Time and global state; Coordination and agreement; Distributed transactions

Oleg Batrashev

Institute of Computer Science

December 11, 2015

Statements

- I use pictures from the book (Distributed Systems: Concepts and Design by George Coulouris et al)
- Chapters 14-17 are covered
  - only a fraction of materials is covered
- To make things easier to understand I have
  1. omit some necessary conditions for the problems, like if the system must be synchronous or not
  2. omit some technical details from the solutions, presenting in a very sketchy way
- Although,
  1. conditions are important I do not think you can squeeze 4 chapters in 1 lecture preserving them
  2. details are important, because that may show you do not just memorized the slides but also understood how algorithms work

Outline

1. Time and global state
   - Physical clocks
   - Logical clocks
   - Global state
2. Coordination and agreement
   - Distributed mutual exclusion
   - Elections
   - Consensus
3. Distributed transactions
   - Transactions
   - Two-phase commit

Time problem

- There is always not enough time

Time problem

- There is always not enough time – just joking :)
- There is no *global time* in distributed systems
  - time is relative like in relativity theory
  - root cause: two computers cannot synchronize time *perfectly*
- Imagine event A on process 1 and event B on proccess 2:
  - processes decide A was before B based on physical clocks,
  - ...imperfectly synchronized clocks...
  - it may happen that B caused A through a message from process 2 to process 1
  - disaster: "effect" *happened before* "cause"

Time problem

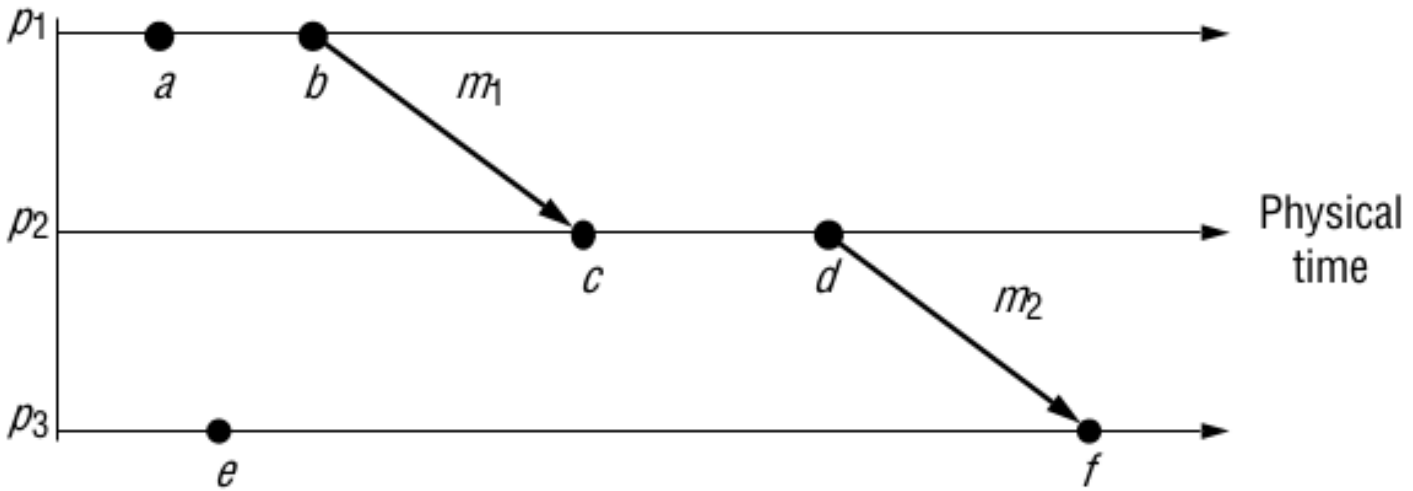
- There is always not enough time – just joking :)
- There is no *global time* in distributed systems
  - time is relative like in relativity theory
  - root cause: two computers cannot synchronize time *perfectly*
- Imagine event A on process 1 and event B on proccess 2:
  - processes decide A was before B based on physical clocks,
  - ...imperfectly synchronized clocks...
  - it may happen that B caused A through a message from process 2 to process 1
  - disaster: "effect" *happened before* "cause"
- **Solution**: use logical clocks
  - *happened-before* relation is a central idea

Synchronizing physical clocks

- Cristian's method
  - send single message and wait for the response with the time t of remote computer
  - T_{round} – total time for the two messages to travel
  - simple estimate – set local time to $t + \frac{T}{2}$
- The Network Time Protocol

