Distributed Systems

Uwe R. Zimmer - The Australian National University

Distribution!

Motivation

Possibly …

- … fits an **existing physical distribution** (e-mail system, devices in a large craft, …).

- … **high performance** due to potentially high degree of parallel processing.
- … **high reliability/integrity** due to redundancy of hardware and software.
- … **scalable**.
- … **integration** of heterogeneous devices.

Different specifications will lead to substantially different distributed designs.

Distributed Systems

What can be distributed?

- **State** ☞ *Common* operations on *distributed* data

- **Function** ☞ *Distributed* operations on *central* data

- **State & Function** ☞ Client/server clusters
- **none of those** ☞ Pure replication, redundancy

Distributed Systems

Common design criteria

- Achieve **De-coupling** / high degree of local autonomy

- **Cooperation** rather than central control

- Consider **Reliability**
- Consider **Scalability**
- Consider **Performance**

Some common phenomena in distributed systems

1. **Unpredictable delays** (communication)
   - ☞ Are we done yet?

2. **Missing or imprecise time-base**
   - ☞ Causal relation or temporal relation?

3. **Partial failures**
   - ☞ Likelihood of individual failures increases
   - ☞ Likelihood of complete failure decreases (in case of a good design)

Distributed Systems

Time in distributed systems

Two alternative strategies:

Based on a shared time ☞ **Synchronize clocks!**

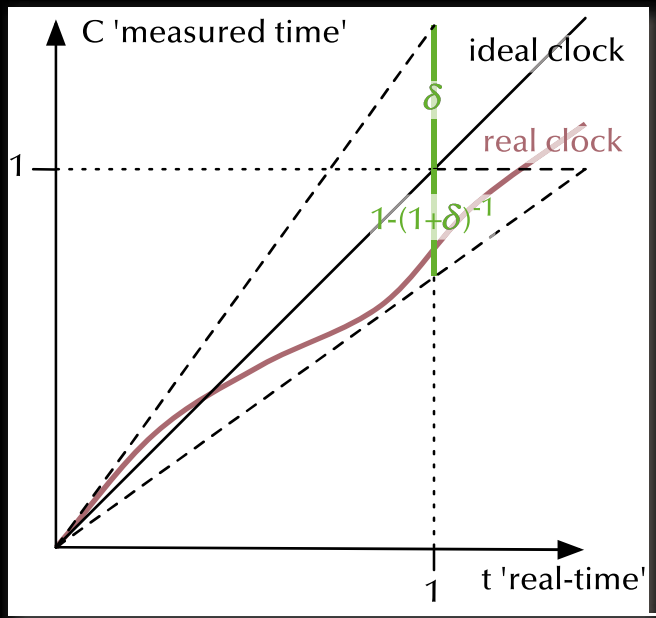
Based on sequence of events ☞ **Create a virtual time!**

Distributed Systems

'Real-time' clocks

are:

- **discrete** – i.e. time is *not* dense and there is a minimal granularity
- **drift affected**:



Maximal clock drift δ defined as:

