

# Distributed Software Development Consensus and Agreement

Chris Brooks

Department of Computer Science  
University of San Francisco

## 6-2: Previously on CS 682

- Time is a big challenge in systems that don't share a clock.
- Insight: we often don't need to know the exact time that events occur.
- Instead, we need to know the order in which they happened.

## 6-3: Cause and Effect

- Cause and effect can be used to produce a partial ordering.
- Local events are ordered by identifier.
- Send and receive events are ordered.
  - If  $p_1$  sends a message  $m_1$  to  $p_2$ ,  $send(m_1)$  must occur before  $receive(m_1)$ .
  - Assume that messages are uniquely identified.
- If two events do not influence each other, even indirectly, we won't worry about their order.

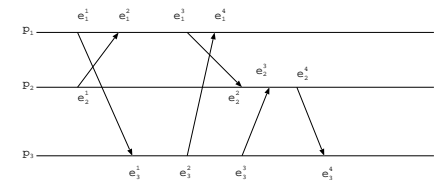
## 6-4: Happens before

- The *happens before* relation is denoted  $\rightarrow$ .
- Happens before is defined:
  - If  $e_i^k, e_i^l$  and  $k < l$ , then  $e_i^k \rightarrow e_i^l$
  - (sequentially ordered events in the same process)
  - If  $e_i = send(m)$  and  $e_j = receive(m)$ , then  $e_i \rightarrow e_j$
  - (send must come before receive)
  - If  $e \rightarrow e'$  and  $e' \rightarrow e''$ , then  $e \rightarrow e''$
  - (transitivity)
- If  $e \not\rightarrow e'$  and  $e' \not\rightarrow e$ , then we say that  $e$  and  $e'$  are concurrent. ( $e || e'$ )
- These events are unrelated, and could occur in either order.

## 6-5: Happens before

- Happens before provides a partial ordering over the global history. ( $H, \rightarrow$ )
- We call this a distributed computation.
- A distributed computation can be represented with a space-time diagram.

## 6-6: Space-time diagram



### 6-7: Space-time diagram

- Arrows indicate messages sent between processes.
- Causal relation between events is easy to detect
- Is there a directed path between events?
- $e_1^1 \rightarrow e_3^4$
- $e_1^2 || e_3^1$

### 6-8: Monitoring a distributed computation

- Recall that we want to know what the global state of the system is at some point in time.
- Active monitoring won't work
  - Updates from different processes may arrive out of order.
- We need to restrict our monitor to looking at *consistent cuts*
- A cut is consistent if, for all events  $e$  and  $e'$ 
  - ( $e \in C$  and  $e' \rightarrow e$ )  $\Rightarrow$   $e' \in C$
- In other words, we retain causal ordering and preserve the 'happens before' relation.

### 6-9: Synchronous communication

- How could we solve this problem with synchronous communication and a global clock?
- Assume FIFO delivery, delays are bounded by  $\delta$ 
  - $send(i) \rightarrow send(j) \Rightarrow deliver(i) \rightarrow deliver(j)$
  - Receiver must buffer out-of-order messages.
- Each event  $e$  is stamped with the global clock:  $RC(e)$ .
- When a process notifies  $p_0$  of event  $e$ , it includes  $RC(e)$  as a timestamp.
- At time  $t$ ,  $p_0$  can process all messages with timestamps up to  $t - \delta$  in increasing order.
- No earlier message can arrive after this point.

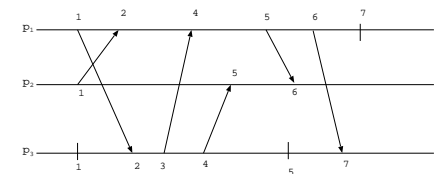
### 6-10: Why does this work?

- If we assume a delay of  $\delta$ , at time  $t$ , all messages sent before  $t - \delta$  have arrived.
- By processing them in increasing order, causality is preserved.
- $e \rightarrow e' \Rightarrow RC(e) < RC(e')$
- But we don't have a global clock!!

### 6-11: Logical clocks

- Each process maintains a logical clock. ( $LC$ ).
- Maps events to natural numbers. (0,1,2,3,...).
- In the initial state, all LCs are 0.
- Each message  $m$  contains a timestamp indicating the logical clock of the sending process.
- After each event, the logical clock for a process is updated as follows:
  - $LC(e) = LC + 1$  if  $e$  is a local or send event.
  - $LC(e) = \max(LC, TS(m)) + 1$  if  $e = receive(m)$ .
- The LC is updated to be greater than both the previous clock and the timestamp.

