ECS656U/ECS796P
Distributed Systems

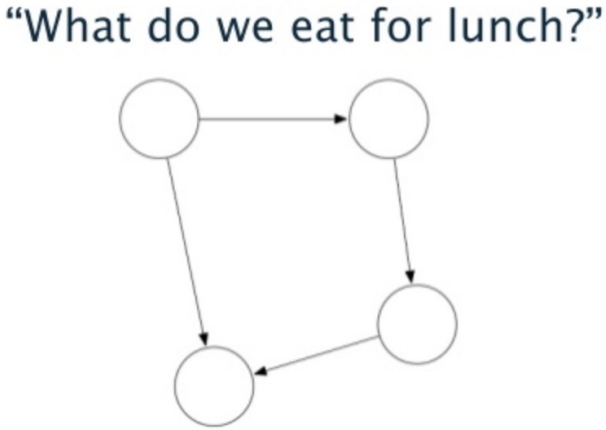
What this lecture is about

Distributed consensus algorithms

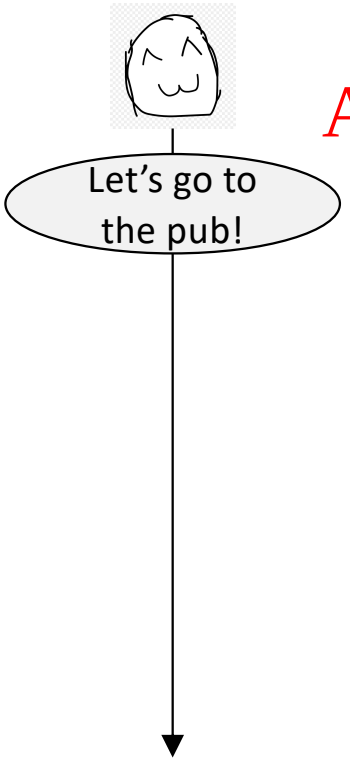
- Introduction to consensus algorithms
- Paxos

What is that?

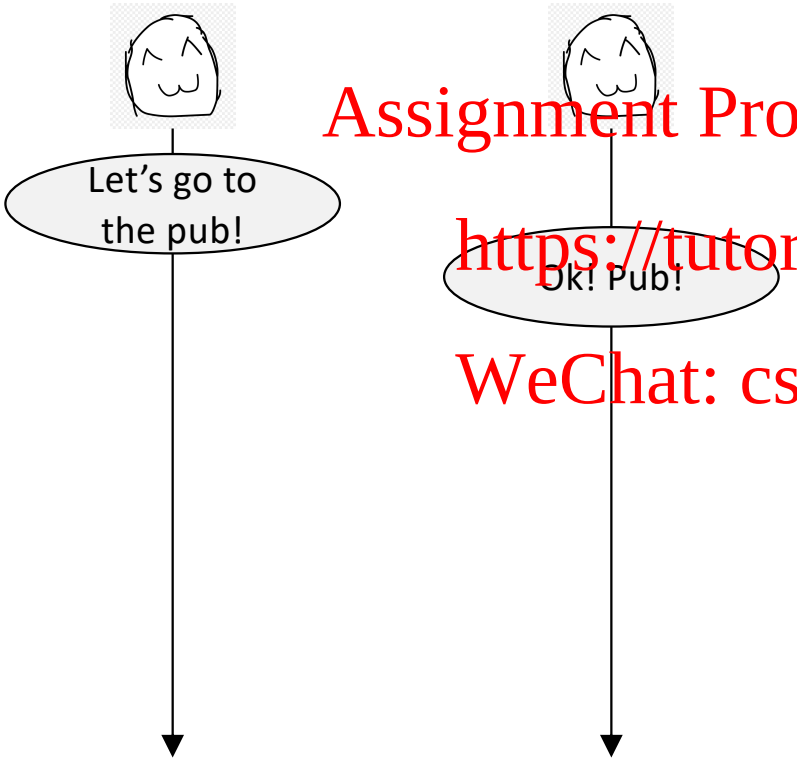
Distributed consensus algorithms deal with reaching agreement among a group of processes connected by an unreliable communications network.



What is that?

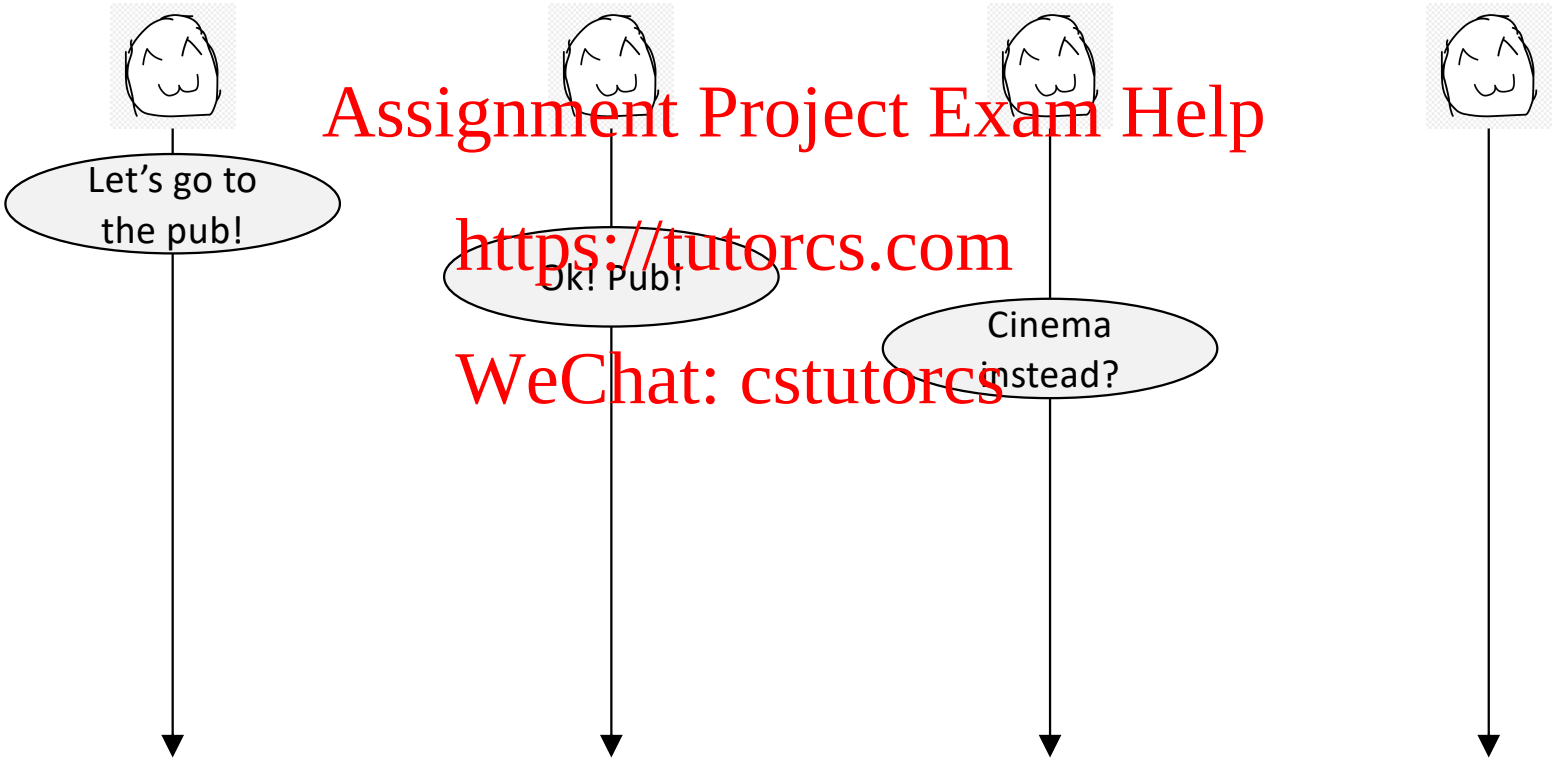


What is that?

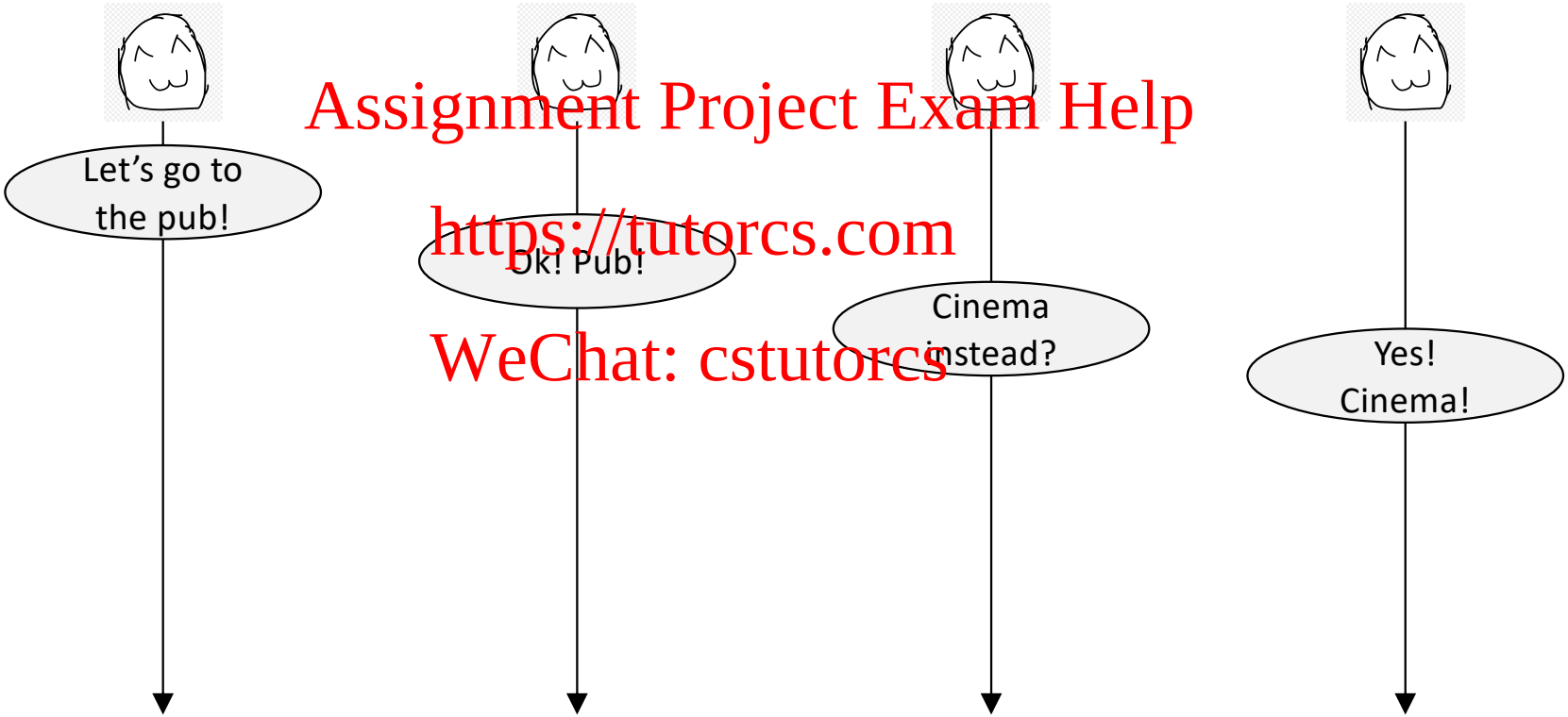


What is that?

Meanwhile, someone that got the message delayed, or was not listening…

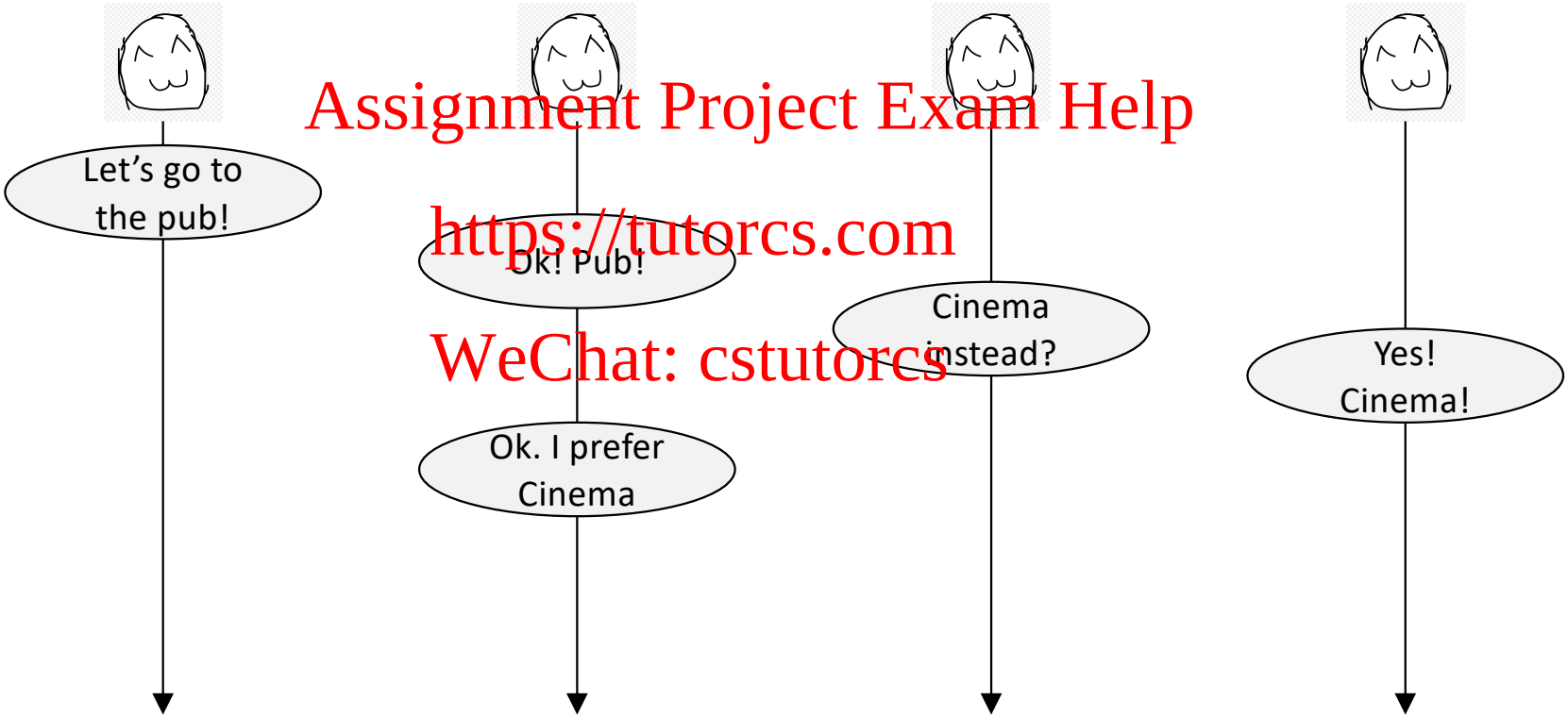


What is that?