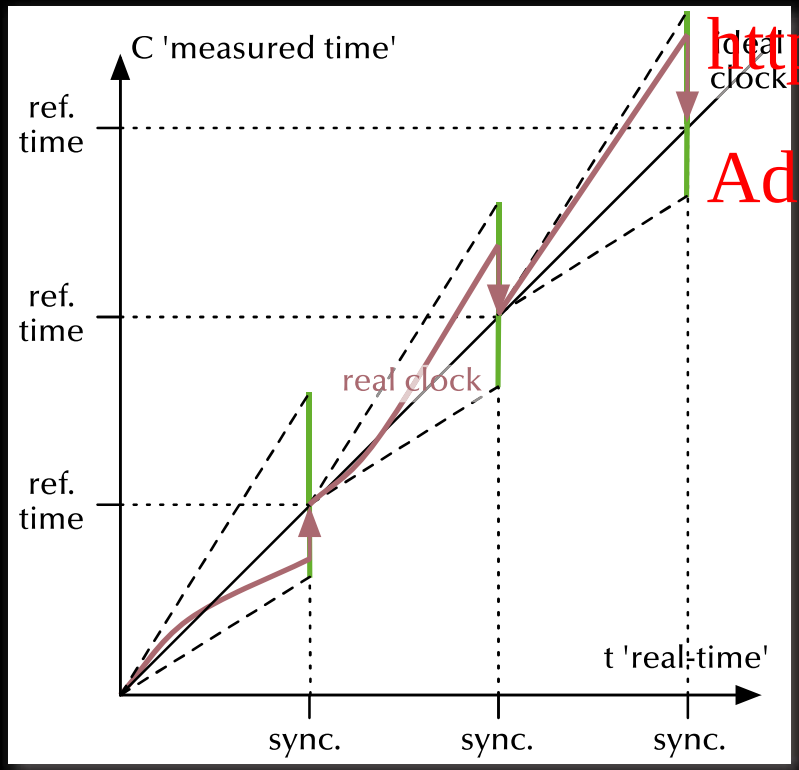
$$(1+\delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1+\delta)$$

often specified as PPM (Parts-Per-Million)

(typical $\approx$ 20 PPM in computer applications)

Synchronize a 'real-time' clock *(bi-directional)*

Resetting the clock drift by regular reference time re-synchronization:



Maximal clock drift δ defined as:

$$(1+\delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1+\delta)$$

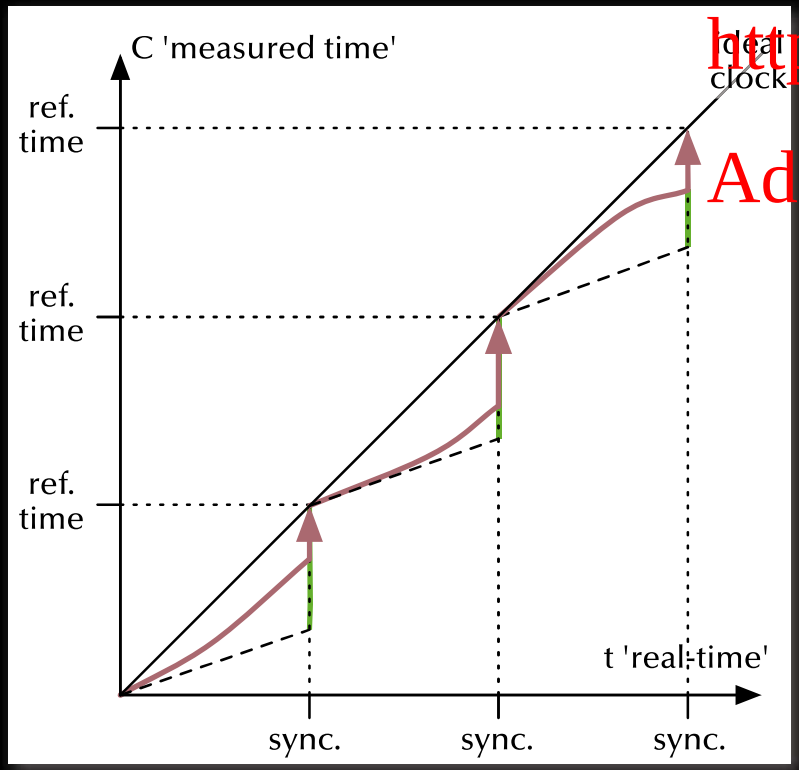
'real-time' clock is adjusted *forwards & backwards*

☞ **Calendar time**

Distributed Systems

Synchronize a 'real-time' clock *(forward only)*

Resetting the clock drift by regular reference time re-synchronization:



Maximal clock drift δ defined as:

$$(1+\delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq 1$$

'real-time' clock is adjusted *forwards* only

☞ **Monotonic time**

Distributed critical regions with synchronized clocks

- ∀ times:
  ∀ received *Request*s: **Add** to local *RequestQueue* (ordered by time)
  ∀ received *Release messages*:
  Delete *corresponding Requests* in local *RequestQueue*

1. **Create** *OwnRequest* and **attach** current time-stamp.
   Add *OwnRequest* to local *RequestQueue* (ordered by time).
   Send *OwnRequest* to *all* processes.
2. **Delay** by 2*L* (*L* being the time it takes for a message to reach all network nodes)
3. **While** Top (*RequestQueue*) ≠ *OwnRequest*: **delay** until new message
4. **Enter** and **leave** critical region
5. **Send** *Release*-message to *all* processes.