### 6-12: Logical clock example



### 6-13: Logical clocks

- Notice that logical clock values are increasing with respect to causal precedence.
  - Whenever  $e \rightarrow e'$ ,  $LC(e) < LC(e')$
- The monitor can process messages according to their logical clocks to always have a consistent global state.
- Are we done?
  - Not quite: this delivery rule lacks *liveness*.
  - Without a bound on message delay, we can never be sure that we won't have another message with a lower logical clock.
  - We can't detect gaps in the clock sequence.
  - Example: we'll wait forever for the nonexistent message for  $e_3$  from  $p_1$ .

### 6-14: Adding liveness

- We can get liveness by:
  - Delivering messages in FIFO order, buffering out-of-order messages.
  - Deliver messages in increasing timestamp order.
  - Deliver a message  $m$  from process  $p_i$  after at least one message from all other processes having a greater timestamp has been buffered.

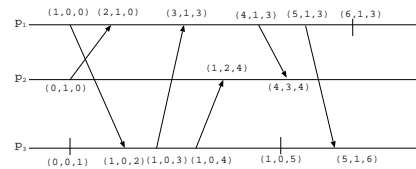
### 6-15: Causal delivery

- Recall that FIFO only refers to messages sent by the same process.
- *causal delivery* is a more general property which says that if  $send(m_1) \rightarrow send(m_2)$ , then  $deliver(m_1) \rightarrow (m_2)$  when different processes are sending  $m_1$  and  $m_2$ .
- Logical clocks aren't enough to give us causal delivery.

### 6-16: Vector clocks

- Solution: keep a "logical clock" for each process.
- these are stored in a vector  $VC$ .
  - Assumes number of processes in known and fixed.
- Update rule:
  - $VC(e)[i] = VC[i] + 1$  for send and internal.
  - $VC(e) = \max(VC, TS(m))$  for receive; then  $VC(e)[i] = VC[i] + 1$
- On receive, the vector clock takes the max on a component-by-component basis, then updates the local clock.

### 6-17: Vector Clock example



### 6-18: Vector clocks

- The monitor can then process events in order of ascending vector clocks.
- This ensures causality.
- Two clocks are inconsistent if  $c_1[i] < c_2[i]$  and  $c_1[i] > c_2[j]$
- If a cut contains no inconsistent clocks, it is consistent.
- Vector clocks allow the implementation of causal delivery.

### 6-19: Using cuts

- Cuts are useful if we need to rollback or restore a system.
- Consistent global cuts provide a set of consistent system states.
- Let us answer questions about the global properties of the system.
- Note that we still don't necessarily have the 'true' picture of the system.
  - Concurrent events may appear in an arbitrary order in a consistent run.
- We have enough information to reproduce all relevant ordering.

### 6-20: Applying causal delivery

- Causal delivery gives us almost all of the functionality that we need from a global clock.
- We can build on top of this to solve more complex coordination problems.
- Coordination often requires not only that all processes agree on state, but that all processes can ensure that every other process sees the same state.

### 6-21: Consensus and agreement

- A fundamental problem in distributed systems is getting a set of processes or nodes to agree on one or more values.
  - Is a procedure continuing or aborted?
  - What value is stored in a distributed database?
  - Which process is serving as coordinator?
  - Has a node failed?
- There are a set of related problems that require a set of processes to coordinate their states or actions.

### 6-22: Coordination via email

- An example:
  - Two people (A and B) want to meet at dusk tomorrow evening at a local hangout.
  - Each wants to show up only if the other one will be there.
  - They can send email to each other, but email may not arrive.
  - Can either one guarantee that the other will be there?

### 6-23: Failure models

- We'll want to distinguish what sorts of failures these algorithms can tolerate.
- No failure
  - Some of the algorithms we'll see can't tolerate a failure.
- Crash failure
  - This means that a node stops working and fails to respond to all messages.
- Byzantine failure
  - A node can exhibit arbitrary behavior.
  - This makes things pretty hard for us ...

### 6-24: Failure detection

- How can we detect whether a failure has happened?
- A simple method:
  - Every  $t$  seconds, each process sends an "I am alive" message to all other processes.
  - Process  $p$  knows that process  $q$  is either *unsuspected*, *suspected*, or *failed*
- If  $p$  sees  $q$ 's message, it knows  $q$  is alive, and sets its status to unsuspected.
- What if it doesn't receive a message?

#### 6-25: Failure detection

- Depends on our communication model.
- Synchronous communication: if after  $d$  seconds (where  $d$  is the maximum delay in message delivery) we haven't received a message from  $p$ ,  $p$  has failed.
- Asynchronous or unreliable communication: if the message is not received, we can say that  $p$  is suspected of failure.

#### 6-26: Failure detection

- Other problems:
  - What if  $d$  is fairly large?
  - We can think processes are still running that have in fact crashed.
- This is what's called an *unreliable* failure detector.
- It will make mistakes, but, given enough information, it may still be of use.
- Can provide hints and partial information.
- As we look at different algorithms, we'll need to think about whether we can detect that a process has failed.