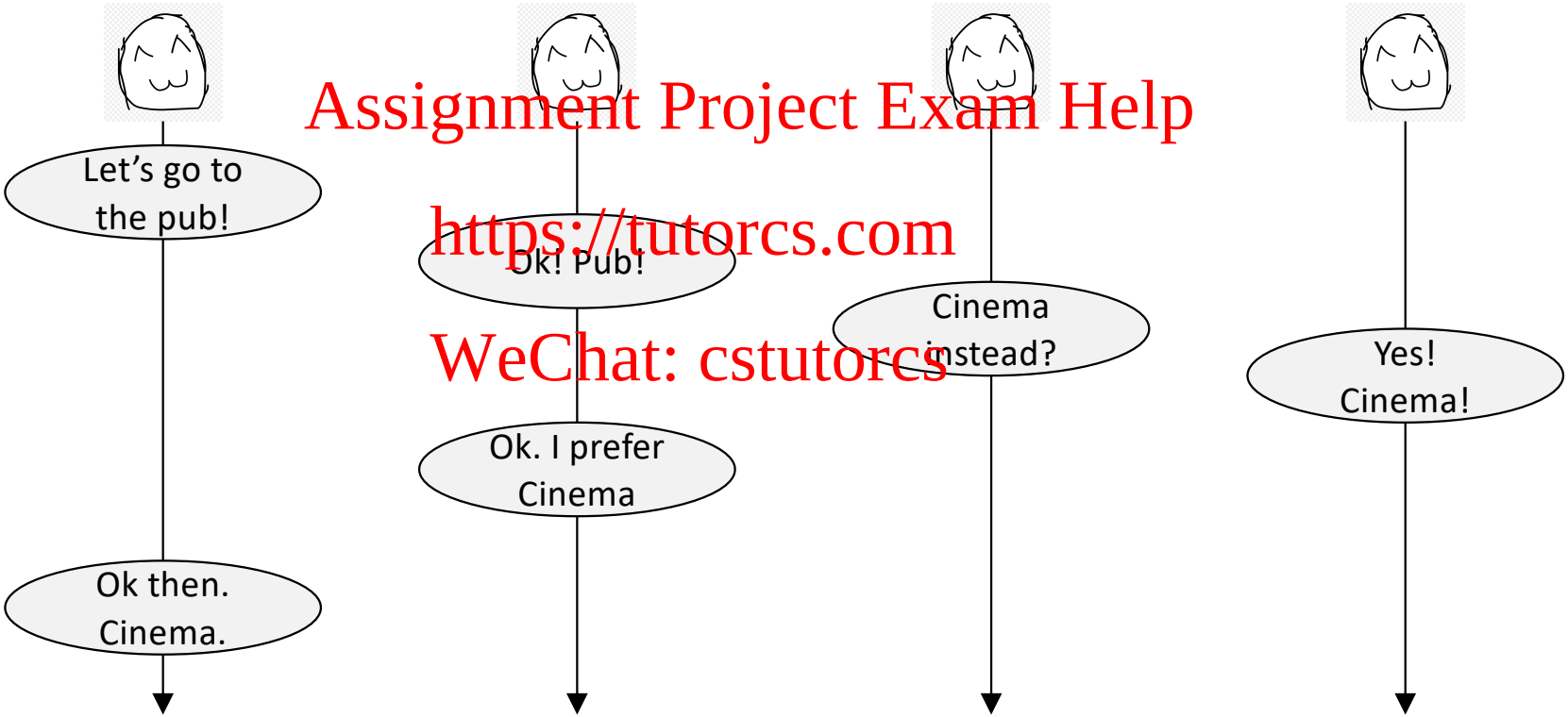
What is that?

Someone needs to back off.



What is that?

We are in democracy. Majority wins.



So, what is consensus?

- Consensus is agreeing on **one** result
- Once a **majority** agrees on a proposal, that is the consensus
- The reached consensus can be **eventually** known by everyone
- The involved parties want to agree on **any result**, not on their proposal
- Communication channels may be **faulty**, that is, messages can get lost

Why do we care?

Example: You transfer money from one person to another. The money went out one account but never arrived the other one because a server or the network itself somewhere had failed.

Why do we care?

We need to be able to **reliably reach agreement** even though there are **failures**

A bit of history

- 1985: FLP (Fisher-Lynch-Patterson) impossibility paper.

we cannot guarantee agreement in an asynchronous system where even one host might fail

The problem of consensus was known to be solvable in a synchronous setting, where processes could proceed in simultaneous steps

- The synchronous solution was resilient to faults: you can easily detect them!

A bit of history

- 1985: FLP (Fisher-Lynch-Patterson) impossibility paper.

*we cannot guarantee agreement in an **asynchronous** system where even one host might fail*

Asynchronous means:

- No upper bound on processing time
- No upper bound on clock drift rate
- No upper bound on networking delay

A bit of history

- 1985: FLP (Fisher-Lynch-Patterson) impossibility paper.

*we cannot guarantee agreement in an **asynchronous** system where even one host might fail.*

Why?

We cannot detect reliably failures. We cannot know for sure the difference between a slow host/network and a failed host

A bit of history

- 1985: FLP (Fisher-Lynch-Patterson) impossibility paper.

*we cannot guarantee agreement in an **asynchronous** system where even one host might fail*

The FLP result shows that in an asynchronous setting, where only one processor might crash, there is no distributed algorithm that solves the consensus problem

A bit of history – cont'd

- 1985: FLP impossibility paper
- 1989: Lamport: The Part-Time Parliament
- Distributed Systems are getting more important, thanks to Internet

How? (a bit of history)

- 1985: FLP impossibility paper
- 1989: Lamport: The Part-Time Parliament
- Distributed Systems are getting more important, thanks to Internet

Let's remember the original WHY we could not guarantee agreement:
We cannot detect reliably **failures**.

Can we categorize failures?

Two types of failures

- *Non-Byzantine*
- Failed nodes stop communicating with other nodes
  - "Clean" failure
  - *Fail-stop* behavior

- *Byzantine*
- Failed nodes will keep sending messages
  - Incorrect and potentially misleading
  - Failed node becomes a *traitor*

Two types of failures

- *Non-Byzantine*
- Failed nodes stop communicating with other nodes
  - "Clean" failure
  - *Fail-stop* behavior

- *Byzantine*
- Failed nodes will keep sending messages
  - Incorrect and potentially misleading
  - Failed node becomes a *traitor*

They are defined as arbitrary deviations of a process from its assumed behavior, e.g., software bug, a hardware malfunction, or a malicious attack.

Two types of failures

- *Non-Byzantine*
- Failed nodes stop communicating with other nodes
  - "Clean" failure
  - *Fail-stop* behavior

- *Byzantine*
- Failed nodes will keep sending messages
  - Incorrect and potentially misleading
  - Failed node becomes a *traitor*

Assumption: asynchronous, non-byzantine model

The goal for consensus

- We want agreement between processes (mutable states)
- Processes are concurrent, asynchronous and failure-prone

Recap

- Consensus: making a decision (liveness) which is also correct (safety)

Liveness property

- Liveness: guarantee that something good will happen
- Examples:
  - Real world: “at least one of the athletes in the 100m final will win gold” is liveness
  - Consensus: All processes decide a value

Safety property

- **Safety**: guarantee that something bad will never happen
- **Examples**:
  - Real world: A peace treaty between nations provide safety as war will never happen
  - Consensus: no two processes decide on different value

Paxos

Many slides from Ion Stoica presentation:
(https://ucbrise.github.io/cs262a-spring2018/)

Paxos

- Paxos is a family of protocols for solving consensus in a network of unreliable processors

Paxos

- Paxos is a family of protocols for solving consensus in a network of unreliable processors

  - *Basic Paxos*
  - *Multi-Paxos*
  - *Cheap Paxos*
  - *Fast Paxos*
  - *Byzantine Paxos*
  - *Generalized Paxos*

Paxos

- Paxos is a family of protocols for solving consensus in a network of unreliable processors

  - *Basic Paxos*
  - *Multi-Paxos*
  - *Cheap Paxos*
  - *Fast Paxos*
  - *Byzantine Paxos*
  - *Generalized Paxos*

Basic Paxos

- Laslie Lamport (one of the initial core developer of LaTeX!!!)

Basic Paxos

- Laslie Lamport (one of the initial core developer of LaTeX!!!)
- Deterministic and fault tolerant consensus protocol

Basic Paxos

- Laslie Lamport (one of the initial core developer of LaTeX!!!)
- Deterministic and fault tolerant consensus protocol
- Named after a Greek Island

(*taken from the example Lamport carries on his paper about elections in the island*)

Basic Paxos

- Laslie Lamport (one of the initial core developer of LaTeX!!!)
- Deterministic and fault tolerant consensus protocol
- Named after a Greek Island
- Guarantees consistent results

Does Paxos solve consensus?

- Provides **safety** and **eventual liveness**

- Safety:
  - Only a value which has been proposed can be chosen
  - Only a single value can be chosen
  - A process never learns a value unless it was chosen

- Eventual liveness:
  - If things go well, at some point in the future, consensus is eventually reached. However, this is not guaranteed.

Does Paxos solve consensus?

- FLP result still applies: Paxos is not **guaranteed** to reach consensus
- There is **NO** time bound
- We talk about **eventual** liveness

So simple, so obvious

"In fact, it is among the simplest and most obvious of distributed algorithms."

- Leslie Lamport

Simple pseudocode

outcome[*p*] The decree written in *p*'s ledger, or BLANK if there is nothing written there yet.

lastTried[*p*] The number of the last ballot that *p* tried to begin, or $-\infty$ if there was none.

prevBal[*p*] The number of the last ballot in which *p* voted, or $-\infty$ if he never voted.

prevDec[*p*] The decree for which *p* last voted, or BLANK if *p* never voted.

nextBal[*p*] The number of the last ballot in which *p* agreed to participate, or $-\infty$ if he has never agreed to participate in a ballot.

Next come variables representing information that priest *p* could keep on his slip of paper:

status[*p*] One of the following values:
idle Not conducting or trying to begin a ballot
trying Trying to begin ballot number *lastTried*[*p*]
polling Now conducting ballot number *lastTried*[*p*].
If *p* has lost his slip of paper, then *status*[*p*] is assumed to equal *idle* and the values of the following four variables are irrelevant.

prevVotes[*p*] The set of votes received in *LastVote* messages for the current ballot (the one with ballot number *lastTried*[*p*]).

quorum[*p*] If *status*[*p*] = *polling*, then the set of priests forming the quorum of the current ballot; otherwise, meaningless.

voters[*p*] If *status*[*p*] = *polling*, then the set of quorum members from whom *p* has received *Voted* messages in the current ballot; otherwise, meaningless.

decree[*p*] If *status*[*p*] = *polling*, then the decree of the current ballot; otherwise, meaningless.

Try New Ballot

Always enabled.

- Set *lastTried*[*p*] to any ballot number *b*, greater than its previous value, such that *owner*(*b*) = *p*.
- Set *status*[*p*] to *trying*.
- Set *prevVotes*[*p*] to $\emptyset$.

Send *NextBallot* Message

Enabled whenever *status*[*p*]=*trying*.

- Send a *NextBallot*(*lastTried*[*p*]) message to any priest.

Receive *NextBallot*(*b*) Message

If $b \geq$ *nextBal*[*p*] then

- Set *nextBal*[*p*] to *b*.

Send *LastVote* Message

Enabled whenever *nextBal*[*p*] > *prevBal*[*p*].

- Send a *LastVote*(*nextBal*[*p*], *v*) message to priest *owner*(*nextBal*[*p*]), where $v_{pst} = p$, v_{bal} = *prevBal*[*p*], and v_{dec} = *prevDec*[*p*].

Receive *LastVote*(*b*, *v*) Message

If *b* = *lastTried*[*p*] and *status*[*p*] = *trying*, then

- Set *prevVotes*[*p*] to the union of its original value and {*v*}.

Start Polling Majority Set *Q*

Enabled when *status*[*p*] = *trying* and $Q \subseteq \{v_{pst} : v \in prevVotes[p]\}$, where *Q* is a majority set.

- Set *status*[*p*] to *polling*.
- Set *quorum*[*p*] to *Q*.
- Set *voters*[*p*] to $\emptyset$.
- Set *decree*[*p*] to a decree *d* chosen as follows: Let *v* be the maximum element of *prevVotes*[*p*]. If $v_{bal} \neq -\infty$ then $d = v_{dec}$, else *d* can equal any decree.
- Set $\mathcal{B}$ to the union of its former value and $\{B\}$, where $B_{dec} = d$, $B_{qrm} = Q$, $B_{vot} = \emptyset$, and B_{bal} = *lastTried*[*p*].

Send *BeginBallot* Message

Enabled when *status*[*p*] = *polling*.

- Send a *BeginBallot*(*lastTried*[*p*], *decree*[*p*]) message to any priest in *quorum*[*p*].

Receive *BeginBallot*(*b*, *d*) Message

If *b* = *nextBal*[*p*] > *prevBal*[*p*] then

- Set *prevBal*[*p*] to *b*.
- Set *prevDec*[*p*] to *d*.
- If there is a ballot B in $\mathcal{B}$ with $B_{bal} = b$ [there will be], then choose any such B [there will be only one] and let the new value of $\mathcal{B}$ be obtained from its old value by setting B_{vot} equal to the union of its old value and $\{p\}$.

Send *Voted* Message

Enabled whenever *prevBal*[*p*] $\neq -\infty$.

- Send a *Voted*(*prevBal*[*p*], *p*) message to *owner*(*prevBal*[*p*]).

Receive *Voted*(*b*, *q*) Message

If *b* = *lastTried*[*p*] and *status*[*p*] = *polling*, then

- Set *voters*[*p*] to the union of its old value and {*q*}

Succeed

Enabled whenever *status*[*p*] = *polling*, *quorum*[*p*] $\subseteq$ *voters*[*p*], and *outcome*[*p*] = BLANK.

