



f = 3, n = 4

A Simple Algorithm for Synchronous Message-Passing Systems

Termination?

Validity?

Intuition for Agreement:

- Assume that a process p_i decides on a value x smaller than that decided by some other process p_j.
- Then, x has remain "hidden" from p_j for (f+1) rounds.
- We have at most f faulty processes. A contradiction!!!

Number of processes? n > f

Round complexity? (f+1) rounds

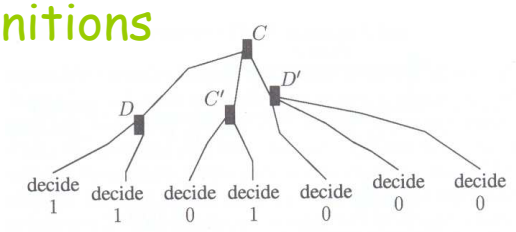
Message Complexity?

- n^2 * |V| messages, where V is the set of input values.

Impossibility of Consensus in Asynchronous Shared-Memory Systems

Theorem 1:. For n ≥ 2, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.

Useful Definitions



- The valence of a configuration C is the set of all values decided upon in any configuration reachable from C.
- C is univalent if this set contains one value; it is 0-valent if this value is 0 and 1-valent if this value is 1.
- If the set contains two values then C is bivalent.
- If C is bivalent and the configuration resulting by letting some process p take a step is univalent, we say that p is critical in C.
- **Recall that:** Two configurations C_1 and C_2 are similar to a process p, denoted $C_1 \sim^p C_2$, if the values of all shared variables and the state of p are the same in C_1 and C_2.

Impossibility of Consensus - Proof

Assume, by the way of contradiction, that A is a wait-free consensus algorithm.

Main Ideas of the Proof

- **We construct an infinite execution in which:**
  - **every process takes an infinite number of steps,**
  - **yet every configuration is bivalent,**
  - **and thus no process can decide.**
- **This contradicts the fact that the algorithm is wait-free.**

Impossibility of Consensus

Lemma 2: Let C_1 and C_2 be two univalent configurations. If $C_1 \sim^p C_2$, for some process p, then C_1 is v-valent, if C_2 is also v-valent, where $v \in \{0,1\}$.

Proof: Suppose C_1 is v-valent.

- Consider an infinite execution a from C_1 in which only p takes steps.
- Since the algorithm is supposed to be wait-free $\Rightarrow$ a is admissible and eventually p must decide in a.
- Since C_1 is v-valent $\Rightarrow$ p must decide v in a.
- The schedule of a can be applied from C_2
- Since $C_1 \sim^p C_2$ and only p takes steps, it follows that p decides v in this execution as well.
- Thus, C_2 is v-valent, as needed.

Impossibility of Consensus

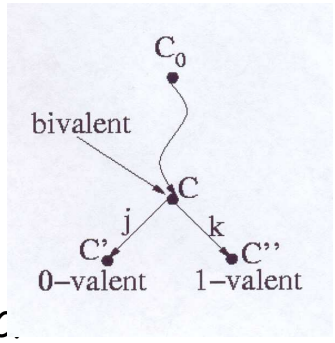
Lemma 3: There exists a bivalent initial configuration.

Proof: By contradiction.

- Let I_0 be the initial configuration in which all processes start with 0 $\Rightarrow$ I_0 is 0-valent.
- Let I_1 be initial configuration in which all processes start with 1 $\Rightarrow$ I_1 is 1-valent.
- Let I_{01} be the initial configuration in which p_0 starts with 0 and the remaining processes start with 1.
- $I_{01} \sim^{p0} I_0 \Rightarrow$ (by Lemma 2) I_{01} is 0-valent
- $I_{01} \sim^{p1} I_1 \Rightarrow$ (by Lemma 2) I_{01} cannot be 0-valent.

This is a contradiction!

Impossibility of Consensus



Lemma 4: If C is a bivalent configuration, then at least one processor is not critical in C.