#### 6-27: Multicast: a brief digression

- The Coulouris chapter talks quite a bit about how to achieve different properties with multicast communication.
  - Reliable multicast
  - Ordered multicast
    - FIFO ordering
    - Total ordering
    - Causal ordering
- The punchline: Totally ordered multicast is equivalent to the consensus problem.

#### 6-28: What is multicast?

- Consider that a process needs to send a message to a *group* of other processes.
- It could:
  - Send a point-to-point message to every other process.
    - Inefficient, plus need to know all other processes in group.
  - Broadcast to all processes in subnet.
    - Wasteful, won't work in wide-area network.
- Multicast allows the process to do a single send. Packet is delivered to all members of the group.

#### 6-29: Multicast groups

- Notice that multicast is a packet-oriented communication.
  - Same send/receive semantics as UDP
- A process joins a multicast group (designated by an IP address)
- It then receives all messages sent to that IP address.
- Groups can be closed or open.
- Multicast can be effectively used to do shared whiteboards, video or audio conferencing, or to broadcast speeches or presentations.
  - Middleware needed to provide ordering.

#### 6-30: Mutual exclusion

- Mutual exclusion is a familiar problem from operating systems.
  - There is some resource that is shared by several processes
  - Only one process can use the resource at a time.
  - Shared file, database, communications medium
- Processes request to enter their *critical section*, then enter, then exit.
- In a centralized system, this can be negotiated with shared objects. (locks or mutexes).
- Distributed systems rely only on message passing!

### 6-31: Mutual exclusion

- Our goals for mutual exclusion:
  - safety: Only one process uses the resource at a time.
  - liveness: everyone eventually gets a turn.
    - This implies no deadlock or starvation.
  - ordering: if process  $i$ 's request to enter its CS happens-before (in the causal sense) process  $j$ 's, then process  $i$  should enter first.

### 6-32: Mutual exclusion: centralized server

- One solution is to use a centralized server to manage access.
- To enter the critical section, a process sends a request to the server.
  - If no one is in a critical section, the server returns a token. When the process exits the critical section, the token is returned.
  - If someone already has the token, the request is queued.
- Requests are serviced in FIFO order.

### 6-33: Mutual exclusion: centralized server

- If no failures occur, this ensures safety and liveness.
- Ordering is not satisfied.
- Central server provides a bottleneck and a single point of failure.

### 6-34: Mutual exclusion: token ring

- Rather than dedicating a server, processes are logically arranged in a ring.
  - This may not have anything to do with network topology; we just assign an order.
- The token is passed clockwise (for example) around the ring.
- When the token is received, a process can enter its critical section.
- If the token is not needed, it is immediately passed on.

### 6-35: Mutual exclusion: token ring

- Achieves liveness and safety, but not ordering.
- Also can use a lot of bandwidth when no process needs the resource.
- Also can't handle failure.
  - What if the process with the token fails?
  - If we can detect failure, we can generate a new token.
  - Leader election can deal with this.

### 6-36: Mutual exclusion: multicast

- Assume each process has a distinct identifier
- Assume each process can keep a logical clock.
- A process has three states: Released, waiting, held.
- When a process  $p$  wants to enter the critical section:
  - $p$  sets its state to waiting
  - $p$  sends a message to all other processes containing its ID and logical timestamp.
  - Once all other processes respond,  $p$  can enter.

#### 6-37: Mutual exclusion: multicast

- When a message is received:
  - If state is held, queue message
  - If state is waiting and timestamp is after local timestamp, queue message.
  - Else, reply immediately.
- When exiting the critical section:
  - set state to released
  - reply to queued requests.

#### 6-38: Mutual exclusion: multicast

- Example: consider  $p_1, p_2, p_3$ .
- $p_3$  doesn't need CS.  $T(p_1) = 41, T(p_2) = 34$ .
- $p_1$  and  $p_2$  request CS.
- $p_3$  replies immediately to both.
- When  $p_2$  gets  $p_1$ 's request, it queues it.
- $p_1$  replies to  $p_2$  immediately.
- Once  $p_2$  exits, it replies to  $p_1$ .

#### 6-39: Mutual exclusion: multicast

- Provides liveness, safety
- Also provides ordering
  - That's the reason for logical clocks.
- Still can't deal with failure.
- Also scaling problems.
- Optimization: can enter the CS when a majority of replies are received.

#### 6-40: Dealing with failures

- If a failure occurs, it must first be detected.
  - As we've seen, this can be difficult.
- Once failure is detected, a new group can be formed and the protocol restarted.
- Group formation involves a two-phase protocol.
  - Coordinator broadcasts group change to all members.
  - Once all reply, a commit is broadcast to all members.
  - Once all members reply to the commit, a new group is formed.