Happened-before relation

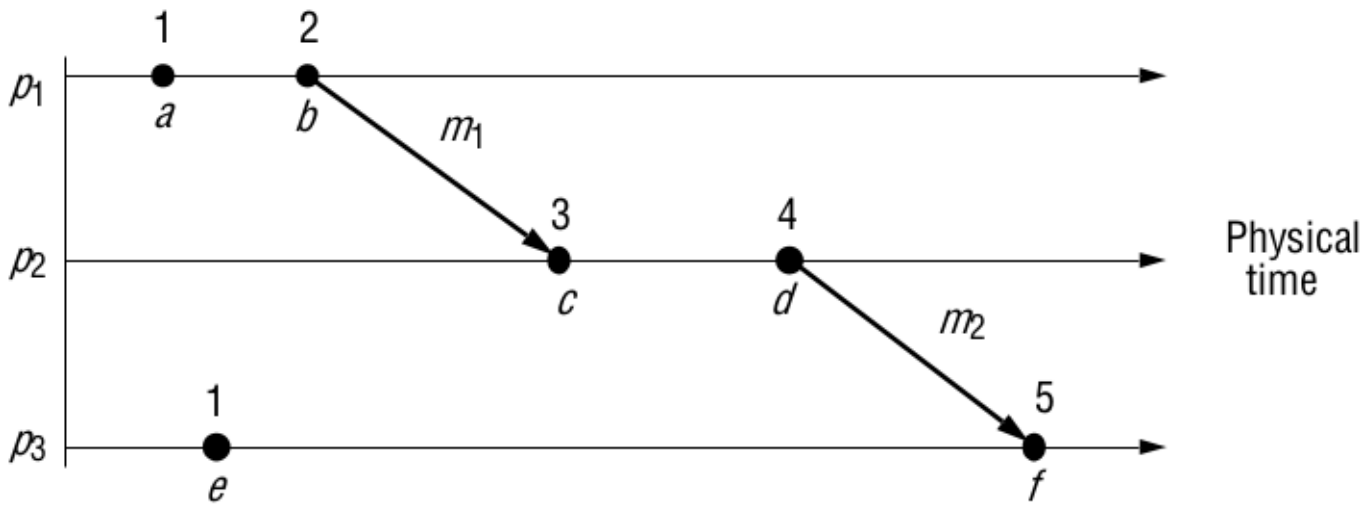
1. On a single process (thread) all events are ordered
   - $e \rightarrow e'$ if e' is after e on the same thread
   - earlier one (e) one *happens-before* ($\rightarrow$) later one (e')
2. Message send event *happens-before* message receive event
   - we are talking about the same message m here
   - $e \rightarrow e'$ if $e = \text{send}(m)$ and $e' = \text{recv}(m)$
3. Transitivity: $e \rightarrow e'$ and $e' \rightarrow e'' \Rightarrow e \rightarrow e''$



Lamport clocks

Each process i keeps local time L_i – some integer value

1. $L_i$is incremented by 1 before each event
2. L is propagated with each message:
   a. sending a message m process p_i piggybacks its time $t = L_i$
   b. on receiving m process p_j computes $L_j := \max(L_{j-1}, t) + 1$



Totally ordered clocks

- Happened-before and Lamport clocks are *partially ordered*
  - PO: may $\exists e, e'$ so that neither $e \rightarrow e'$ nor $e' \rightarrow e$
  - LC: usually happen that for some events $L_i = L_j$
- Some algorithms may want events to be *totally ordered*

Solution

Order Lamport clocks by adding proccess number (L_i, i)

Vector clocks

- Problem: upon receiving (m_1, t_1) and (m_2, t_2) we cannot tell if corresponding send events are ordered
  - e.g. $\text{send}(m_1) \rightarrow \text{send}(m_2)$
  - i.e. whether m_2 sender knew about everything m_1 sender has done before sending m_1
  - e.g. m_1 sender wants to become master and broadcasted m_1

Vector clocks

- Problem: upon receiving (m_1, t_1) and (m_2, t_2) we cannot tell if corresponding send events are ordered
  - e.g. $\text{send}(m_1) \rightarrow \text{send}(m_2)$
  - i.e. whether m_2 sender knew about everything m_1 sender has done before sending m_1
  - e.g. m_1 sender wants to become master and broadcasted m_1