- Set *outcome*[*p*] to *decree*[*p*].

Send *Success* Message

Enabled whenever *outcome*[*p*] $\neq$ BLANK.

- Send a *Success*(*outcome*[*p*]) message to any priest.

Receive *Success*(*d*) Message

If *outcome*[*p*] = BLANK, then

- Set *outcome*[*p*] to *d*.

A political analogy

- A part-time parliament

A political analogy

- Each round
  - Phase 1: A leader is elected (election)
  - Phase 2: Leader proposes a value (bill), processes acks
  - Phase 3: Leader multicast final value (law)

Agents

- Three types of roles
  - Proposer: It receives a request from the client, and attempts to get a quorum of acceptors to agree on it
  - Acceptor: It is a participant in the maintenance of the distributed storage. A state change in a Paxos cluster does not occur until a majority (quorum) of acceptors agree upon it
  - Learner: It learns the agreed upon value. They can be later queried to know what the consensus value was

In practice..

- Paxos nodes **can take** multiple roles, even all of them
  - A single node can send proposals to other nodes, they can contribute to reaching consensus and they learn the final agreed upon value
- Paxos nodes **must know** how many acceptors a majority is

In practice..

- Paxos nodes **must be persistent**: they cannot forget what they accepted
  - Even if the communication channel is faulty, they cannot forget

- A Paxos run aims at reaching a **single consensus**
  - Once a consensus is reached, it cannot progress to another consensus
  - In order to reach another consensus, a different Paxos run must happen

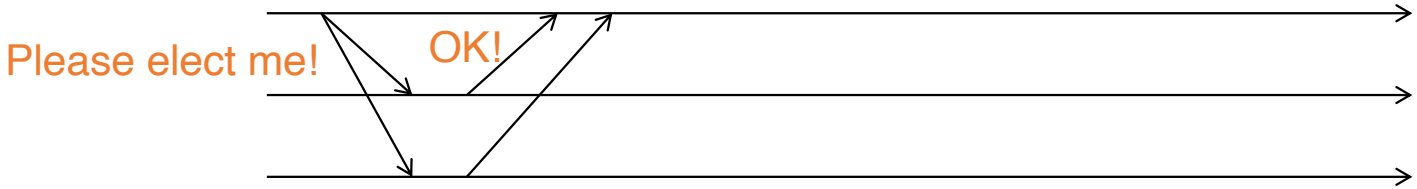
Recall..

- Rounds are asynchronous
  - Time synchronization not required
  - If you are in round j and hear a message from round j+1, abort everything and move to round j+1

- Each round consists of three phases
  - Phase 1: A leader is elected (Election)
  - Phase 2: Leader proposes a value, processes acks (Bill)
  - Phase 3: Leader multicasts final value (Law)

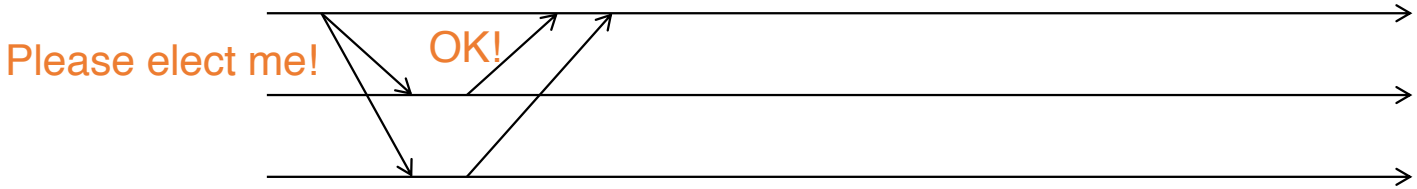
Phase 1 – Election

- Potential leader chooses a unique ballot ID, higher than anything it has seen so far
- Sends ballot ID to all processes
- Processes respond to highest ballot ID



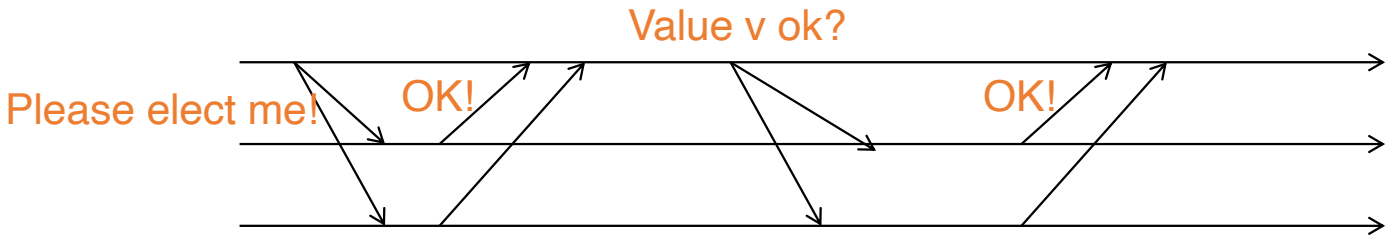
Phase 1 – Election

- If majority (i.e., quorum) respond OK then you are the leader
  - If no one has majority, start new round



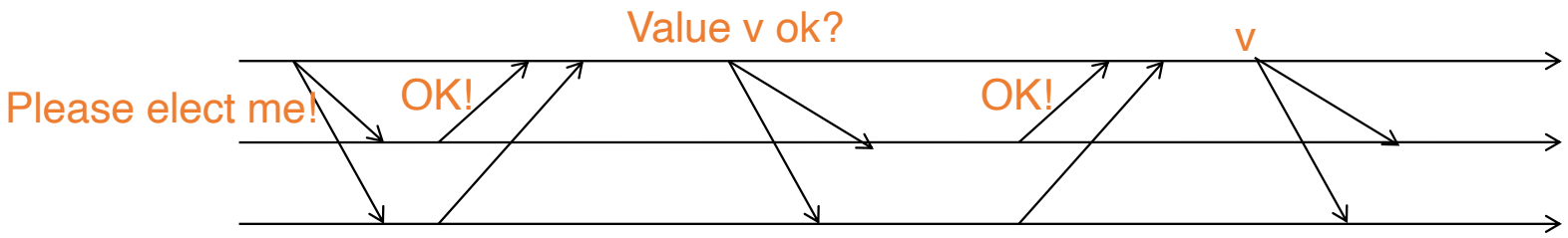
Phase 2 – Proposal (Bill)

- Leader sends proposal value v to all
  - Use v=v’ if some process already decided in a previous round and sent you its decided value v’
  - Otherwise propose its own value
- Recipient log on disk, and responds OK

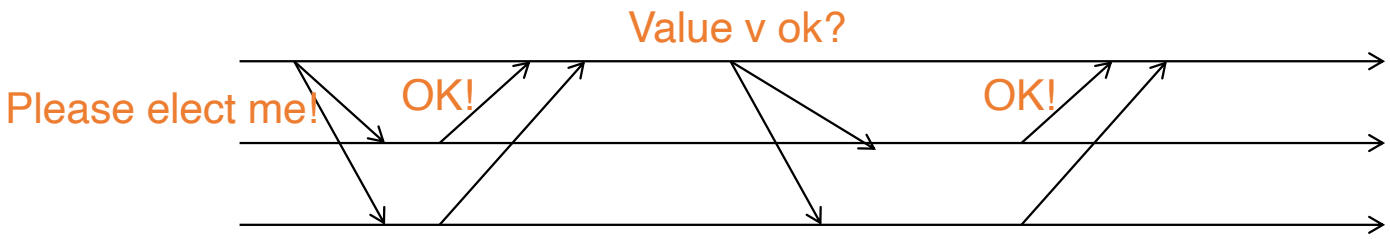


Phase 3 – Decision (Law)

- If leader hears OKs from majority, it lets everyone know of the decision
- Recipients receive decisions, log it on disk

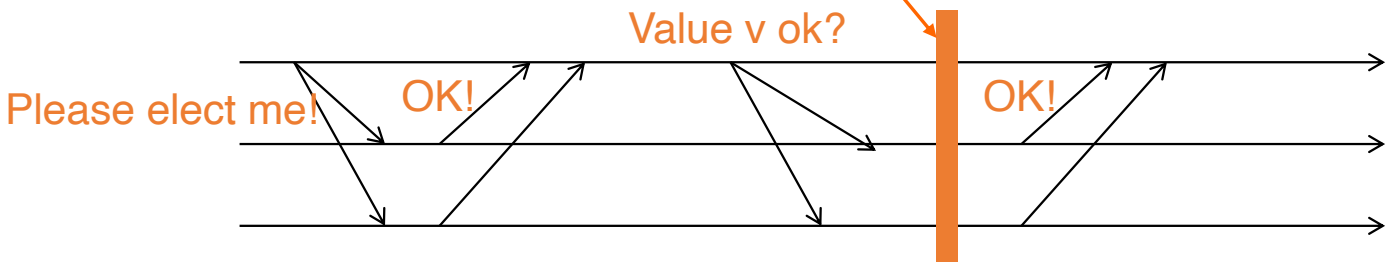


When is Consensus Achieved?



When is Consensus Achieved?

- When a majority of processes hear proposed value and accept it:

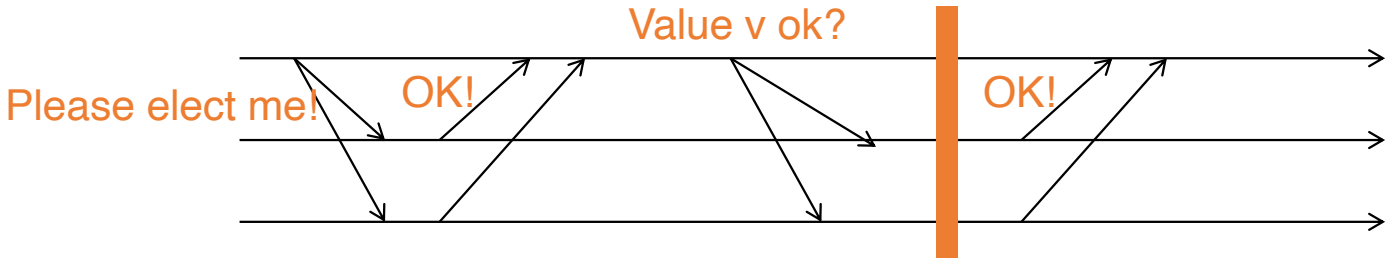


When is Consensus Achieved?

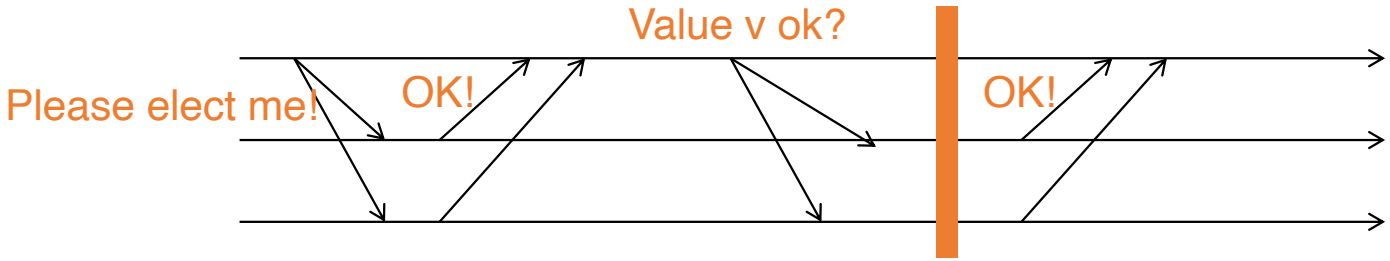
- When a majority of processes hear proposed value and accept it:
  - Are about to respond (or have responded) with OK!

- At this point decision has been made even though
  - Processes or even leader may not know!

- What if leader fails after that?

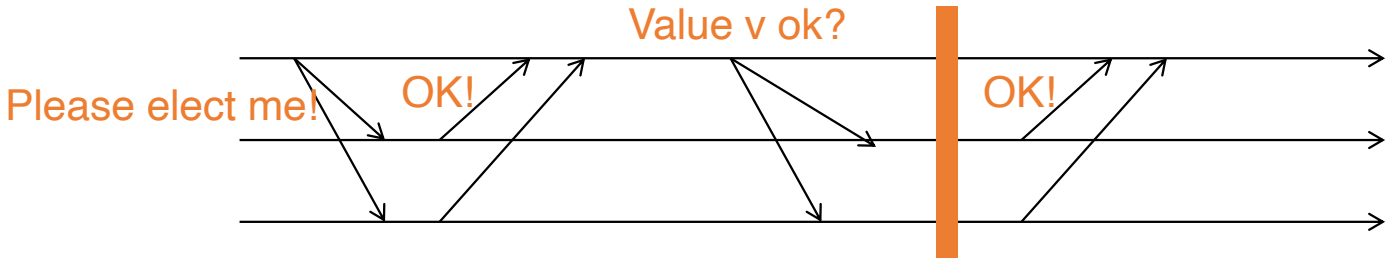


Easy right? ☺



Easy right? ☺

- Let's have a look in more details now…



Basic Paxos Protocol

Phase 1a: **"Prepare"**
Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

* = record to stable storage

Basic Paxos Protocol

Phase 1a: **"Prepare"**
Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

Phase 1b: **"Promise"**
If N > *number of any previous promises or acceptances*,
* promise to never accept any future proposal less than *N*,
- send a ***promise(N, U)*** response
(where *U* is the highest-numbered proposal accepted so far (if any))

Acceptor

* = record to stable storage

Basic Paxos Protocol

Phase 1a: **"Prepare"**
Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

Phase 1b: **"Promise"**
If N > *number of any previous promises or acceptances*,
* promise to never accept any future proposal less than *N*,
- send a ***promise(N, U)*** response
(where *U* is the highest-numbered proposal accepted so far (if any))

Phase 2a: **"Accept!"**
If proposer received promise responses from a quorum,
- send an ***accept(N, W)*** request to those acceptors
(where ***W*** is the value of the highest-numbered proposal among the ***promise*** responses, or any value if no ***promise*** contained a proposal)

Acceptor

* = record to stable storage

Basic Paxos Protocol

Phase 1a: **"Prepare"**
Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

Phase 1b: **"Promise"**
If N > *number of any previous promises or acceptances*,
* promise to never accept any future proposal less than *N*,
- send a ***promise(N, U)*** response
(where *U* is the highest-numbered proposal accepted so far (if any))

Phase 2a: **"Accept!"**
If proposer received promise responses from a quorum,
- send an ***accept(N, W)*** request to those acceptors
(where ***W*** is the value of the highest-numbered proposal among the ***promise*** responses, or any value if no ***promise*** contained a proposal)