Distributed Systems

Distributed critical regions with synchronized clocks

Analysis

- No deadlock, no individual starvation, no livelock.

- Minimal request delay: $2L$.

- Minimal release delay: L.
- Communications requirements per request: $2(N-1)$ messages
  (can be significantly improved by employing broadcast mechanisms).
- Clock drifts affect fairness, but not integrity of the critical region.

Assumptions:

- L is known and constant ☞ violation leads to loss of mutual exclusion.
- No messages are lost ☞ violation leads to loss of mutual exclusion.

Distributed Systems

Virtual (logical) time [Lamport 1978]

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

with $a \rightarrow b$ being a causal relation between a and b,
and $C(a)$, $C(b)$ are the (virtual) times associated with a and b.

$a \rightarrow b$ iff:

- a happens **earlier than** b in the *same sequential* control-flow or
- a denotes the **sending event** of message m,
  while b denotes the **receiving event** of the *same message* m or
- there is a **transitive causal relation** between a and b: $a \rightarrow e_1 \rightarrow \ldots \rightarrow e_n \rightarrow b$

Notion of concurrency:

$$a \parallel b \Rightarrow \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow ?$$

$$C(a) = C(b) \Rightarrow ?$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg (b \rightarrow a)$$

$$C(a) = C(b) \Rightarrow a \parallel b$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

$$C(a) = C(b) < C(c) \Rightarrow \neg(c \rightarrow a)$$

$$C(a) < C(b) < C(c) \Rightarrow \neg(c \rightarrow a)$$

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

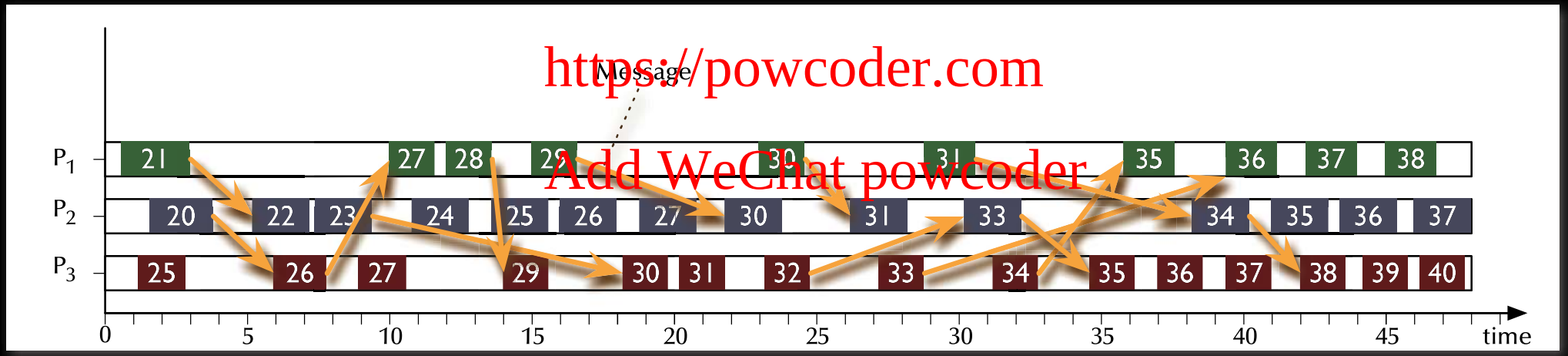
$$C(a) = C(b) < C(c) \Rightarrow \neg(c \rightarrow a) = (a \rightarrow c) \vee (a \parallel c)$$

$$C(a) < C(b) < C(c) \Rightarrow \neg(c \rightarrow a) = (a \rightarrow c) \vee (a \parallel c)$$

Distributed Systems

Virtual (logical) time

Time as derived from causal relations:



☞ Events in concurrent control flows are not ordered.

