

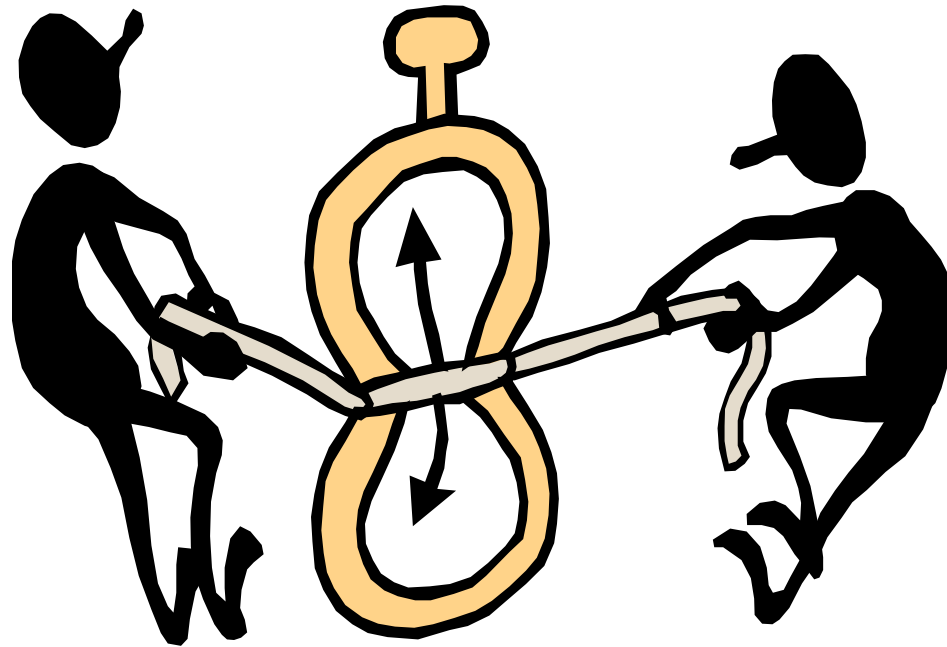
# CS603: Distributed Systems

## Lecture 5 & 6: Global States, Consensus

# Ordering Events in Distributed Systems

---

- Time is essential for ordering events in a distributed system
  - Physical time: local clock; global clock
  - Logical time: Lamport clocks, vector clocks



# This Lecture

---

- What happens if by “looking at the system”, the execution of the system results in an inconsistent cut?
- How does causality relates to this?
- What happens if for example an action is based on information that another process has not yet received?
- Can we detect such cases?
- Applications that can benefit: garbage collection, deadlock detection, transaction termination

# History of Events: Some Definitions

---

Given a **process**  $p_i$

- **Event**  $e_i^j$  is the event  $j$  at process  $i$
- **History** of process  $p_i$ ,  $h_i$  is a sequence of events that happened at  $p_i$

$$h_i = \langle e_i^0, e_i^1, \dots \rangle$$

- **Prefix history** at  $p_i$  up to  $k$ , is the the history of  $p_i$  up to the  $k^{\text{th}}$  event

$$h_i^k = \langle e_i^0, e_i^1, \dots, e_i^k \rangle$$

- **State**  $S_i^k$  is the state of process  $p_i$  immediately before the  $k^{\text{th}}$  event