Acceptor

Phase 2b: **"Accepted"**
If N >= *number of any previous promise*,
* accept the proposal
- send an ***accepted*** notification to the learner

* = record to stable storage

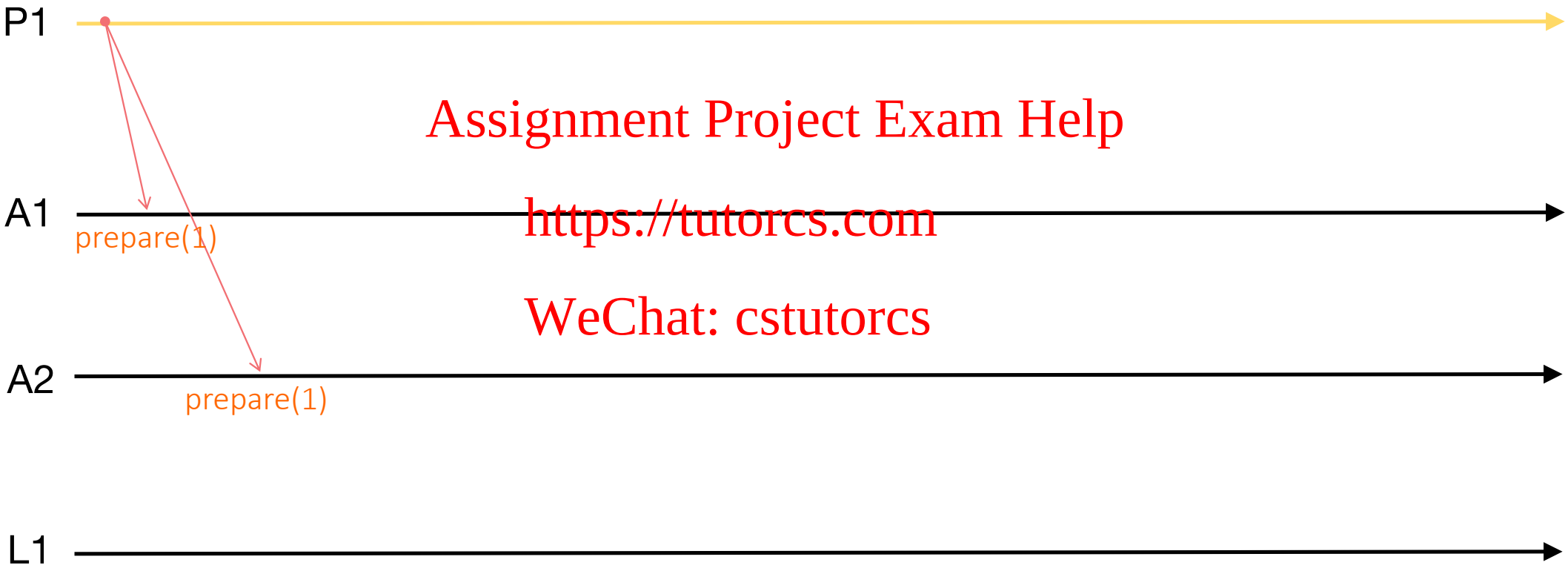
Milestones

- If the majority of acceptors promise, no ID < IDp can make it through

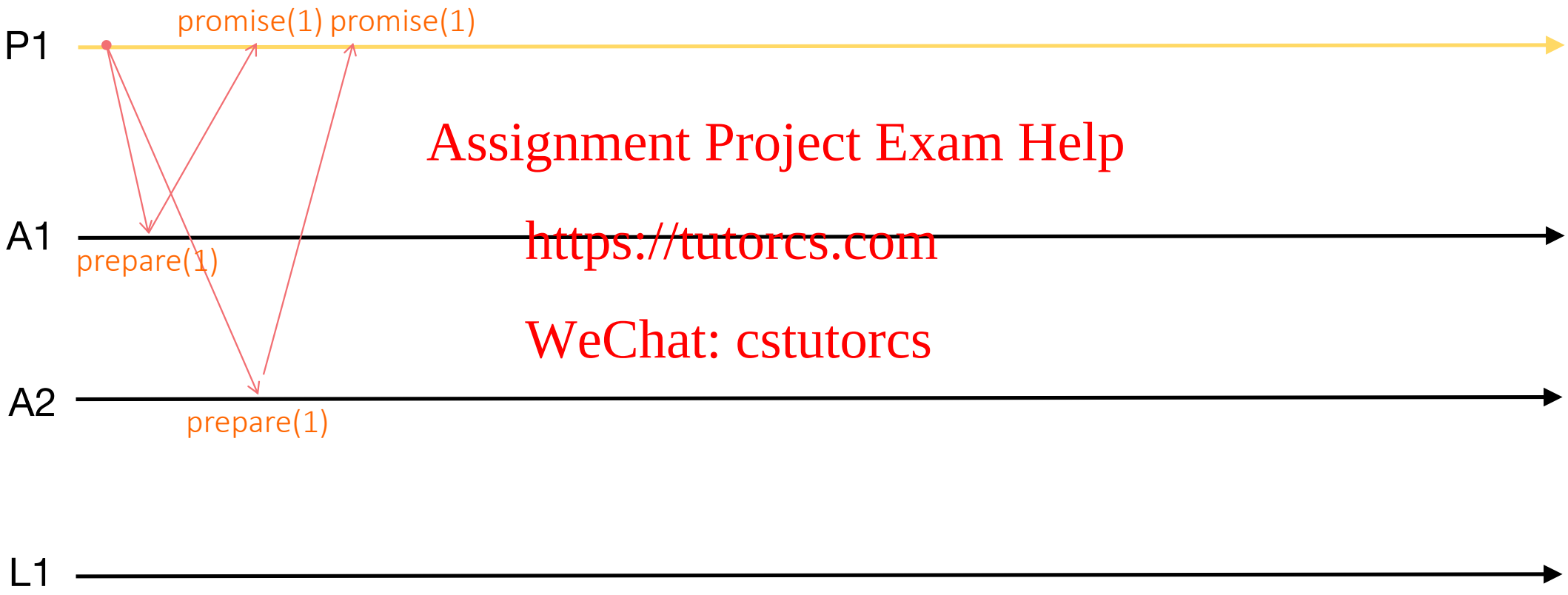
- If a majority of acceptors accept (IDp value), consensus is reached. Consensus is and will always be on a **value**

- If a proposer/learner gets the majority of accept for a specific IDp, they know that consensus has been reached on a **value**

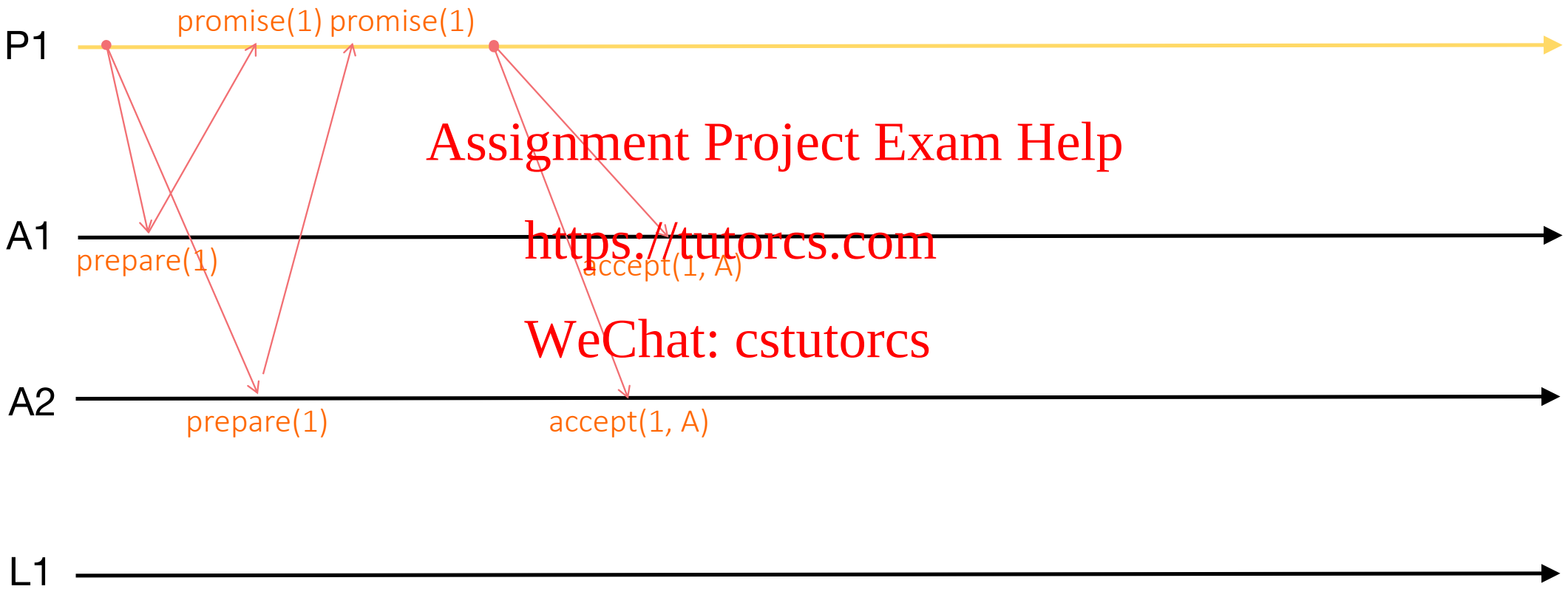
Time 0: P1 wants to propose “A”



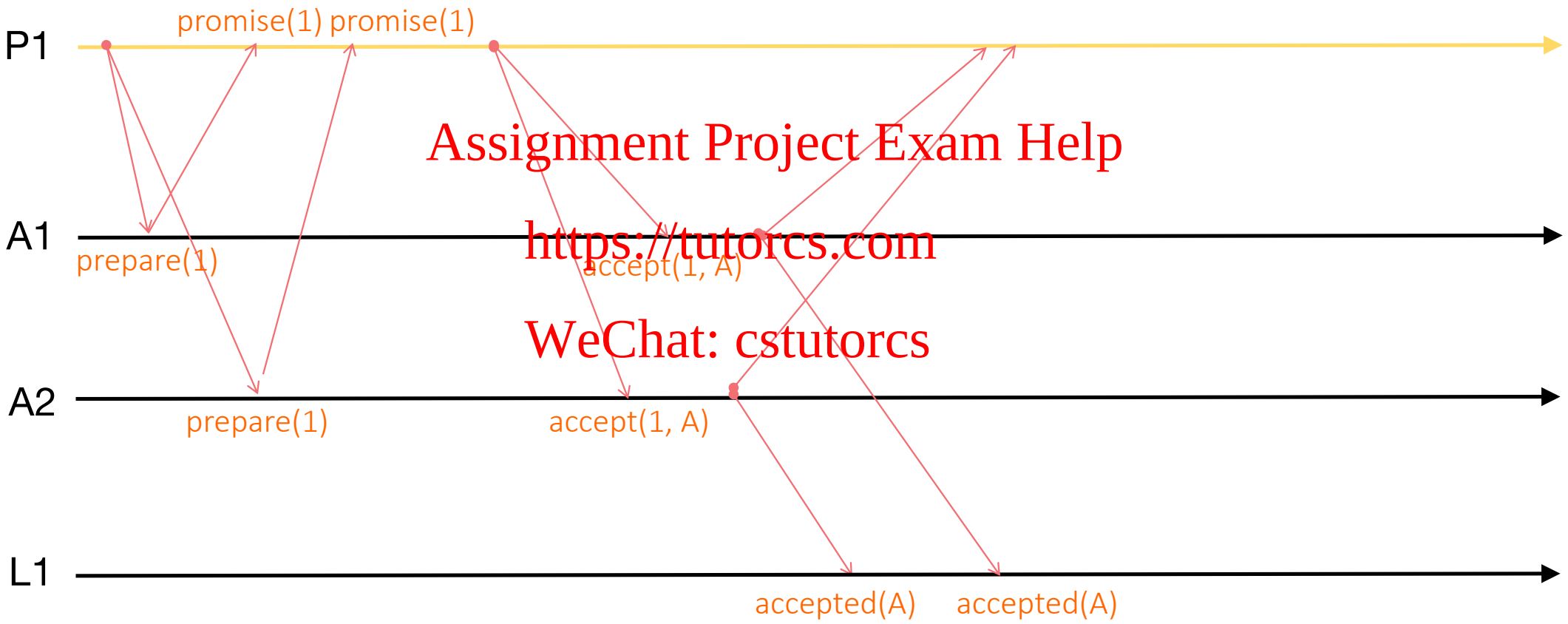
Time 0: P1 wants to propose “A”



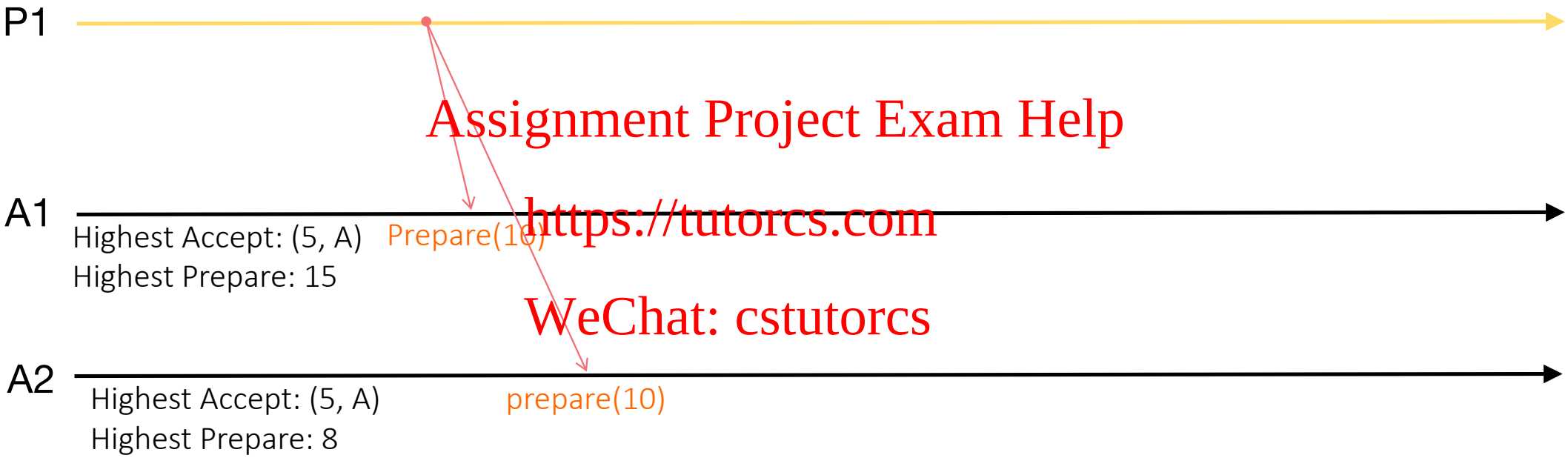
Time 0: P1 wants to propose “A”