☞ No global order of time.

Implementing a virtual (logical) time

1. $\forall P_i$: $C_i = 0$

2. $\forall P_i$:

   $\forall$ local events: $C_i = C_i + 1$;

   $\forall$ send events: $C_i = C_i + 1$; Send (message, C_i);

   $\forall$ receive events: Receive (message, C_m); $C_i = \max(C_i, C_m) + 1$;

Distributed critical regions with logical clocks

- ∀ times: ∀ received *Requests*:

  Add to local *RequestQueue* (ordered by time)

  Reply with *Acknowledge* or *OwnRequest*

- ∀ times: ∀ received *Release messages*:

  Delete corresponding *Requests* in local *RequestQueue*

1. **Create** *OwnRequest* and **attach** current time-stamp.
   Add *OwnRequest* to local *RequestQueue* (ordered by time).
   Send *OwnRequest* to *all* processes.
2. **Wait for** Top (*RequestQueue*) = *OwnRequest* & no outstanding replies
3. **Enter** and **leave** critical region
4. **Send** *Release*-message to *all* processes.

Distributed Systems

Distributed critical regions with logical clocks

Analysis

- No deadlock, no individual starvation, no livelock.

- Minimal request delay: $N-1$ *requests* (1 broadcast) + $N-1$ *replies*.

- Minimal release delay: $N-1$ *release* messages (or 1 broadcast).
- Communications requirements per request: $3(N-1)$ messages (or $N-1$ messages + 2 broadcasts).
- Clocks are kept recent by the exchanged messages themselves.

Assumptions:

- No messages are lost ☞ violation leads to stall.

Distributed critical regions with a token ring structure

1. **Organize** all processes in a logical or physical **ring** topology

2. **Send** one *token* message to one process

3. ∀ times, ∀processes: **On receiving** the *token* message:
   1. If required the process
      enters and **leaves** a critical section (while holding the token).
   2. The *token* is **passed** along to the next process in the ring.

Assumptions:

- Token is not lost ☞ violation leads to stall.

(a lost token can be recovered by a number of means – e.g. the 'election' scheme following)

Distributed critical regions with a central coordinator

A global, static, central coordinator

- ☞ Invalidates the idea of a distributed system

- ☞ Enables a very simple mutual exclusion scheme

Therefore:

- A global, central coordinator is employed in some systems … yet …
- … if it fails, a system to come up with a new coordinator is provided.

Electing a central coordinator (the Bully algorithm)

Any process *P* which notices that the central coordinator is gone, performs:

1. *P* **sends** an *Election*-message
   to all processes with *higher* process numbers.

2. *P* **waits** for response messages.

   - ☞ If no one responds after a pre-defined amount of time:
     P declares itself the new coordinator and sends out a *Coordinator*-message to all.
   - ☞ If any process responds,
     then the election activity for *P* is over and *P* waits for a *Coordinator*-message