# History of Events: More Definitions

---

Given a **set of processes**

- **Global history:** the set of all processes' histories

$$H = \square_i (h_i)$$

- **Global state:** the set of states at each process

$$S = \square_i (S_i^{k_i})$$

- **Cut:** a set of prefix histories

$$C \square H = h_1^{c1} \square h_2^{c2} \square \dots \square h_n^{cn}$$

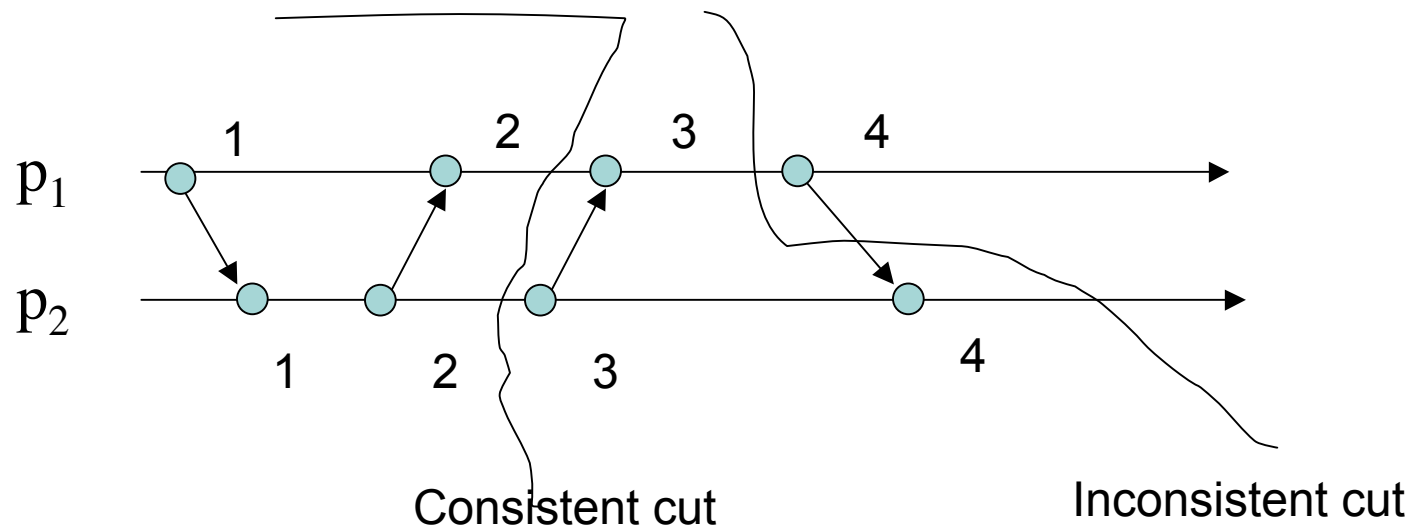
- **Frontier** of a cut: the set of last event that happened in each prefix history

$$C = \{e_i^{ci}, i = 1, 2, \dots, n\}$$

# Consistent Cuts

*Definition: A cut  $C$  is consistent if for any event  $e$  in the cut, if an event  $f$  'happened before'  $e$ , then  $f$  is also in the cut  $C$*

$$\forall e \in C \text{ (if } f \prec e \text{ then } f \in C)$$



# Global States: More Definitions

---

- **Consistent global state:** a global state that corresponds to a consistent cut
- **Run:** a total ordering of events in history  $H$  that is consistent with each process history  $h_i$ 's ordering
- **Linearization:** a run consistent with happens-before relation in  $H$ ; Linearizations pass through consistent global state
- **Reachability:** a global state  $S_k$  is reachable from global state  $S_i$ , if there is a linearization,  $L$ , that passes through  $S_i$  and then through  $S_k$ .

# Global State Predicate

---

- How do we use global states to reason about distributed systems?
- **Global state predicate**: a function from the set of global states to {TRUE, FALSE}
- **Stable global state predicate**: one that once it becomes true, it remains true in all future states reachable from that state.
- Examples:
  - “the system is deadlocked”
  - “all tokens in a token ring have disappeared”
  - “the computation has finished”

# Remember Safety and Liveness

---

- **Safety**: a condition that must hold in every finite prefix of a sequence (from an execution)

*“nothing bad happens”*

- **Liveness**: a condition that must hold a certain number of times

*“something good happens”*

# Stable Global States and Safety

---

- Look for undesirable properties, “bad things”
- Assume that a ‘bad thing’ BT (for example deadlock) is a global state predicate and  $S_0$  is the initial state of the system, then  
“Safety with respect to BT” means
  - $S$  reachable from  $S_0$ ,  $BT(S) = \text{FALSE}$

# Stable Global States and Liveness

---

- Look for desirable properties, “good things”
- Assume that a “good thing” GT (for example reaching termination) is a global-state-predicate and  $S_0$  is the initial state of the system then