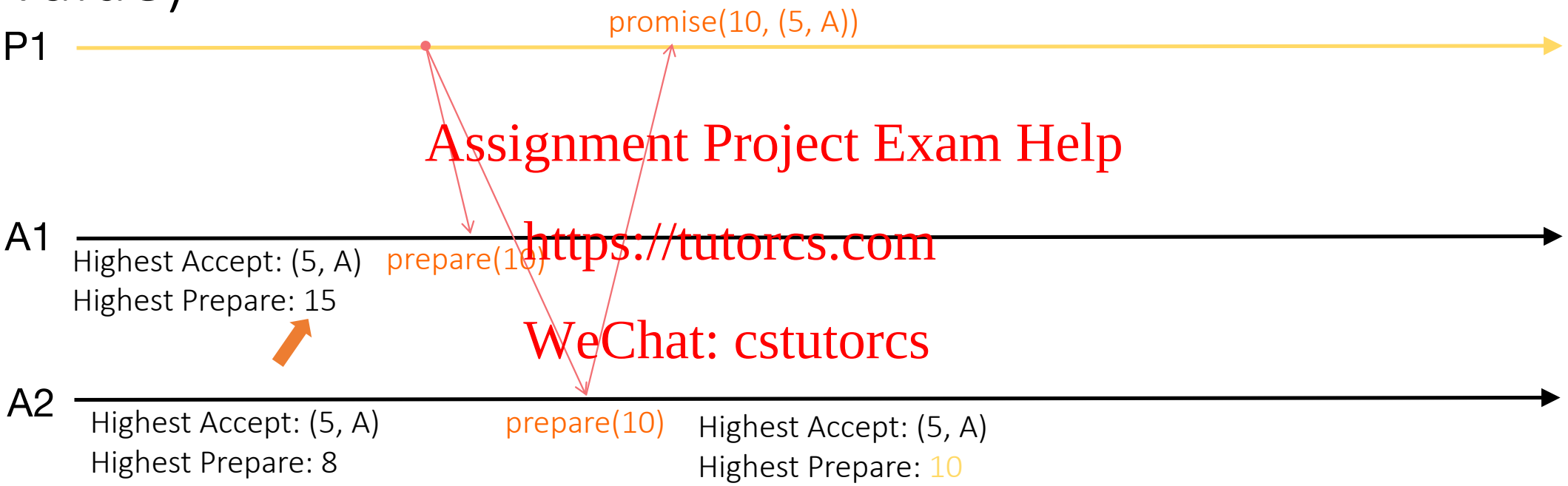
Time 0: P1 wants to propose “A”



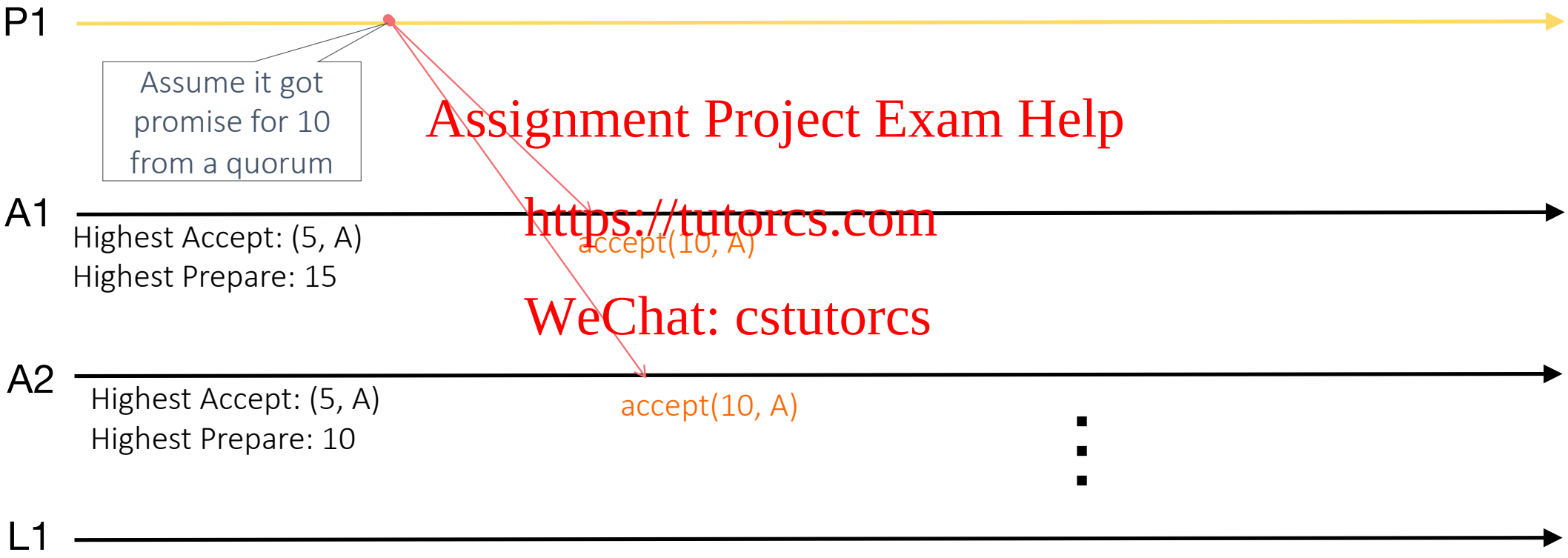
If the acceptor has accepted something before..



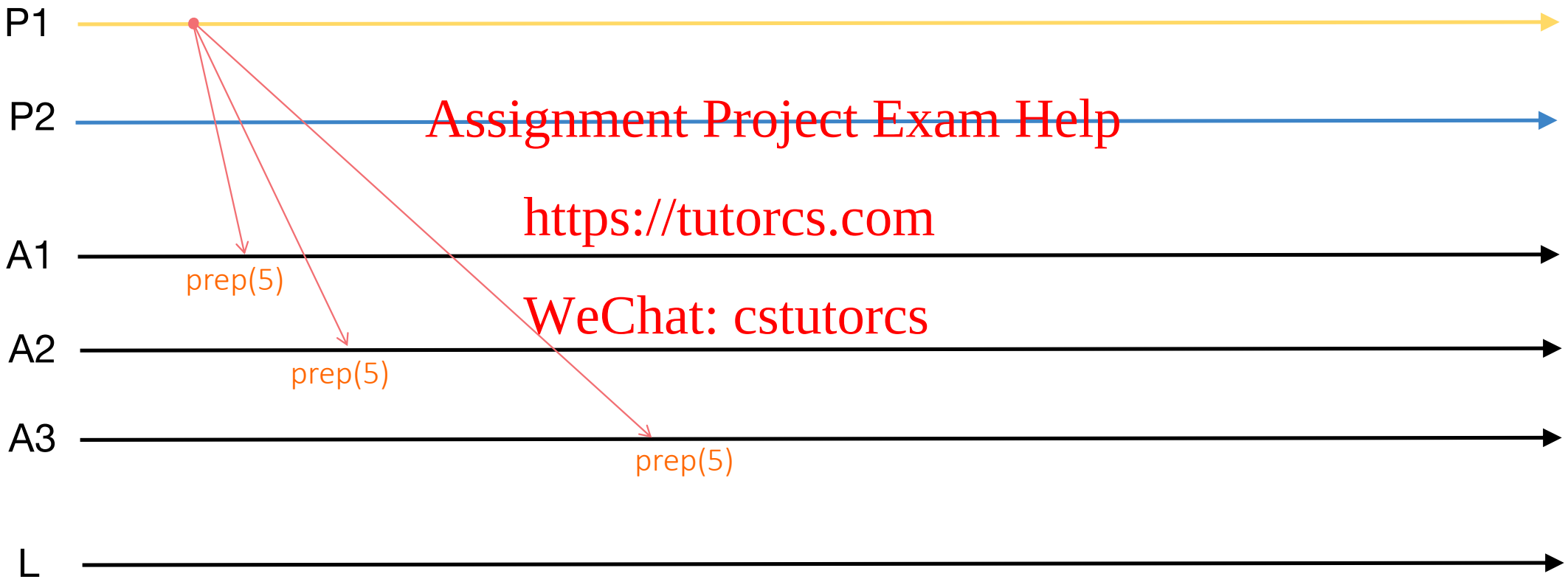
It needs to reply with PROMISE ID and (accepted ID, value)



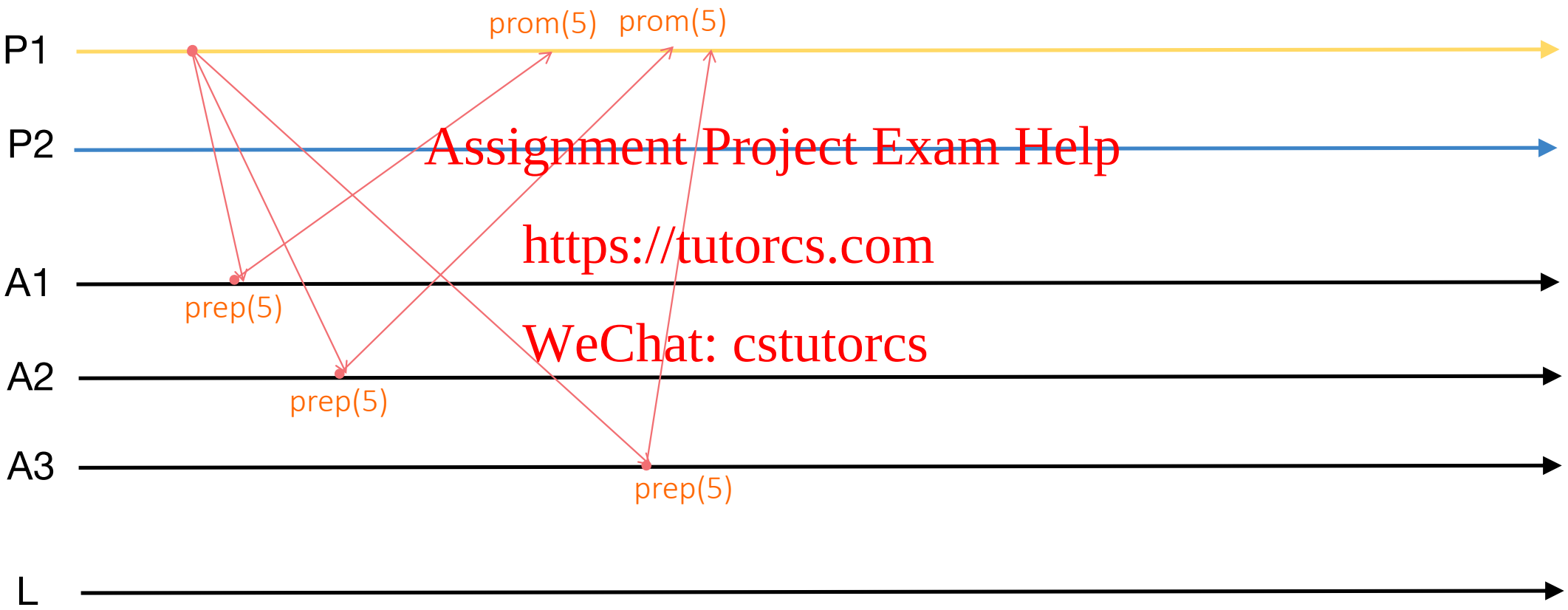
The proposer needs to pick the value with the highest ID that it got



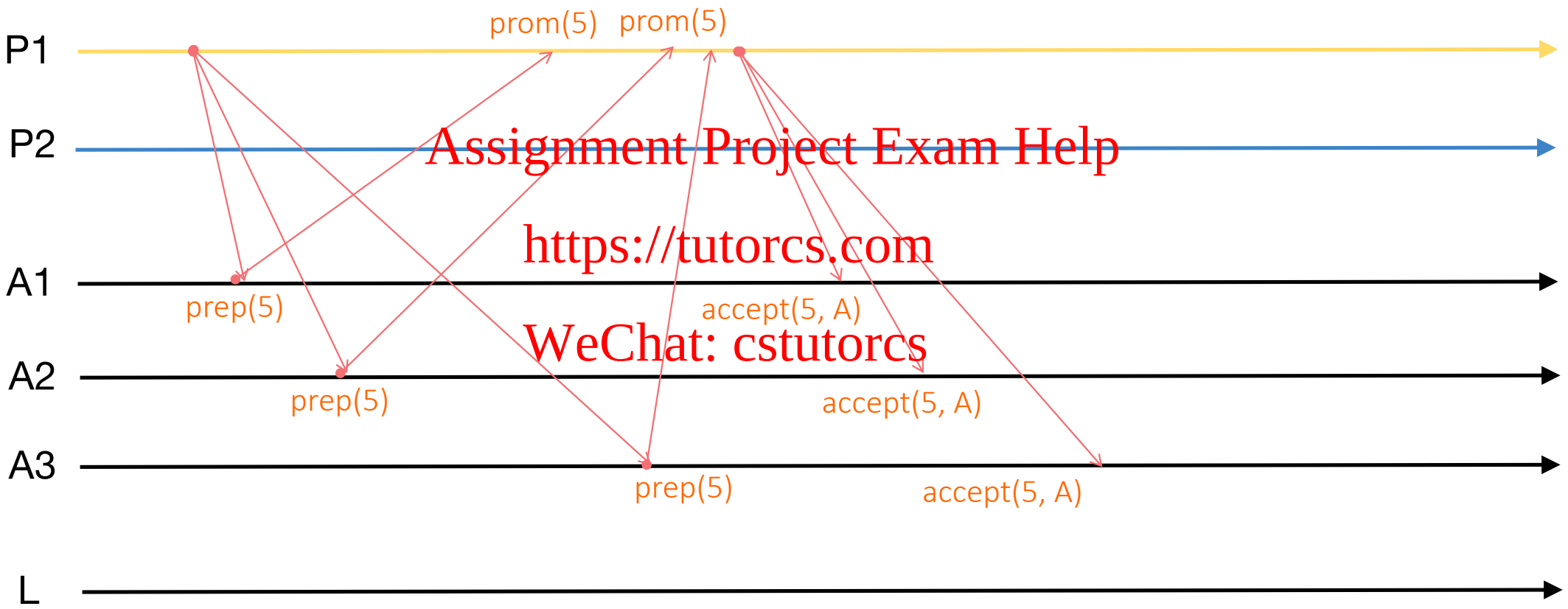
Recap example: P1 want to propose value A