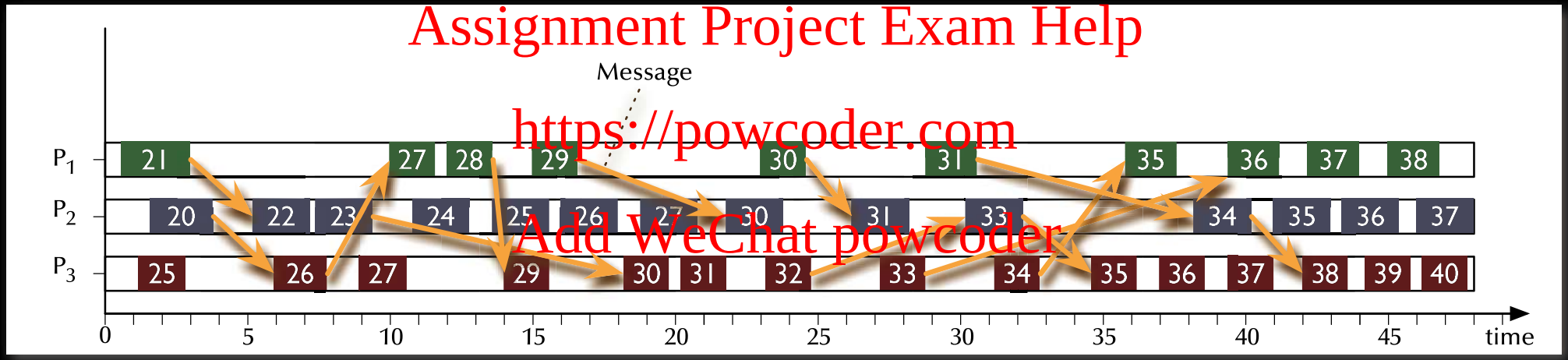
All processes P_i perform at all times:

- If P_i **receives** a *Election*-message from a process with
  a *lower* process number, it **responds** to the originating process
  and starts an election process itself (if not running already).

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?

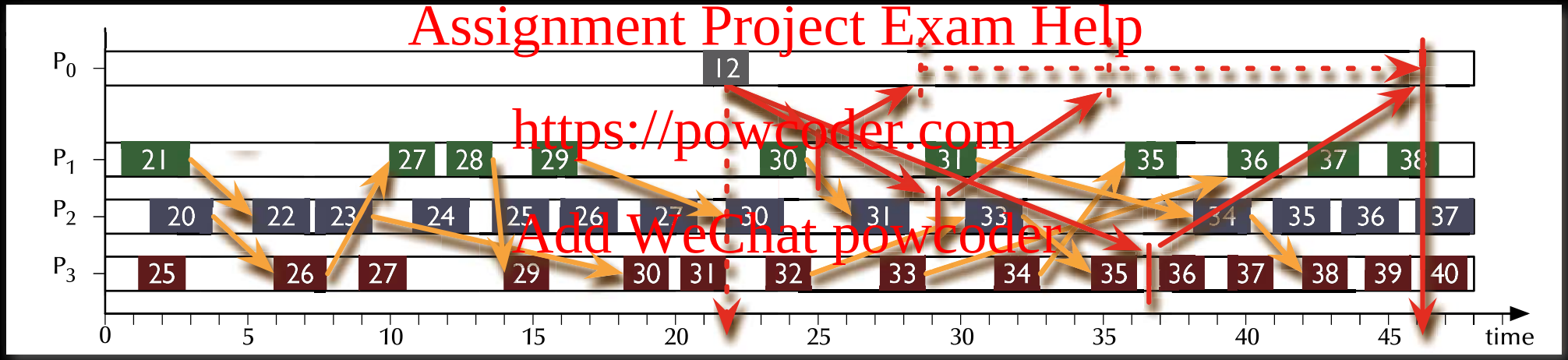


This "god's eye view" does in fact not exist.

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?