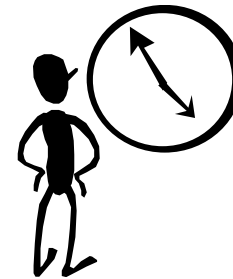
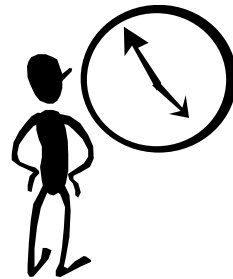
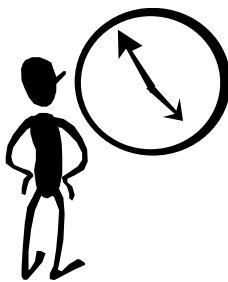
Liveness with respect to GT means:

For any linearization L starting at  $S_0$   $\exists$  state  $S_L$  reachable from  $S_0$  such that  $GT(S_L) = \text{TRUE}$

# Determining Global States

---

- If synchronized clocks are available, each process records its state at a known time  $t$
- How to obtain to state of the messages that transit the channels?
- What if synchronized clocks are not available?



# Snapshot Algorithm: Model

---

- **Goal: recording a consistent global state of an asynchronous system.**
- System Model:
  - **No failures and all messages arrive intact and only once**
  - Communication channels are unidirectional and FIFO ordered
  - There is a communication path between any two processes in the system
- Other assumptions
  - Any process may initiate the snapshot algorithm
  - The snapshot algorithm does not interfere with the normal execution of the processes
  - Each process in the system records its local state and the state of its incoming channels

# Chandy/Lamport Snapshot Algorithm

---

- Marker-sending rule for a process  $p$ :
  - Saves its own local state
  - Sends a marker to all other processes on their corresponding channels before sending any other message
- Marker-receiving rule for a process  $q$

If  $q$  has not recorded its state then

- $q$  records its state
- $q$  records the state of channel  $c$  as “empty”
- turn on recording of messages over other incoming channels
- for each outgoing channel  $C$ , send a marker on  $C$

else

- $q$  records the state of  $c$  as all the messages received over  $c$  after  $q$  recorded its state and before  $q$  received the marker along  $c$

# Consensus in Distributed Systems

---

- According to Merriam-Webster dictionary it means **general agreement**
- When do we need consensus in distributed systems?
  - Read-Modify-Write Memory
  - Database commit
  - Transactional filesystem
  - Totally ordered broadcast

# The Consensus Problem

---

- Input:
  - Each process has a value, either 1 or 0
- Properties:
  - *Agreement*: all nodes decide on the same value
  - *Validity*: if a process decides on a value, then there was a process that started with that value

# Consensus in a Synchronous System: No Faults

---

- Assumptions:
  - Communication is synchronous
- Algorithm - **requires 1 round**:
  - Each process sends its value to all the other processes
  - If all received values including its own are 1, then a process decides 1, otherwise decides 0



WHY IS THIS ALGORITHM CORRECT?

# Consensus in a Synchronous System with Crash Failures: Model

---

- Any process can crash, once crashed a process does not recover
- At most  $f$  processes can crash
- Communication is synchronous
- Network is a fully connected graph

# Consensus in a Synchronous System with Crash Failures: Properties

---

- Input:
  - 1 or 0 to each process
- Properties:
  - *Agreement*: all non-faulty processes decide on the same value
  - *Validity*: if a process decides on a value, then there was a process that started with that value
  - *Termination*: A non-faulty process decides in a finite time

NOTE: FAULTY PROCESSES MAY DECIDE DIFFERENTLY FROM CORRECT PROCESSES

# Consensus in a Synchronous System with Crash Failures: Algorithm

---

- Algorithm tolerates at most  $f$  failures, out of  $n$  nodes
- Each process maintains  $V$  the set of values proposed by other processes (initially it contains only its own value)
- In every round a process:
  - Sends to all other processes the values from  $V$  *that it has not sent before*
- After  $f+1$  rounds each process decides on the minimum value in  $V$

$$f < n \quad \square$$

# Consensus in a Synchronous System with Crash Failures: Algorithm

---

```
 $P_i::$   
var  
   $V$ : set of values initially  $\{v_i\}$ ;  
  
for  $k := 1$  to  $f + 1$  do  
  send  $\{v \in V \mid P_i \text{ has not already sent } v\}$  to all;  
  receive  $S_j$  from all processes  $P_j, j \neq i$ ;  
   $V := V \cup S_j$ ;  
endfor;  
  
 $y := \min(V)$ ;
```