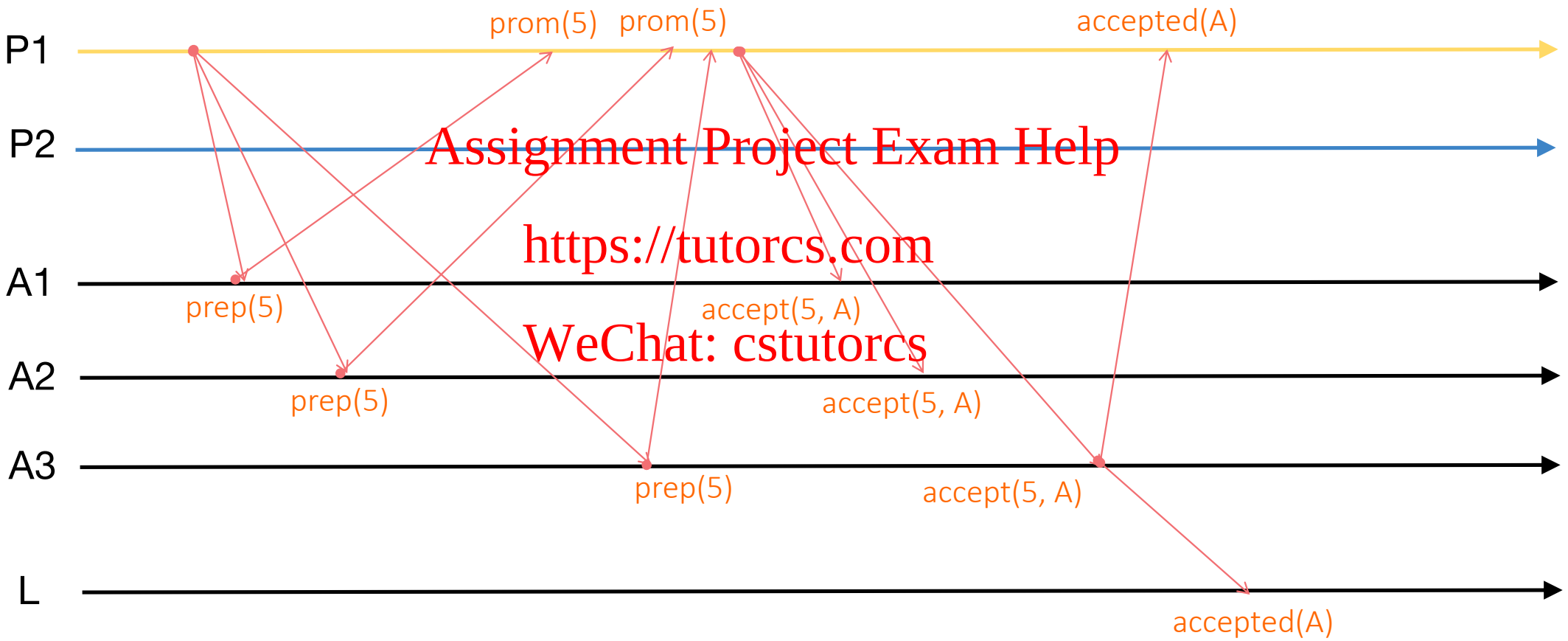
Recap example: P1 want to propose value A



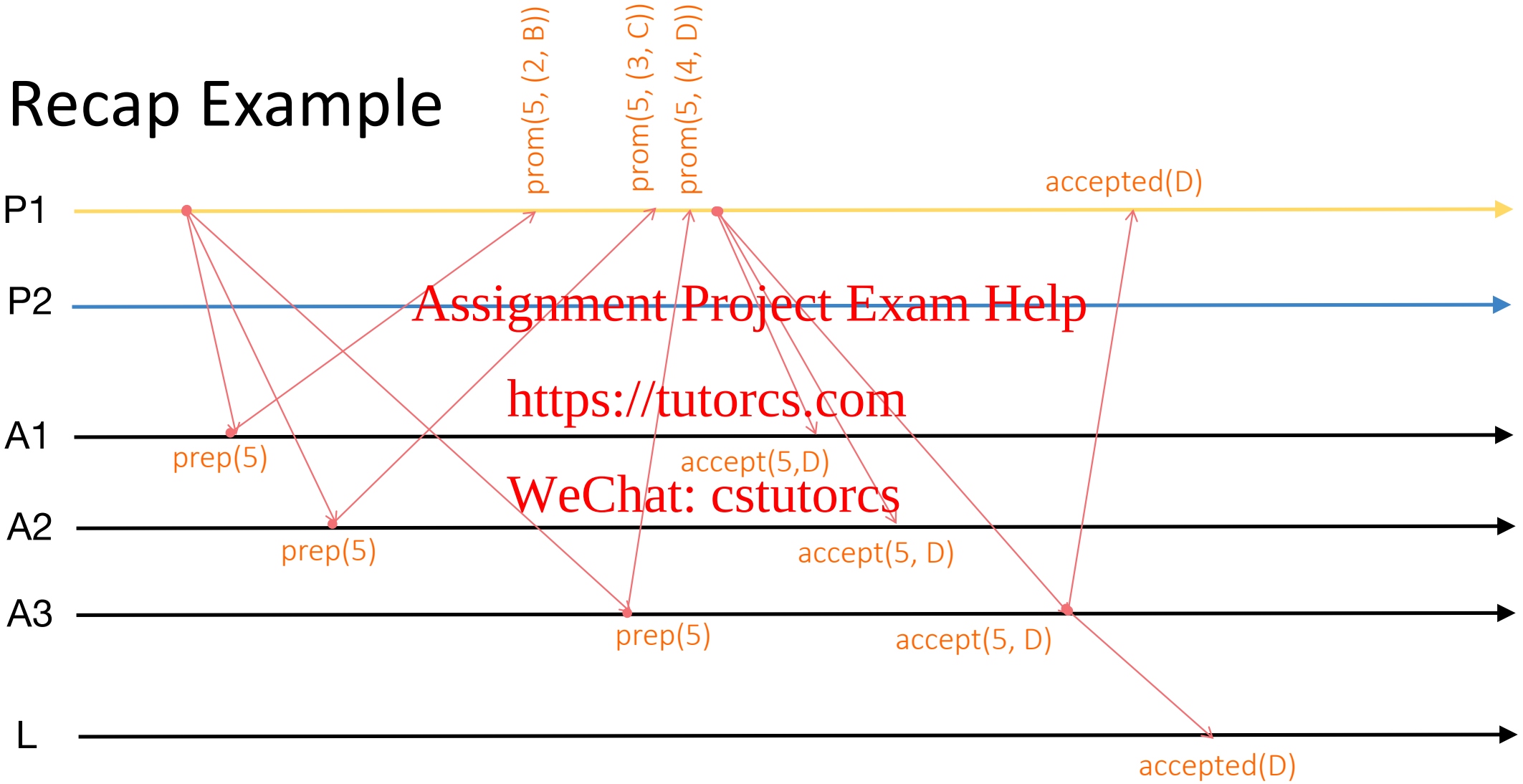
Recap example: P1 want to propose value A



Recap example: P1 want to propose value A

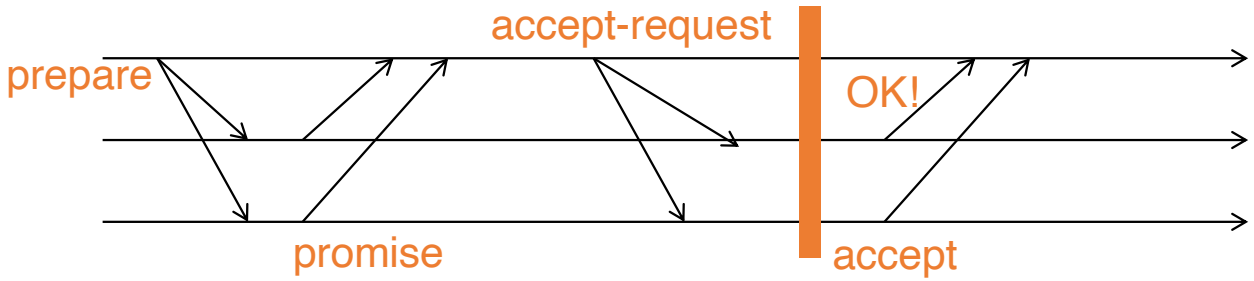


Recap Example



What can go wrong? (Liveness)

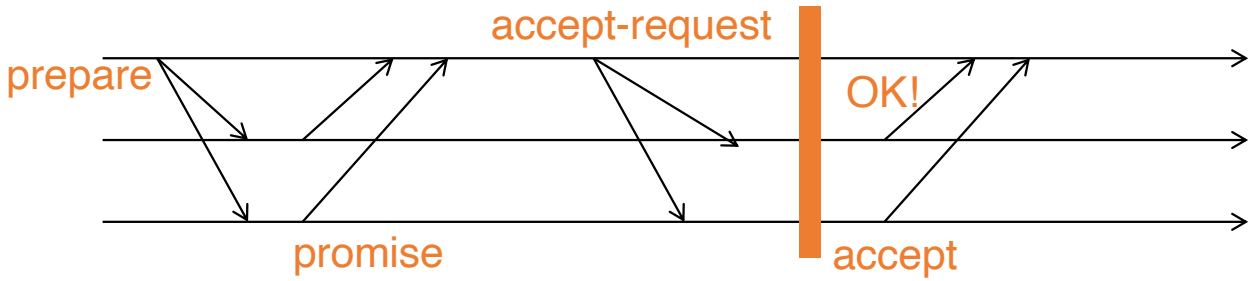
- Process fails
  - still works as long as majority are up
- Leader fails
  - Start another round



What can go wrong? (Liveness)

- Message dropped
  - If too flaky, start another round

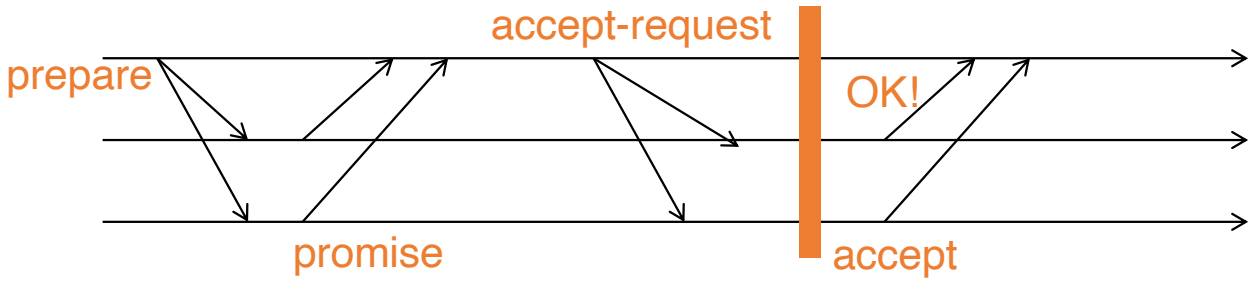
- Note that anyone can start a round any time



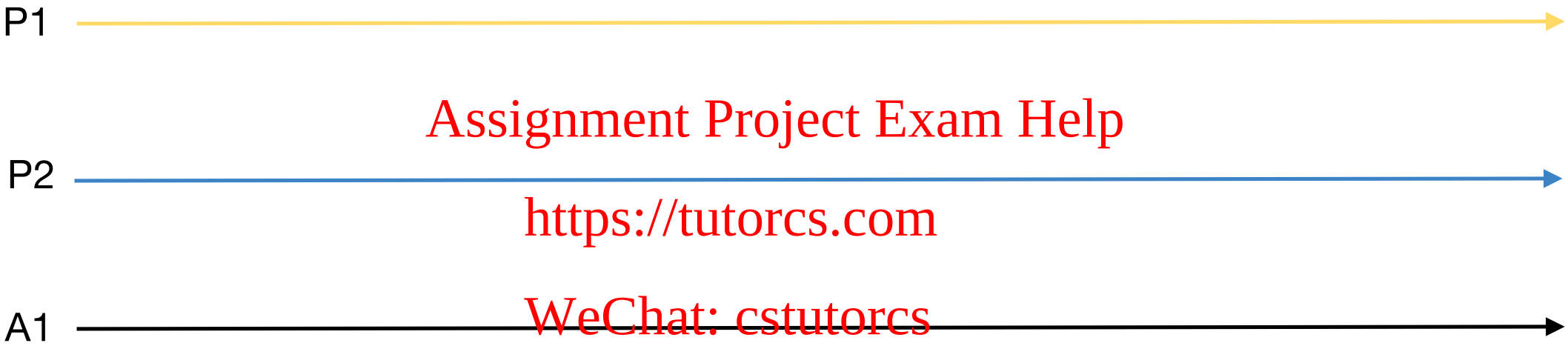
What can go wrong? (Liveness)

- Protocol may never end – tough luck, buddy!
  - Impossibility result not violated
  - If things go well sometime in the future, consensus reached!