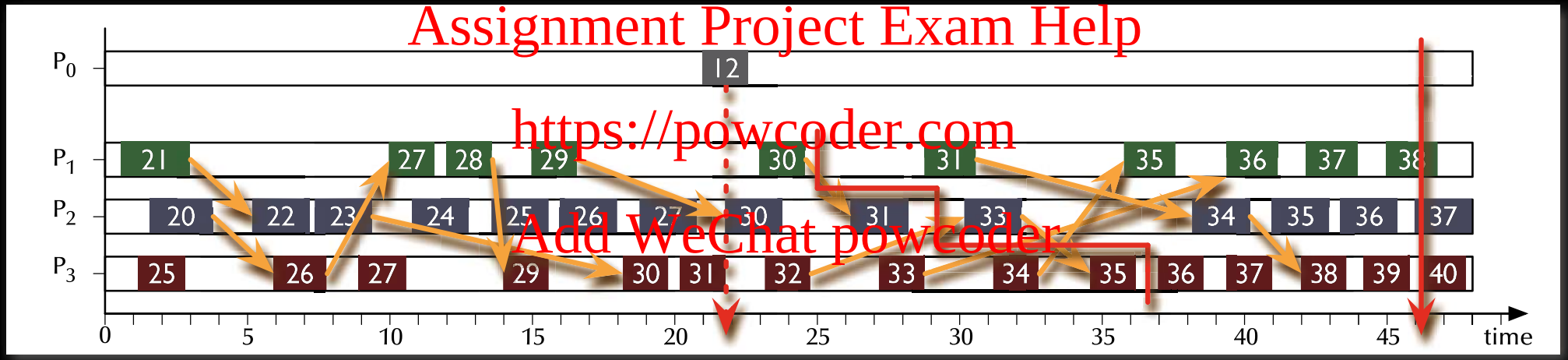


Instead: some entity probes and collects local states.

☞ What state of the global system has been accumulated?

Distributed states

☞ How to read the current state of a distributed system?



Instead: some entity probes and collects local states.

☞ What state of the global system has been accumulated?

☞ Connecting all the states to a global state.

Distributed states

A consistent global state (snapshot) is define by a unique division into:

- "The Past" P (events before the snapshot):

$$(e_2 \in P) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in P$$

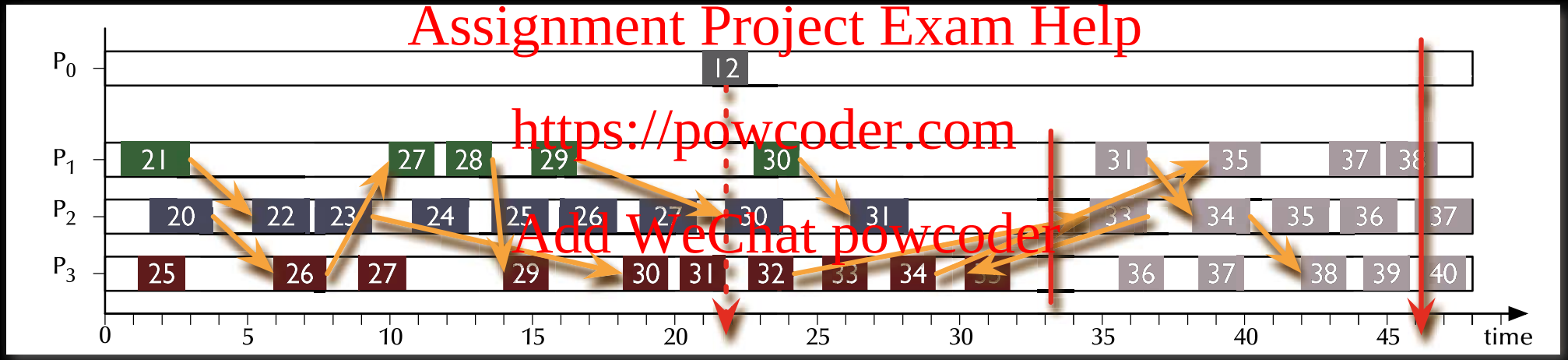
- "The Future" F (events after the snapshot):

$$(e_1 \in F) \wedge (e_1 \rightarrow e_2) \Rightarrow e_2 \in F$$

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?



Instead: some entity probes and collects local states.