#### 6-41: Election algorithms

- How can we decide which process should play the role of server or coordinator?
- We need for all processes to agree.
- We can do this by means of an election.
- Any process can start an election
  - for example, if it notices that the coordinator fails.
- We would still like safety (only one process is chosen) and liveness (the election process is guaranteed to find a winner).
  - Even when more than one election is started simultaneously.

#### 6-42: Choosing a leader

- Assume each process has an identifying value.
- Largest value will be the new leader.
  - We could use load, or uptime, or a random number.

#### 6-43: Ring-based election algorithms

- Assume processes are arranged in a logical ring.
- A process starts an election by placing its identifier and value in a message and sending it to its neighbor.

#### 6-44: Ring-based election algorithms

- When a message is received:
  - If the value is greater than its own, it saves the identifier and forwards the value to its neighbor.
  - Else if the receiver's value is greater and the receiver has not participated in an election already, it replaces the identifier and value with its own and forwards the message.
  - Else if the receiver has already participated in an election, it discards the message.
  - If a process receives its own identifier and value it knows it is elected. It then sends an elected message to its neighbor.
  - When an elected message is received, it is forwarded to the next neighbor.

#### 6-45: Ring-based election algorithms

- Safety is guaranteed - only one value can be largest and make it all the way through the ring.
- Liveness is guaranteed if there are no failures.
- Inability to handle failure once again ...

#### 6-46: Bully algorithm

- The *bully* algorithm can deal with crash failures.
  - Assumption: synchronous, reliable communication
- When a process notices that the coordinator has failed, it sends an election message to all higher-numbered processes.
- If no one replies, it declares itself coordinator and sends a new-coordinator message to all processes.
- If someone replies, its job is done.
- When process  $q$  receives an election message from a lower-numbered process:
  - Return a reply.
  - Start an election.

#### 6-47: Bully algorithm

- Guarantees safety and liveness.
- Can deal with crash failures
- Assumes that there is bounded message delay
  - Otherwise, how can we distinguish between a crash and a long delay?

#### 6-48: Consensus

- All of these algorithms are examples of the consensus problem
  - All processes must agree on a state
- Let's take a step back and think about when the consensus problem can be solved.

#### 6-49: Consensus

- We'll start with a set of processes  $p_1, p_2, \dots, p_n$ .
- All processes can propose a value, and everyone must agree at the end.
- We'll assume that communication is reliable.
- Processes can fail.
  - Both Byzantine and crash failures.
- We'll also specify whether processes can digitally sign messages.
  - This limits the damage Byzantine failures can do.
- We'll specify whether communication is synchronous or asynchronous.

#### 6-50: Consensus

- Our goals:
  - Termination: eventually every process decides on a value
  - Agreement: All processes agree on a value
  - Integrity: If all correct processes propose the same value, that value is the one selected.

#### 6-51: Byzantine generals

- All commanders must agree whether to attack or not.
- Commanders can be treacherous and send different messages to each commander
  - Slightly different from consensus: one process supplies a value that others must agree on.
- Can this be solved? Can the head commander get consensus?
  - Depends on our communication assumptions.

#### 6-52: Consensus in synchronous systems

- If we don't have reliable communication, consensus is impossible, even without failures.
  - Email example
- With reliable communication, we can solve consensus for crash failures.
- Multiple rounds of communication are required to account for failures.

#### 6-53: Byzantine generals in a synchronous system

- In the Byzantine generals problem, we can solve it in a synchronous system if less than  $1/3$  of the processors fail.
  - Three-processor problem is impossible.
- Couloris shows how to solve this - intuition is to use majority.
- If processes can sign their messages, we can solve the Byzantine generals problem for any number of failures.

#### 6-54: Byzantine generals in an asynchronous system

- In asynchronous systems, the news is not so good.
- In asynchronous systems, it is impossible to guarantee that we will reach consensus, even in the presence of a single crash failure.
- This means that we can't do:
  - Asynchronous Byzantine generals
  - Asynchronous totally ordered multicast

#### 6-55: What do we do in practice?

- Notice the word *guaranteed*. We may be able to reach consensus in some cases, we just can't promise it.
- We can introduce systems that can survive crash failures.
- We can introduce failure detectors.
- We can use randomized behavior to foil Byzantine processes.

#### 6-56: Summarizing

- We can survive  $1/3$  Byzantine failures in a synchronous system with reliable delivery.
- In an asynchronous system, we can't guarantee consensus after a single crash failure.
- Without reliable communication, consensus is impossible to guarantee.
- In general, we can trade off process reliability for network reliability.

#### 6-57: Summary

- Consensus can take a number of forms:
  - Mutual exclusion
  - Leader election
  - Consensus
- Many special-purpose algorithms exist.
- General results about what is possible can help in designing a system or deciding how (or whether) to tackle a problem.