- Example: two or more simultaneous proposer

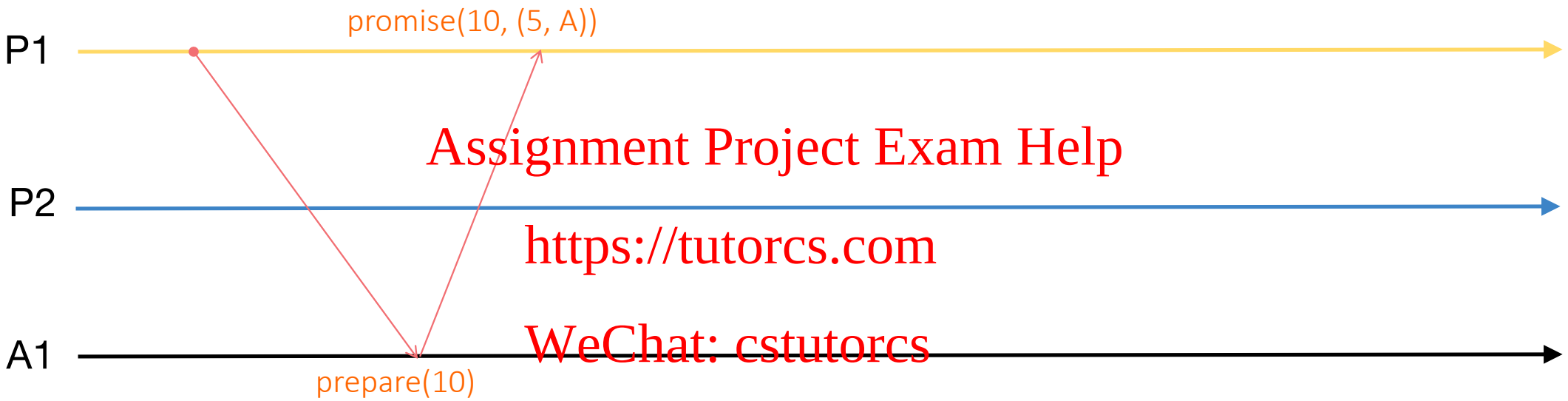


Livelock



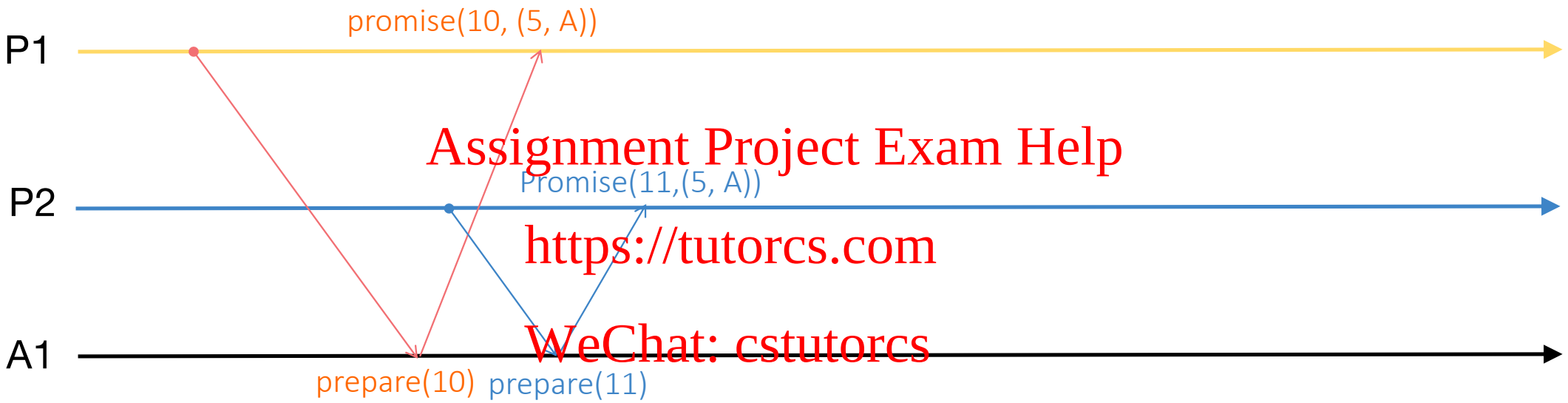
A1: Highest accept; (5, A)
Highest prepare: 8

Livelock



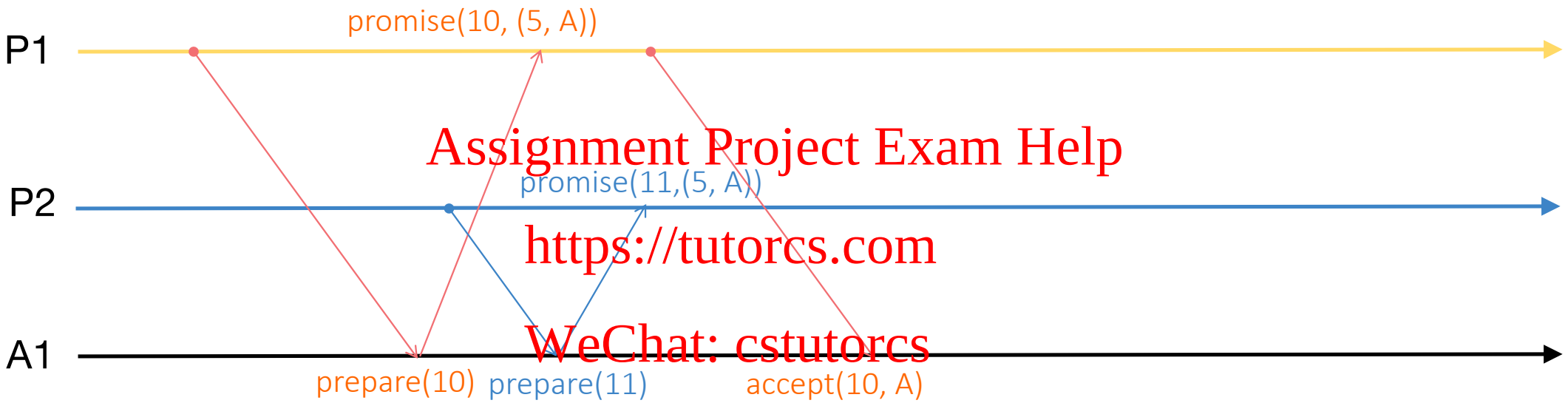
A1: Highest accept; (5, A)
Highest prepare: 10

Livelock



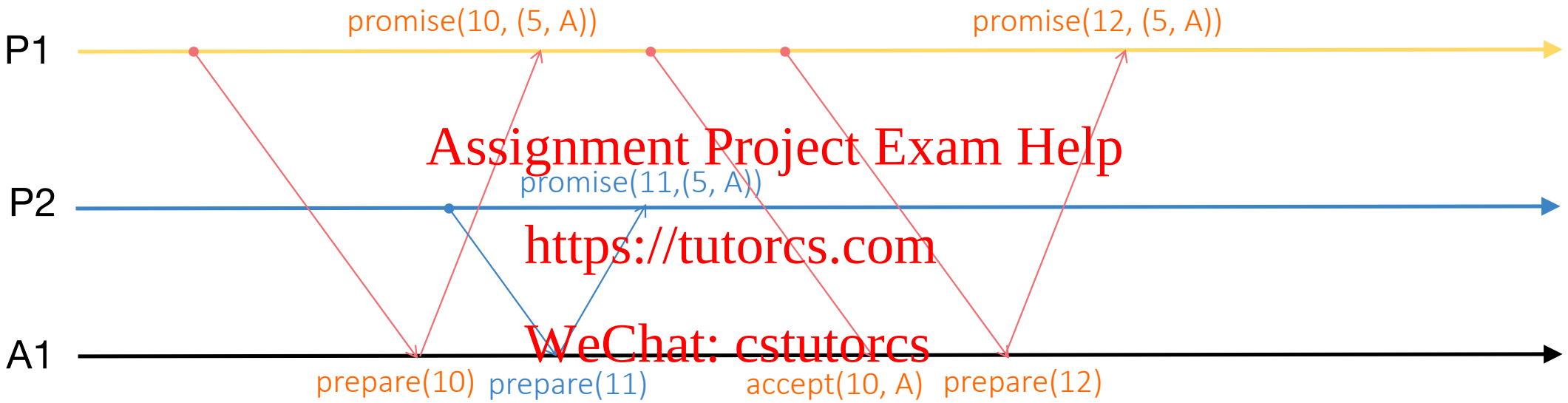
A1: Highest accept; (5, A)
Highest prepare: 11

Livelock



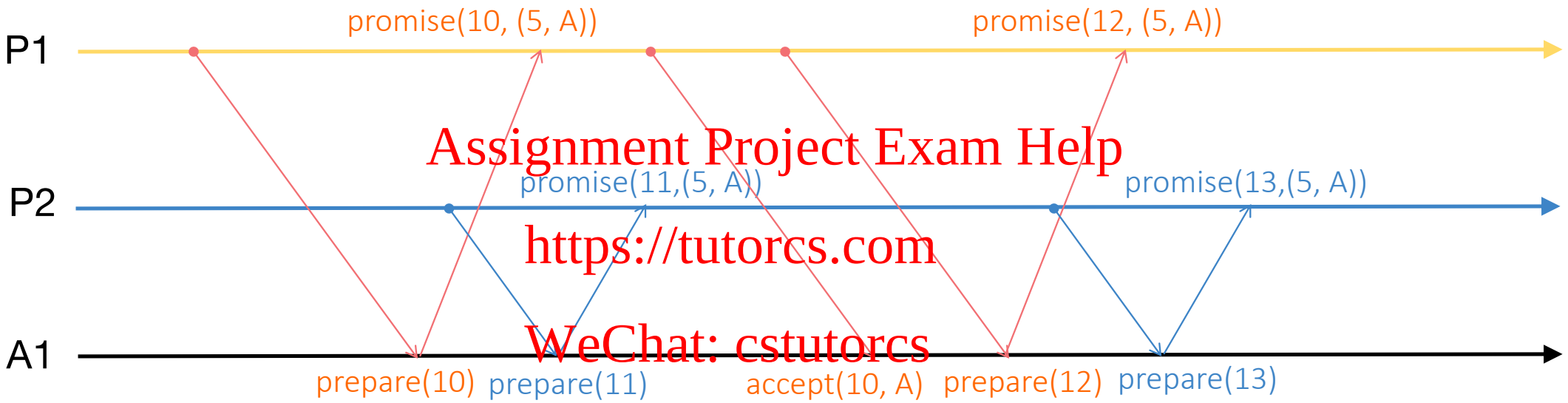
A1: Highest accept; (5, A)
Highest prepare: 11

Livelock



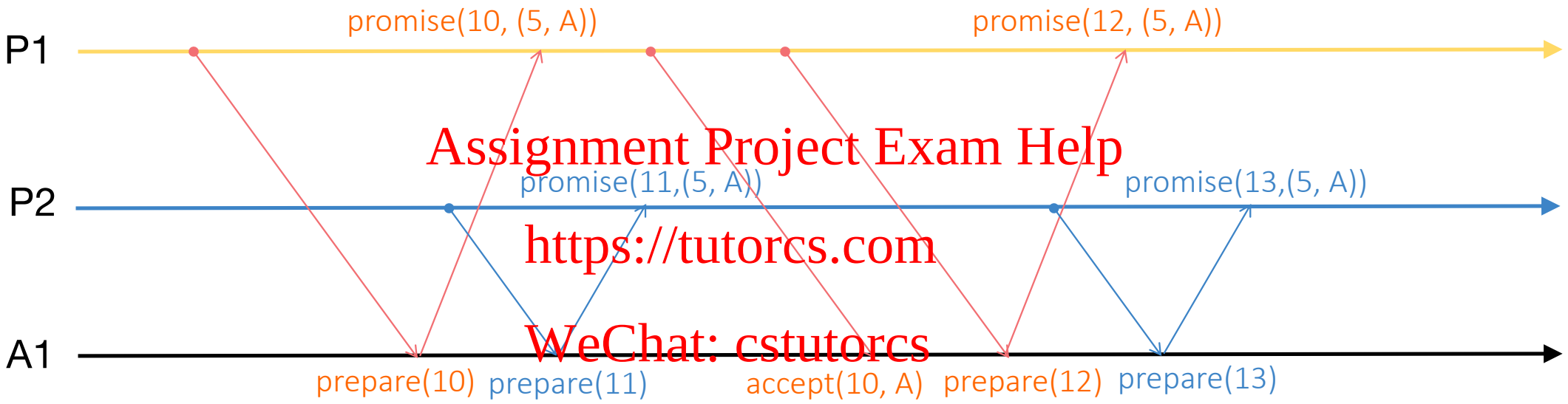
A1: Highest accept; (5, A)
Highest prepare: 12

Livelock



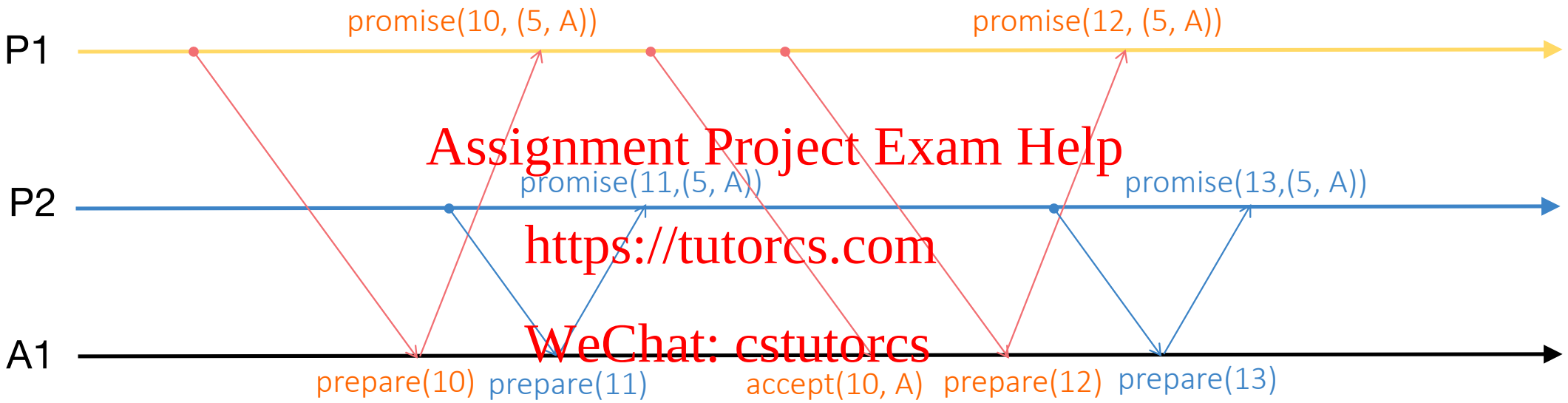
A1: Highest accept; (5, A)
Highest prepare: 13

Livelock



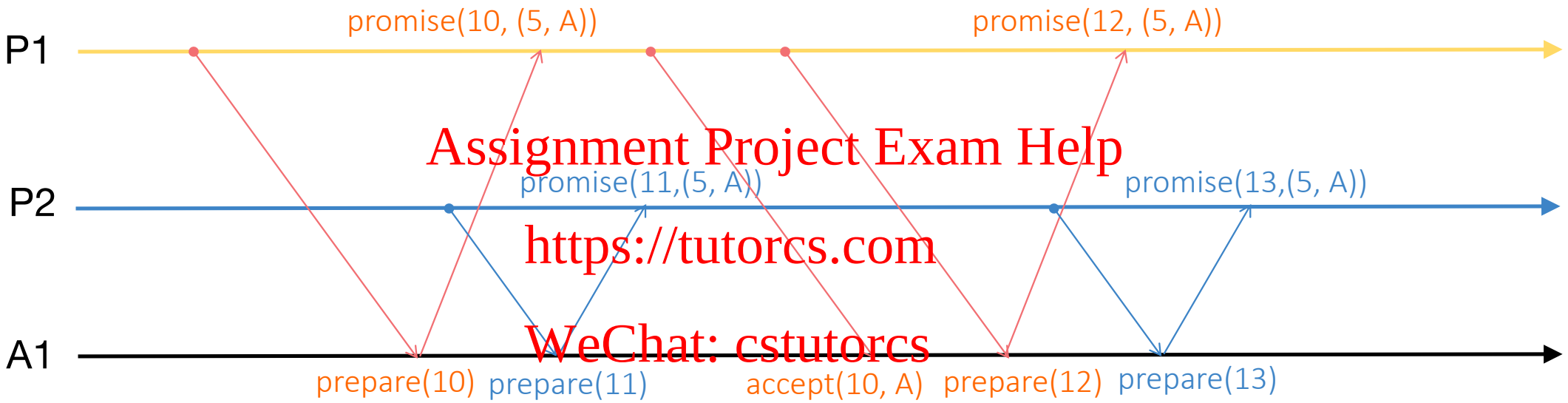
So now?