# A Closer Look...

---

- Remember that communication is synchronous
- A process in a round may:
  - send messages to any set of processes
  - receive messages from any set of processes
  - do local processing
  - make a decision
  - crash
- **If a process  $p$  crashes in a round, then any subset of the messages sent by  $p$  in this round can be lost**

# Sketch Proof for the Agreement Property

---

Assume by contradiction that two processes decide on different values, this means they had different final set of values, **let's say  $p$  has a value  $v$  that  $q$  does not have**

- **How come that  $p$  got  $v$  and  $q$  did not?** The only possible case is that a third process  $s$ , sent  $v$  to  $p$ , and crashed before sending  $v$  to  $q$ . SO in ROUND  $f + 1$ , process  $s$  crashed (1 process)
- Because  $q$  does not have  $v$  it means that any process that may have sent  $v$  crashed also in round  $f$
- Proceeding in this way, we infer at least one crash in each of the preceding rounds.
- **STOP! We can have at most  $f$  crashes and we obtained that there are  $f+1$  crashes (one in each round)  $\rightarrow$  contradiction.**

**How about termination?**

# Variant: Uniform Consensus

---

- Input:
  - 1 or 0 to each process
- Properties:
  - *Uniform Agreement*: all processes (correct or faulty) decide on the same value
  - *Validity*: if a process decides on a value, then there was a process that started with that value
  - *Termination*: A non-faulty process decides in a finite time

---

# ASYNCHRONOUS SYSTEMS

# Consensus in Asynchronous Systems

---

- There is no asynchronous algorithm that achieves agreement on a one-bit value in the presence of crash faults. The result is true even if no crash actually occurs!
- Also known as the FLP result
- Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, April 1985, 32(2):374

# Execution, Configuration, Events

---

- Set of processes  $p_i$ , each process with a state  $s_i$
- **Configuration  $C_t$** : set of state of each process at some moment
- **Events**: send and deliver, events can change the state at a process
- **Execution**: sequence of configuration and events

# Consensus in a Synchronous System with Crash Failures: Properties

---

- Input:
  - 1 or 0 to each process
- Properties:
  - *Agreement*: all non-faulty processes decide on the same value
  - *Validity*: if a process decides on a value, then there was a process that started with that value
  - *Termination*: A non-faulty process decides in a finite time

# Proof Sketch

---

- Classify configurations as
- 0 - valent, will result in deciding 0
- 1 - valent, will result in deciding 1
- Bivalent - states with a bivalent outcome, decision is not already predetermined, outcome is unpredictable, it can be either 0 or 1.

# Proof Sketch (2)

---

- The goal is to construct an execution that does not decide, showing that the protocol remains forever indecisive
- Start with an initially bivalent state, identify an execution that would lead to a uni-valent state, let's say 0-valent
- The switch from bivalent to univalent is due to **an event  $e = (p,m)$**  in which some process  $p$  receives some message  $m$

# Proof Sketch (3)

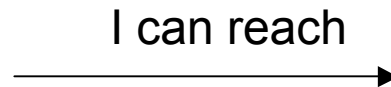
---

- Still constructing the execution that does not decide
- We will delay the  $e$  event for a while. Delivery of  $m$  would make the run univalent but  $m$  is delayed (fair-game in an asynchronous system)
- Since the protocol is indeed fault-tolerant there must be a run that leads to the other univalent state
- Now let  $m$  be delivered, this will bring the system back in a bivalent state
- **Decision can be delayed indefinitely**
- **WHY IS THIS SCENARIO POSSIBLE?**

# Proof: More Details

---

Initial bivalent configuration



Bivalent configuration

- *Lemma 1:* There exists an initial configuration that is bivalent.
- *Lemma 2:* Starting from a bivalent configuration  $C$  and an event  $e = (p, m)$  applicable to  $C$ , consider  $\mathcal{C}$  the set of all configurations reachable from  $C$  without applying  $e$  and  $\mathcal{D}$  the set of all configurations obtained by applying  $e$  to the configurations from  $\mathcal{C}$ , then  $\mathcal{D}$  contains a bivalent configuration.