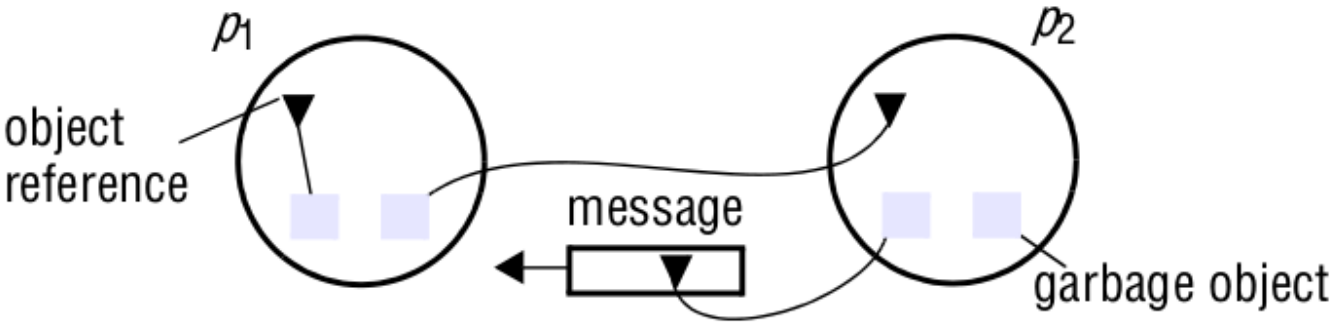
Solution: use *vector clocks*

1. process p_i keeps track of times of all other processes $L_i[j]$
2. L is propagated with each message
   1. sending m from process i piggyback the local vector to it L_i
   2. receiving m in process j update local vector L_j

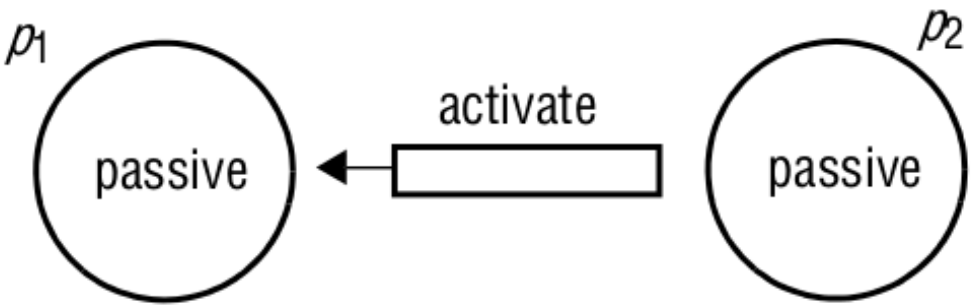
For interested the details are in the book.

The need for global state

- Distributed garbage collection



- Distributed deadlock detection
- Distributed termination detection



Local and global states

Local state – on each process p_i we have

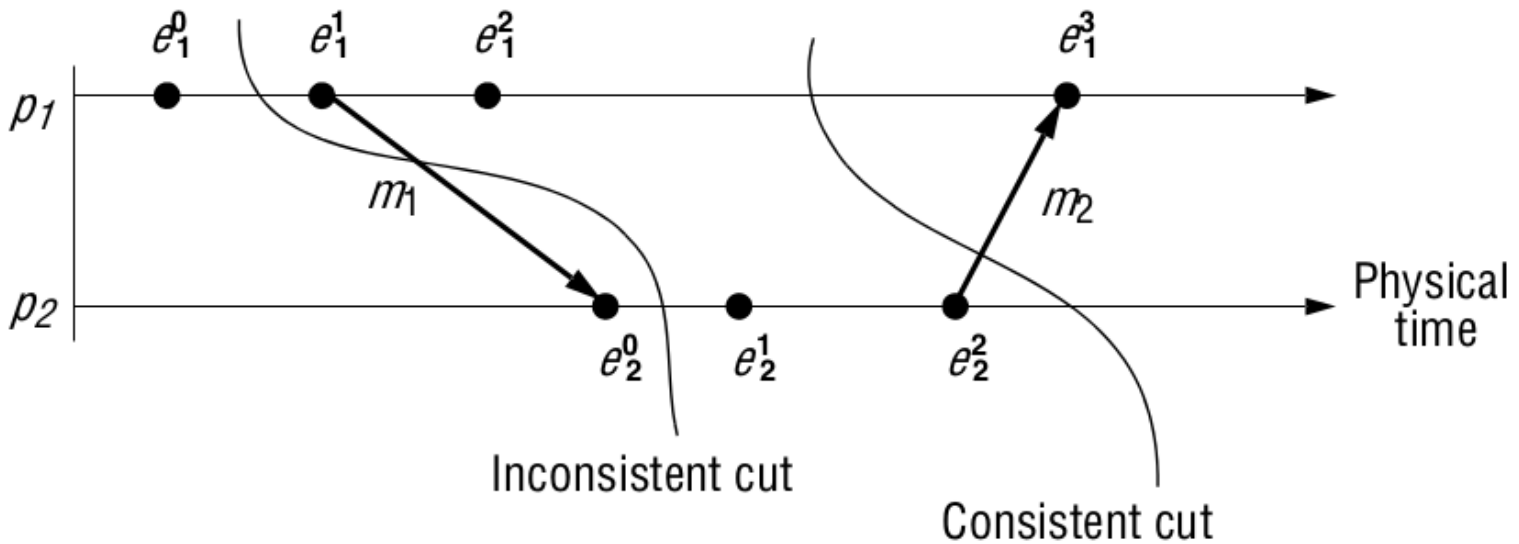
- history of events $\langle e_i^0, e_i^1, e_i^2, \ldots \rangle$
- state s_i^k – immediatelly before event e_i^k occurs