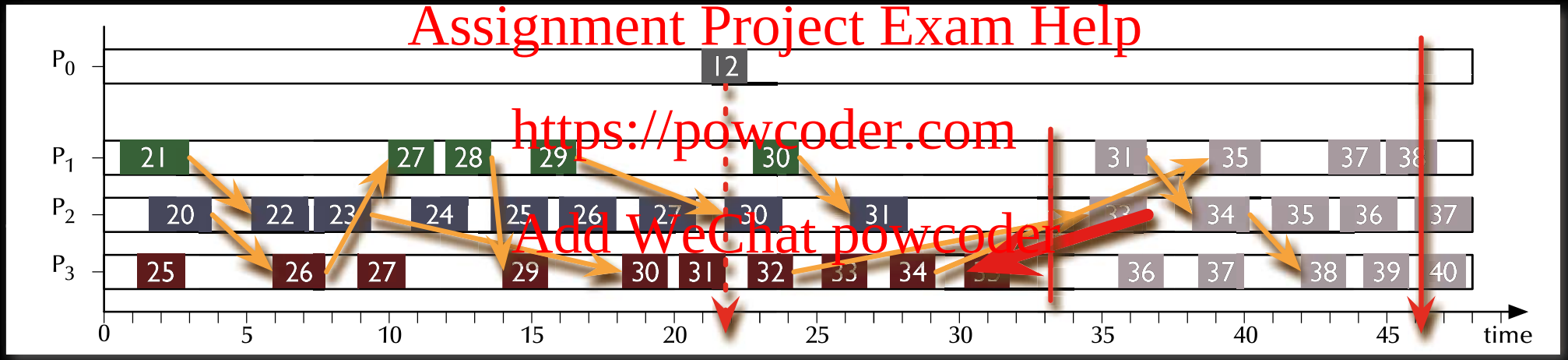
☞ What state of the global system has been accumulated?

☞ Sorting the events into past and future events.

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?



Instead: some entity probes and collects local states.

☞ What state of the global system has been accumulated?

☞ Event in the past receives a message from the future!

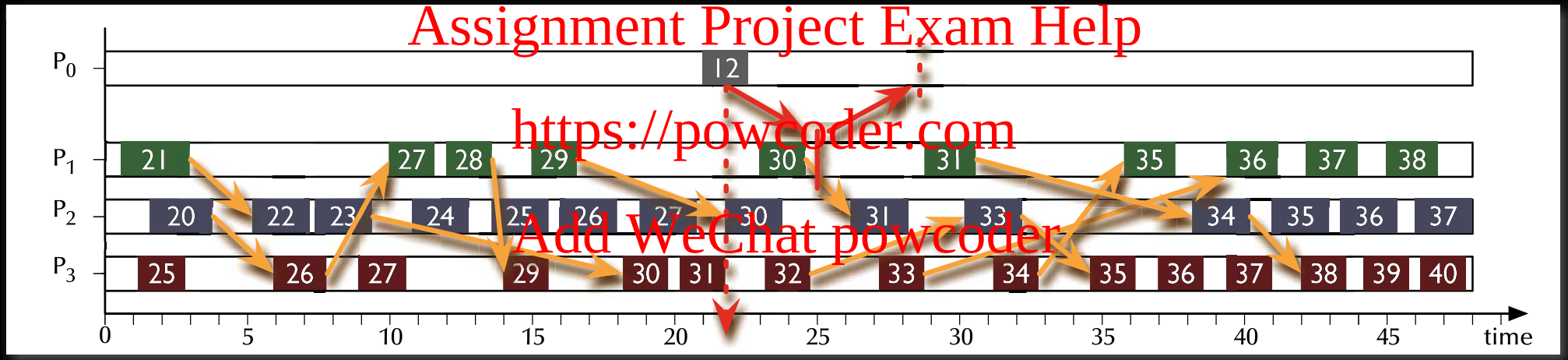
Division not possible ☞ Snapshot inconsistent!

Snapshot algorithm

- Observer-process P_0 (any process) **creates** a snapshot token t_s and **saves** its local state s_0.
- P_0 **sends** t_s to all other processes.
- $\forall P_i$ which **receive** t_s (as an individual token-message, or as part of another message):
  - **Save** local state s_i and **send** s_i to P_0.
  - **Attach** t_s to all further messages, which are to be sent to other processes.
  - **Save** t_s and **ignore** all further incoming t_s's.
- $\forall P_i$ which previously received t_s and **receive** a message m without t_s:
  - **Forward** m to P_0 (this message belongs to the snapshot).