**Theorem: There is always a run of events in an asynchronous distributed system such that the group of processes never reach consensus**

# Lemma 1: Proof Sketch

---

- Lemma 1: There exists an initial configuration that is bivalent
- Assume by contradiction that there is no bivalent initial configuration. List all initial configurations. There must be both 0-valent and 1-valent initial configurations. (Why????)
- Consider a 0-valent initial configuration  $C_0$  adjacent to a 1-valent configuration  $C_1$ : they differ only in the value corresponding to process  $p$ .

# Lemma 1 (cont.)

---

- **Lemma 1: There exists an initial configuration that is bivalent**
- Let this process  $p$  crash.
- Note that both  $C_0$  and  $C_1$  will lead to the same final configuration with the exception of internal state of  $p$  (they were identical, the only difference was determined by  $p$ ).
- If decision reached is 1, then  $C_0$  must be bivalent, if decision is 0 then  $C_1$  must be bivalent.
- *Thus, there exists an initial configuration that is bivalent.*

# Lemma 2

- **Lemma 2: Starting from a bivalent configuration, there is always another bivalent configuration that is reachable**
- Consider an event  $e = (p, m)$  that can be applied to a bivalent initial configuration  $\mathbf{C}$
- $\mathcal{C}$  the set of all configurations reachable from  $\mathbf{C}$  without applying  $e$
- $\mathcal{D}$  the set of all configurations obtained by applying  $e$  to a configuration in  $\mathcal{C}$
- We want to show that  $\mathcal{D}$  contains a bivalent configuration.

# Lemma 2 (cont.)

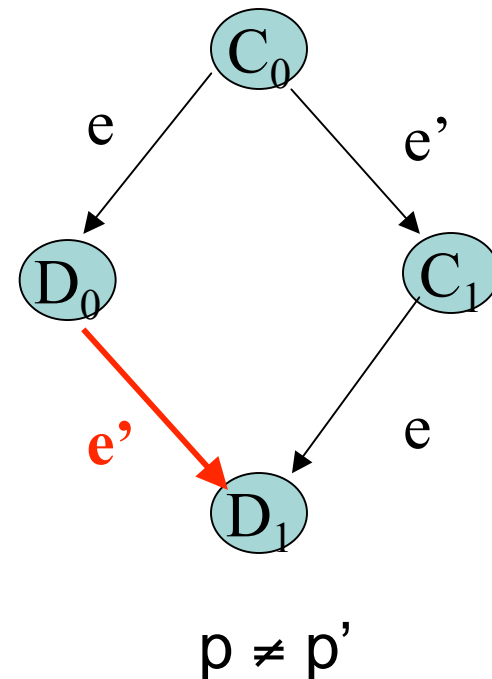
---

- Lemma 2: .....We want to show that  $\mathcal{D}$  contains a bivalent configuration
- Assume that there is no bivalent configuration in  $\mathcal{D}$
- However there are adjacent configurations  $C_0$  and  $C_1$  in  $\mathcal{C}$  such that  $C_1 = C_0$  followed by event  $e'=(p',m')$   
WHY (remember initial configuration is bivalent)
- Then denote  $D_0 = C_0$  followed by  $e=(p,m)$  and  $D_1 = C_1$  followed by  $e=(p,m)$
- $D_0$  is 0-valent and  $D_1$  is 1-valent