Global state – local states of all processes

- physical time t when everyone saves its local state
  - can't perfectly synchronize physical clocks!
- is there meaningful global state if local states are recorded at different moments in time?
  - yes there is!
- do not forget to save channel states
  - sender saves sent messages as its local state and later discards those received by the recepient

Consistent cuts

- **Cut** is defined by the points where we save local states
- **Consistent cut** does not contain "effect" without its "cause"
  - e.g. message receive event without message send event
- cc is the state that may have happened as a real-time global state, if CPU speed or message travel times were different
  - try to "move" events along axes



The 'snapshot' algorithm of Chandy and Lamport

- Idea – piggyback marker on a message
  - signifies that the sender saved its local state just before sending this message
- Receiver of such marker (unless already done so)
  - saves its local state before processing the message
  - starts recording messages from other incoming channels
- Think about the picture from the previous slide
  - where the cut would be
  - if recorded messages really form a channel state

The book has more formal definition

1 Time and global state
- Physical clocks
- Logical clocks
- Global state

2 Coordination and agreement
- Distributed mutual exclusion
- Elections
- Consensus

3 Distributed transactions
- Transactions
- Two-phase commit

Concepts

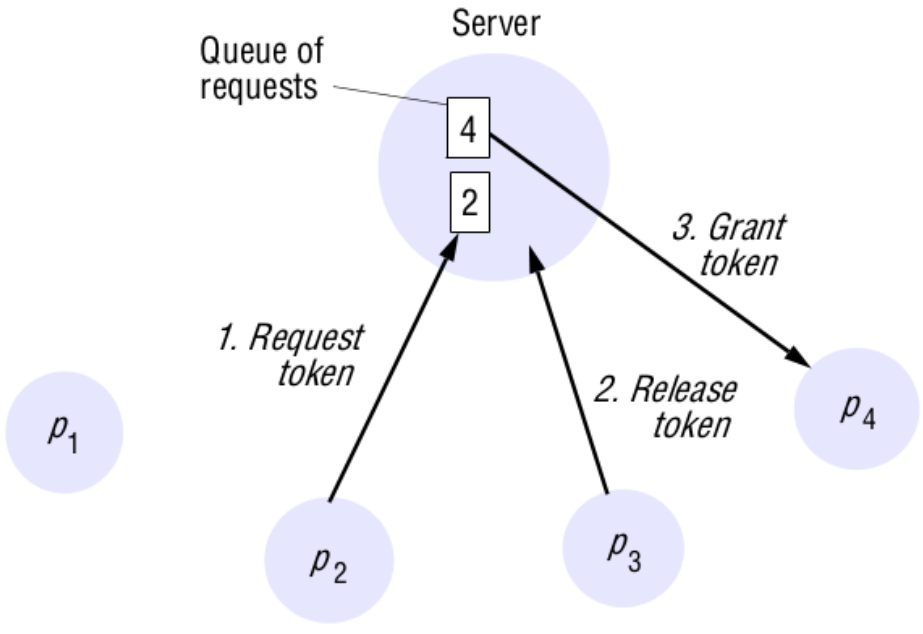
Processes access common resources using critical section:

- enter critical section (CS)
- access shared resources in critical section
- leave critical section – other processes may enter

Requirements for mutual exclusion:

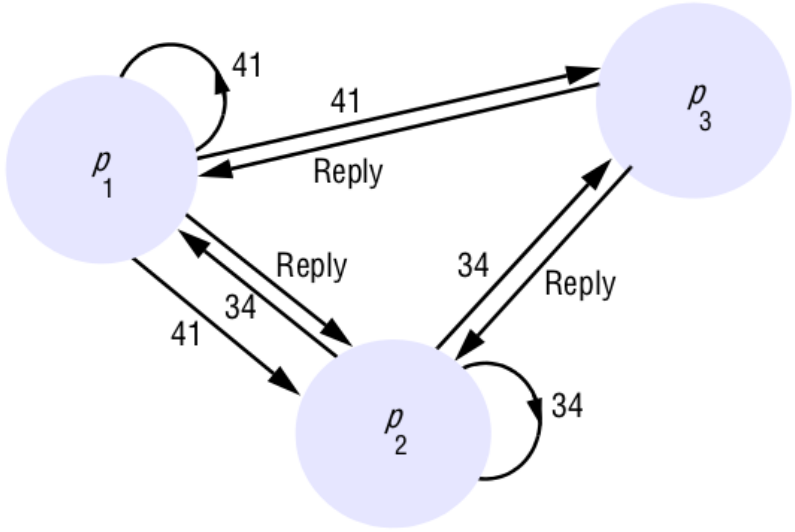
1. ME1 (safety) At most one process may execute inside CS at a time
2. ME2 (liveness) Requests to enter and exit CS eventually succeed
3. ME3 (ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order
    - request = send event

The central server algorithm



- Simple: token is requested, granted and released
- ME3 does not hold, because server does not know if there is happened-before relation between two send events

An algorithm using multicast and logical clocks



- send multicast to others if want to enter CS
  - piggyback your Lamport clock time
  - enter CS when received confirmation from all other processes
- send confirmation only if requester time is less than yours
  - also reply when leaving leaving critical section
  - this way Lamport clock value define the order of entering CS

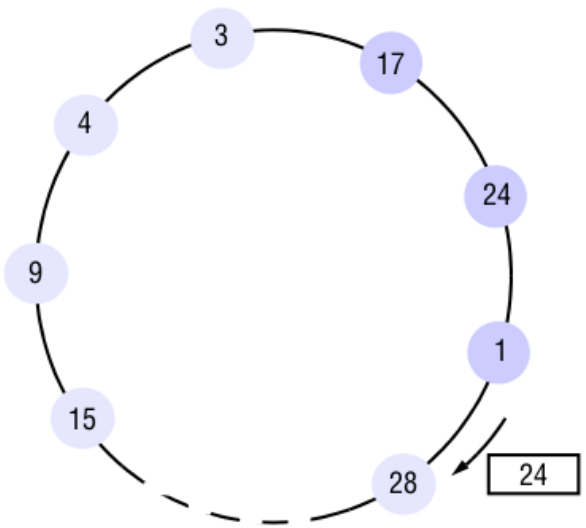
Concepts

- processes choose one *coordinator* process using election algorithm
- any process may start an election
- coordinator must be unique even if several processes started an election concurrently
- could say that a process with highest 'identifier' is elected
  - 'identifier' can be anything like $\langle \frac{1}{\text{load}}, i \rangle$, choosing least loaded process

Requirements:

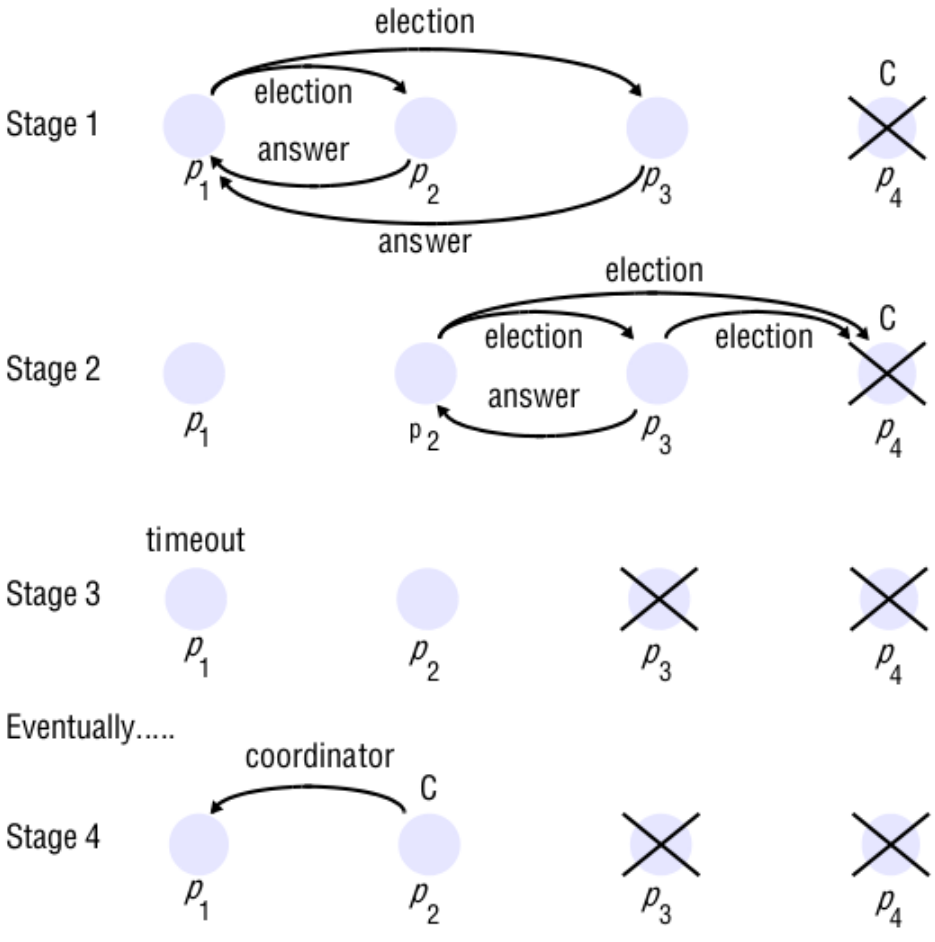
1. E1 (safety) Either nobody is yet elected or the *non-crashed* process with the highest identifier
2. E2 (liveness) All processes participate and agree on the elected, or crash

The ring based algorithm



- a process is either *participant* or *not*
  - initially not
- start election by sending your ID to the ring
  - mark yourself as participant
- receive message from the ring
  - if your ID is smaller – forward
  - it is your ID – finish the election
  - if your ID is larger
    - already particiant – ignore
    - otherwise mark yourself as participant and send your ID to the ring
- finish the election by sending 'elected' message to the ring

The bully algorithm



- see the processes with higher ID
  - send them election message and wait for response
- they reply to "lower" processes ("I'm the boss")
- they start the election themselves
- eventually the process with the highest ID understands "he is the boss" now

Definition of the consensus problem

- Processes need to agree on some value – *concensus*
  - each may propose different candidate for the value
  - e.g. *non-faulty* controllers of spaceship engine must agree on whether to proceed or abort
  - or *non-faulty* generals must agree on whether to attack or retreat
- Processes may exhibit Byzantine (arbitrary) failures
  - go crazy and start sending bad messages to others
  - be hacked and try to fool *non-faulty* processes out of concensus
- Communication is reliable - one-to-one for everyone
  - Nobody can intervene in the communication of others
  - Nobody can see the communication of others
  - No signing of messages: otherwise you could prove others what messages process A sent to you (maybe it is trying to fool everyone)