- Proof: By the way of contradiction. Assume that all processes are critical in C.
- Since C is bivalent and all processes are critical in $C \Rightarrow$ there exists two process p_j and p_k such that:
  - if p_j takes a step from C, then the resulting configuration C' is 0-valent, and
  - if p_k takes a step from C the resulting configuration C'' is 1-valent.

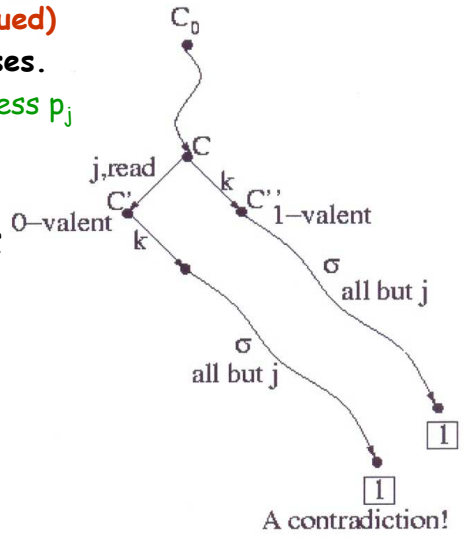
Impossibility of Consensus

Proof of Lemma 4 (continued)

Consider the following cases.

1. The first step of process p_j from C is a read.

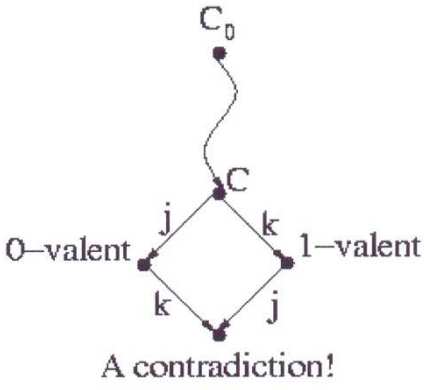
   The case where the first step of p_k from C is a read is symmetric.



Impossibility of Consensus

Proof of Lemma 4 (continued)

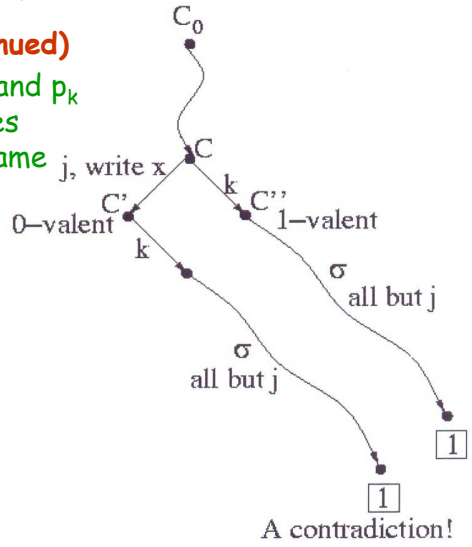
2. The first steps of p_j and p_k from C are both writes and they are to different variables.



Impossibility of Consensus

Proof of Lemma 4 (continued)

2. The first steps of p_j and p_k from C are both writes and they are to the same variable.



Impossibility of Consensus

Proof of Theorem 1

- We inductively create an admissible execution C_0 i_1 C_1 i_2 … in which the configurations remain bivalent forever.
  - By Lemma 3, there is an initial bivalent configuration; let it be C_0.
  - Suppose the execution has been created up to bivalent configuration C_k.
  - By Lemma 4, some process is not critical in C_k; denote this process by p_{ik}.
  - Then, p_{ik} can take a step without resulting in a univalent configuration.
  - We apply the event i_k to C_k to obtain C_{k+1} which is also bivalent.
- If we repeat this procedure forever, we will construct an execution in which all the configurations are bivalent. Thus, no process ever decides, contradicting the termination property of the algorithm and implying Theorem 1.