Livelock



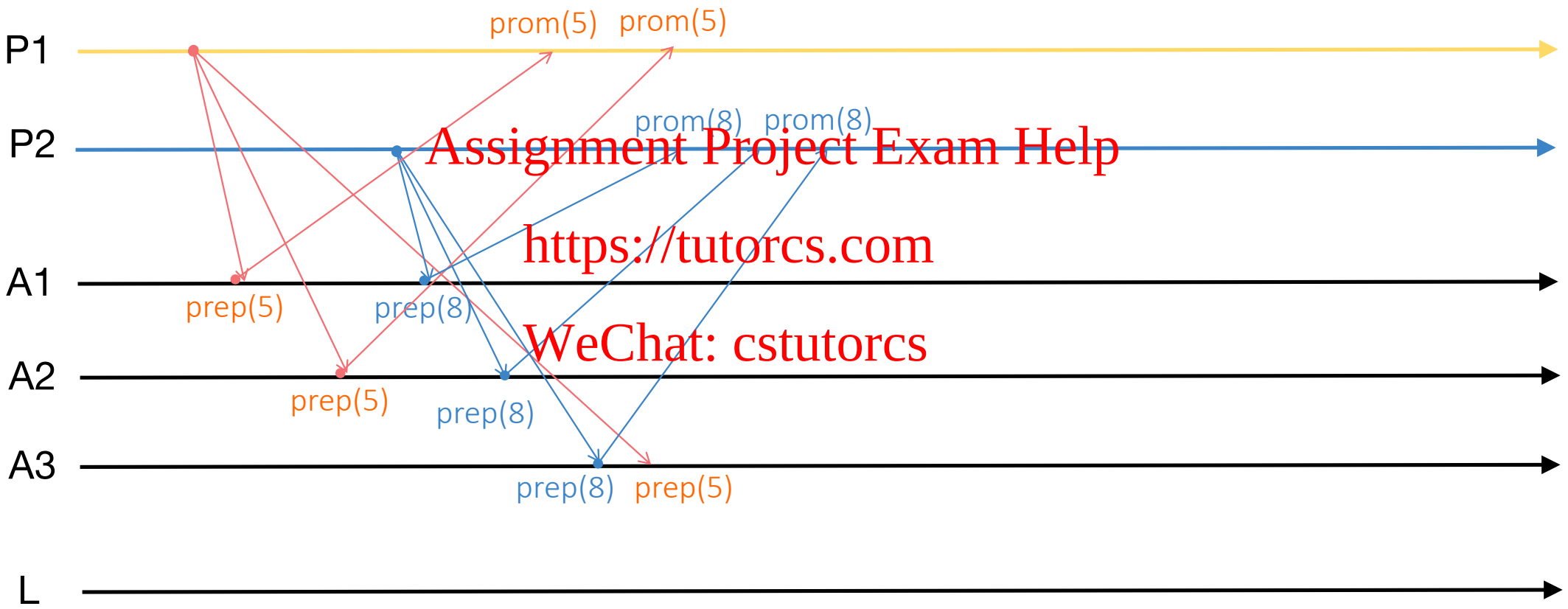
This is a hot spot that can stall the Paxos run. A solution is to set an exponential back off in place.

Livelock

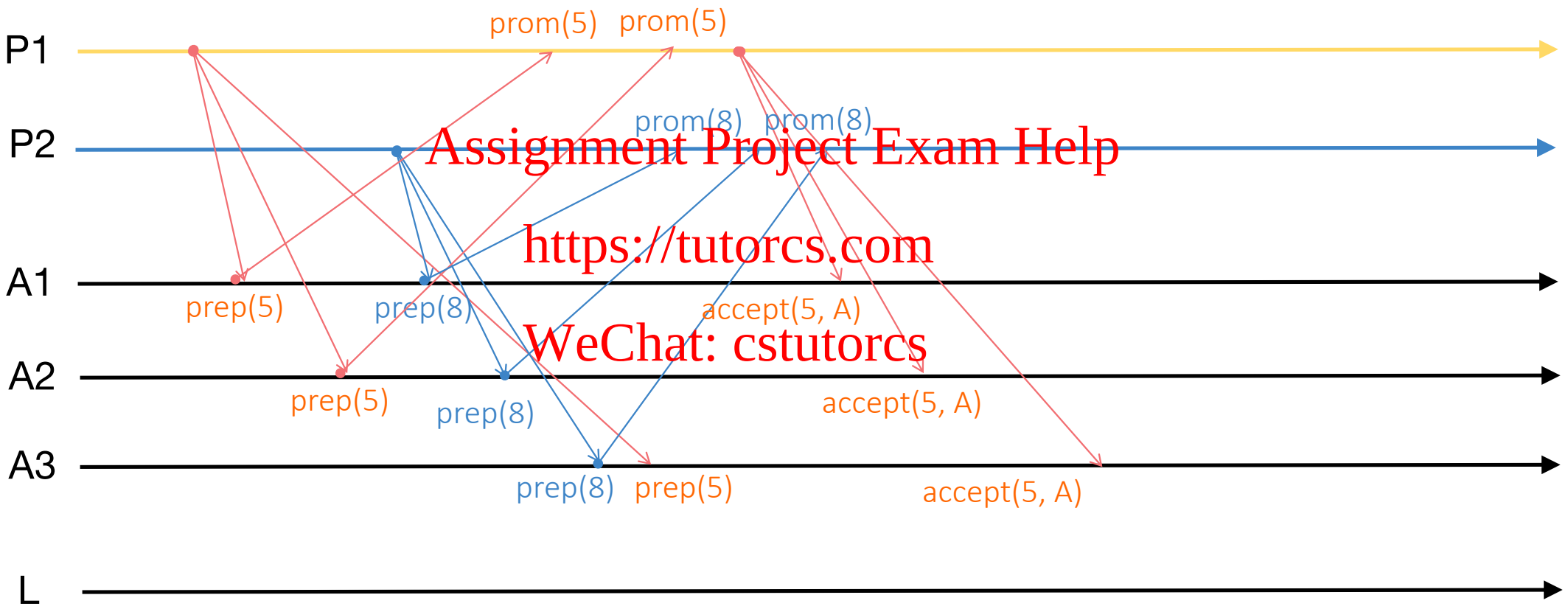


Indeed, if there is enough time….

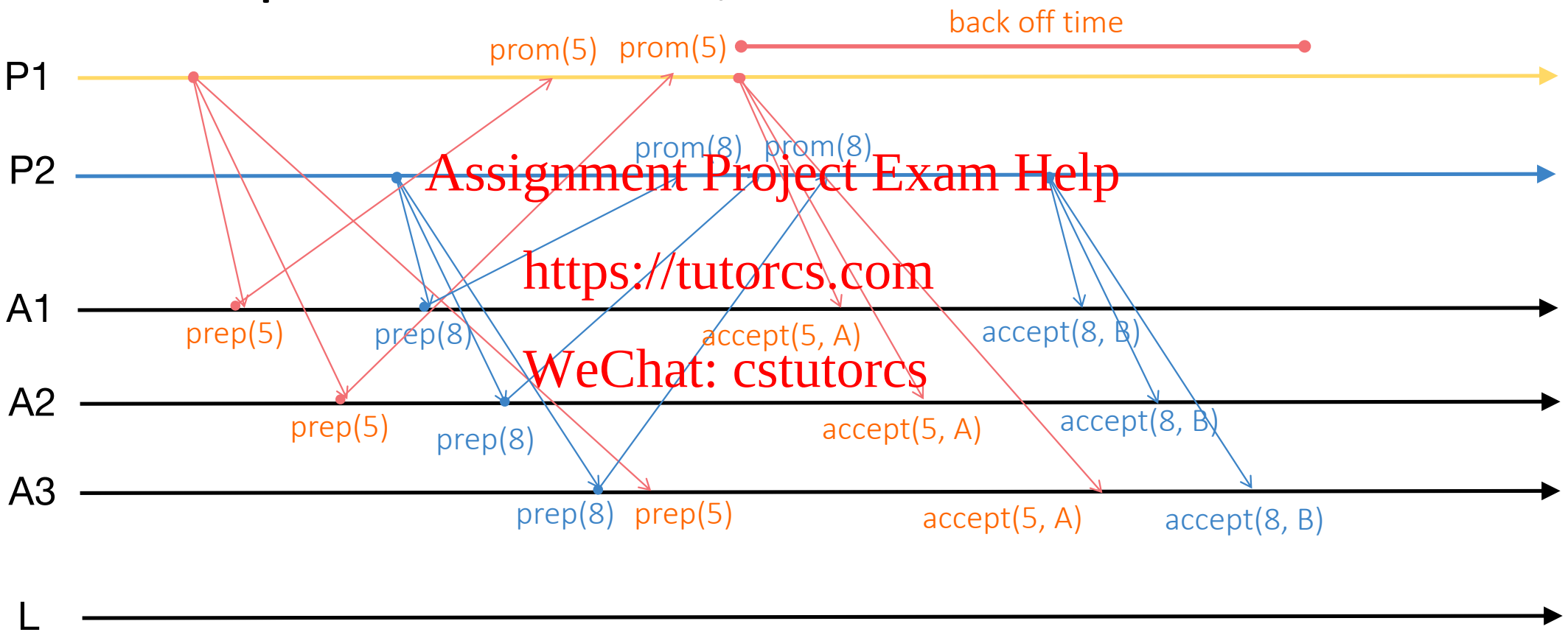
P1 wants A, and P2 wants B



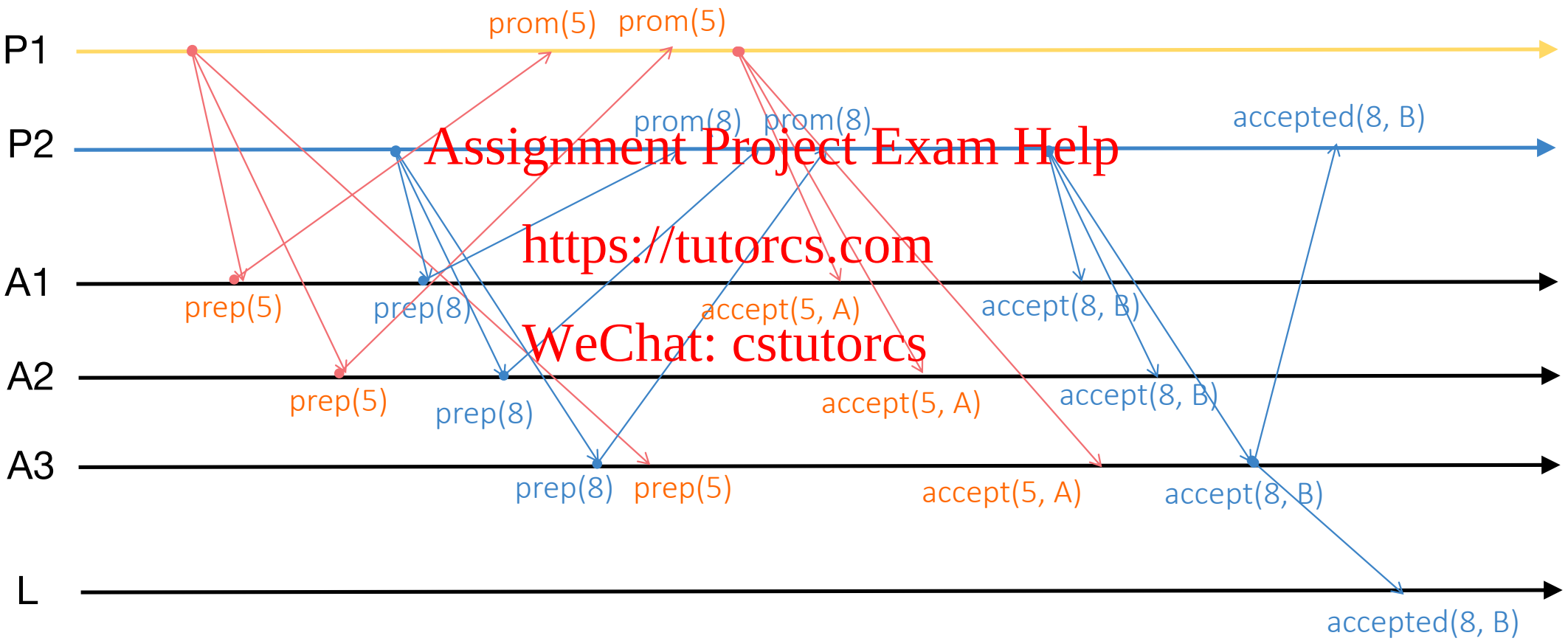
Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



Example: P1 wants A, and P2 wants B



Others

- Acceptors might send “NACKs” responses if they are not going to accept a proposal. This would tell the proposer that it can stop its attempt to create consensus with proposal N
- We said the proposal number needs to be strictly increasing and globally unique. How to do it?

Others

- Acceptors might send “NACKs” responses if they are not going to accept a proposal. This would tell the proposer that it can stop its attempt to create consensus with proposal N
- We said the proposal number needs to be strictly increasing and globally unique. How to do it?

Tick: set low-order bits to proposer’s (server) ID