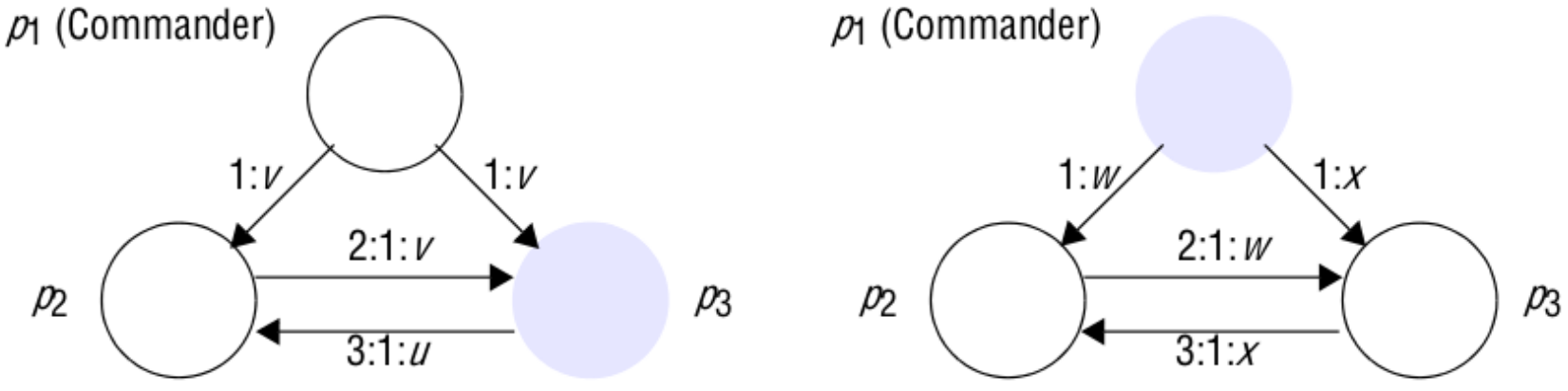
The Byzantine generals problem

- ≥ 3 generals decide on whether to attack or retreat
- one general is the *commander* and he issues the command, others are lieutenants and should follow the order
- generals may be 'treacherous', e.g. bribed by enemy
  - at the end we do not care what decision they take
  - important is that all loyal generals do the same
- commander may also be treacherous and issue different commands to lieutenants!

The requirements are:

1. The decision of all *non-faulty* processes (generals) is the same
2. If the commander is *non-faulty*, then *non-faulty* lieutenants must follow his order

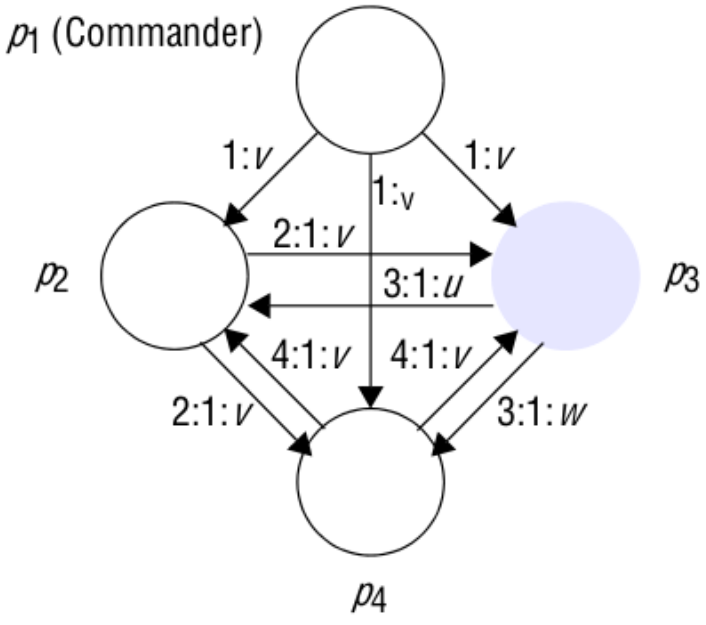
The BG problem: impossibility with 3 generals

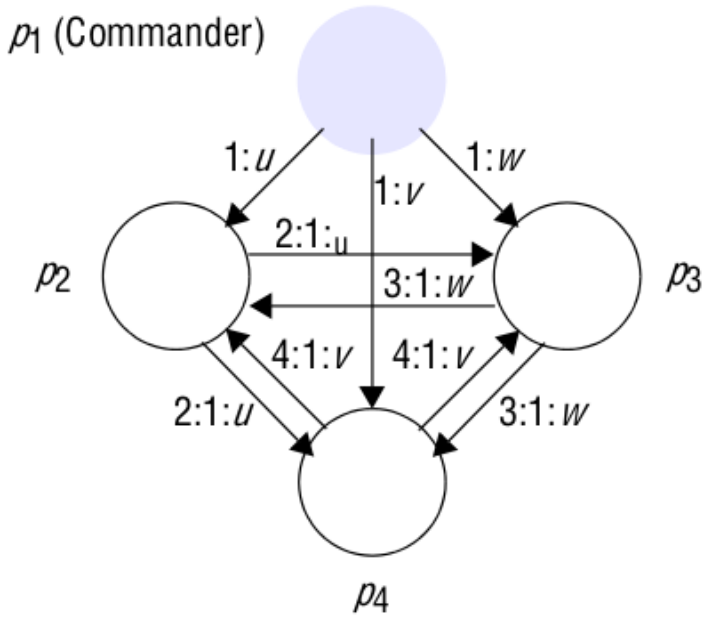


Faulty processes are shown in grey

- the algorithm proceeds in 2 steps
  1. the commander (p1) sends lieutenants some commands
  2. lieutenants (p2 and p3) exchange the commands that the commander sent them
- if p2 receives the same value, it follows the order
  - it does not matter whether p1 or p3 is faulty – the requirements are satisfied
- if p3 receives different values, it cannot figure out who is faulty