# Lemma 2 (cont.)

---

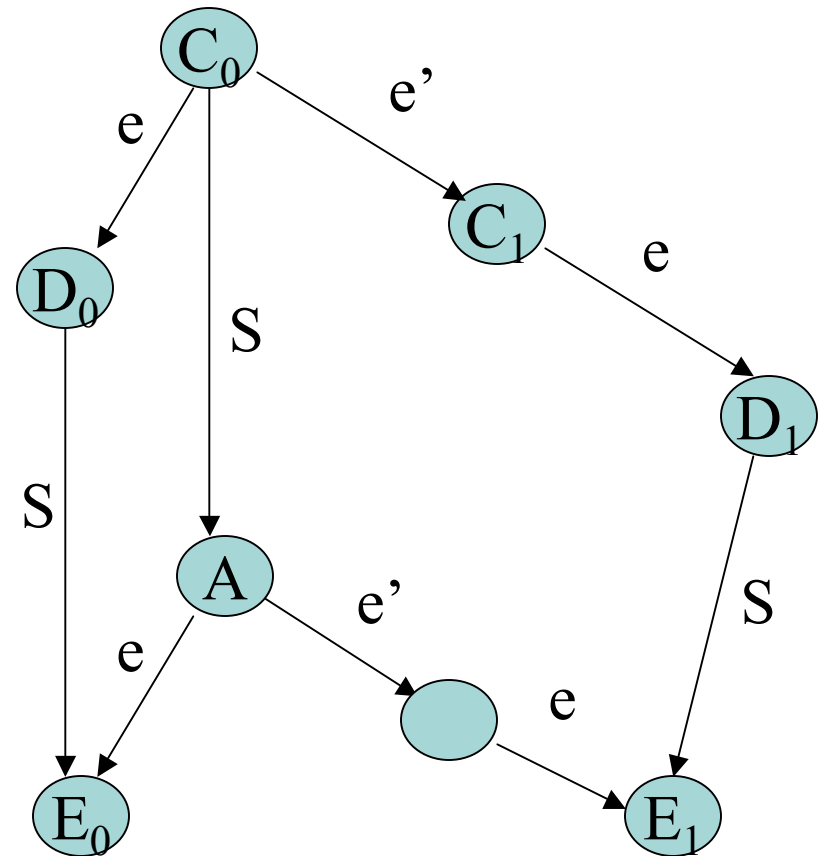
- Case 1:  $p$  and  $p'$  are different.
- If we apply  $e'$  to  $D_0$  we obtain  $D_1$  since  $e$  and  $e'$  are disjoint..
- **CONTRADICTION, any successor of a 0-valent configuration must be 0-valent.**



# Lemma 2 (cont.)

---

- Case 2: Process  $p$  is the same process as  $p'$
- $S$  is a run that reaches a decision, consider  $A$  that configuration.
- **We obtain that  $A$  is bivalent, contradiction!**



# *FLP impact on distributed systems design*

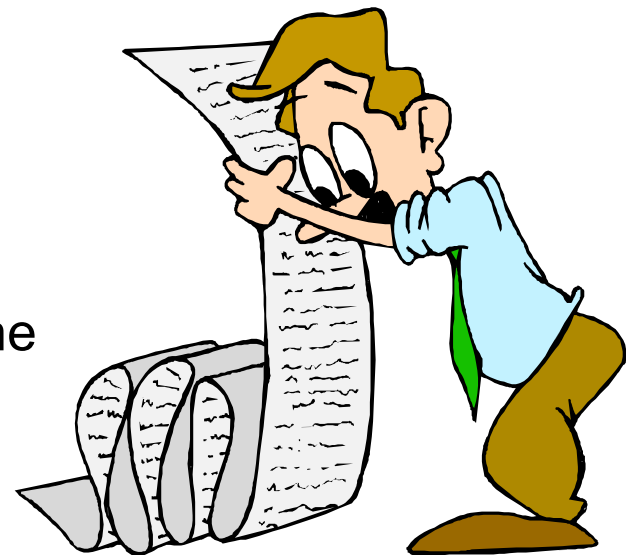
---

- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are extremely unlikely (“probability zero”)
  - Yet they imply that we can’t find a totally correct solution
  - And so “consensus is impossible” ( “not always possible”)
- A distributed system trying to agree on something in which process  $p$  plays a key role will not terminate if  $p$  crashes

# Consensus: Summary so Far

---

- Considered only benign failures
- In synchronous systems we have a  $f+1$  rounds consensus algorithm that can tolerate  $f$  failures,  $f < n$
- In asynchronous systems
  - We can not solve consensus
  - We can order events and determine consistent snapshots



# Next Week

---

- We will continue with consensus in the presence of Byzantine failures
- Project1 is due Sunday Jan. 29
- Homework1 is due next Sunday Feb 5.

# REQUIRED READING

---

- K. Mani Chandy and Leslie Lamport, Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, Vol. 3, No. 1, February, 1985, pp. 63-75.
- Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, April 1985, 32(2):374-382.