The BG problem: solution with 4 generals





- the algorithm proceeds in similar way as with 3 generals
- non-faulty lieutenants make decision based on majority of votes they receive (2 or 3 out of 3)
  - no matter who is faulty the requirements are satisfied
  - easy proof: if the commander is faulty everyone receives the same set of votes; faulty lieutenant cannot frustrate the others

Outline

1. Time and global state
   - Physical clocks
   - Logical clocks
   - Global state
2. Coordination and agreement
   - Distributed mutual exclusion
   - Elections
   - Consensus
3. Distributed transactions
   - Transactions
   - Two-phase commit

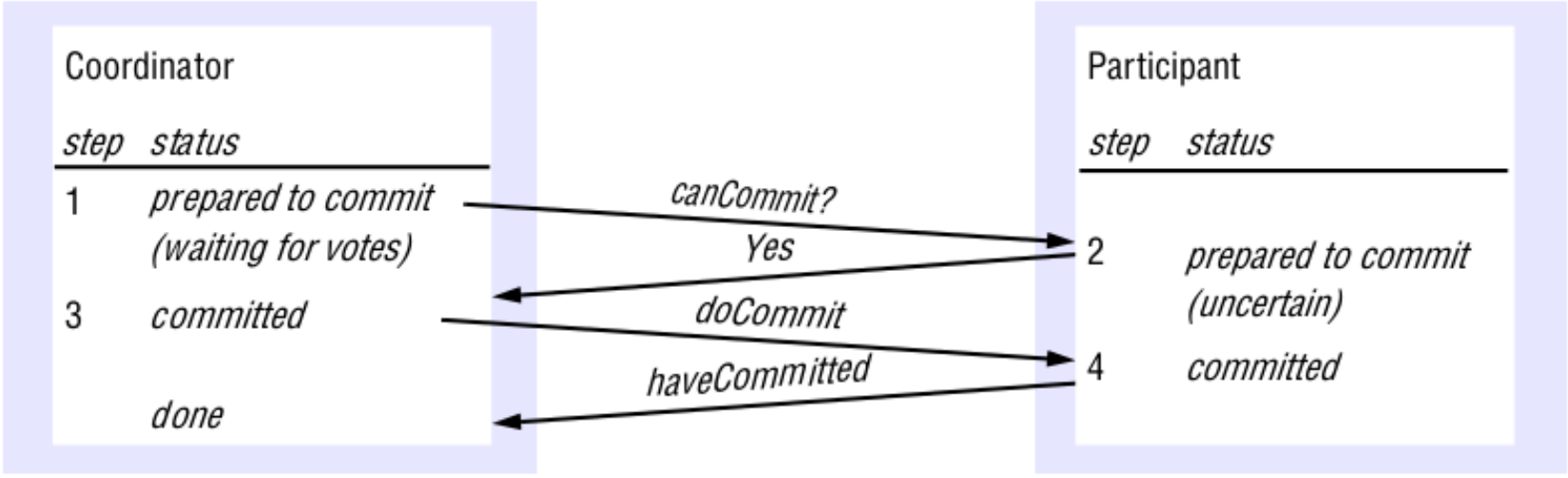
Distributed transactions

There are values on several servers, databases, ...
Even if processes or network fail, we want to be sure about ACID properties:

1. we change them all or none at all (Atomicity)
2. they are 'good' (Consistency)
3. when values are being changed no other process intervenes (Isolation)
4. new values are permanently stored (Durability)

Two-phase commit protocol deals with 1 and 4. It is classical and famous.

Two-phase commit



1. coordinator asks if everyone is OK to change their local value
2. participants decide and, if yes, store the required steps into permanent storage
   - no values are yet changed
   - it is just they can do it if needed even after failure and restart
3. coordinator stores the final decision, it cannot be undone
4. just make actions final, that were prepared in step 2