Distributed states

☞ Running the snapshot algorithm:

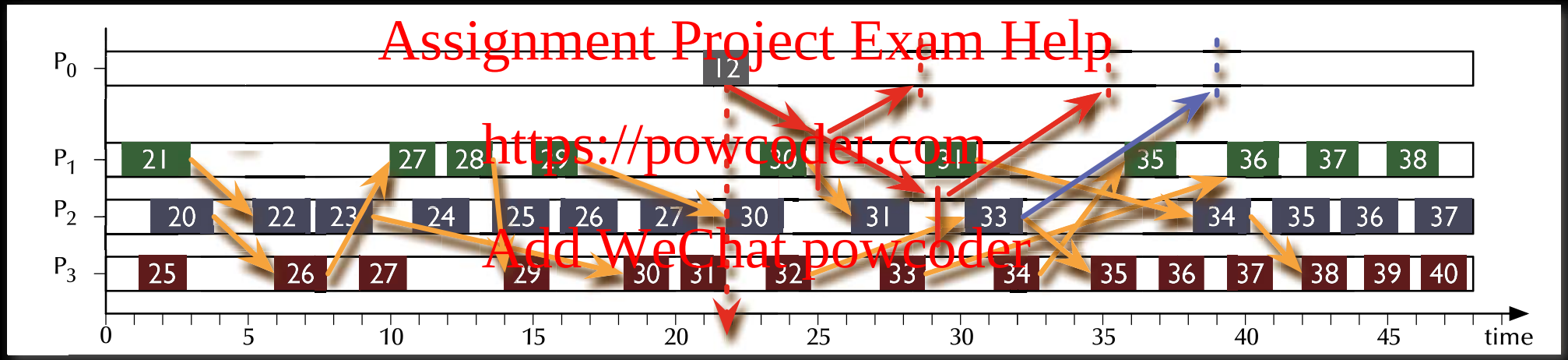


- Observer-process P_0 (any process) **creates** a snapshot token t_s and **saves** its local state s_0.
- P_0 **sends** t_s to all other processes.

Distributed Systems

Distributed states

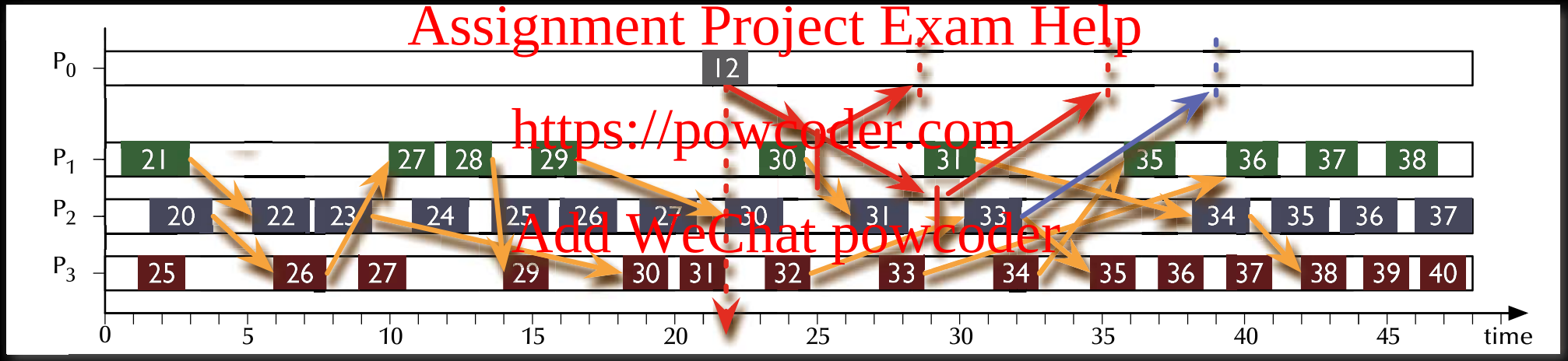
☞ Running the snapshot algorithm:



- $\forall P_i$ which **receive** t_s (as an individual token-message, or as part of another message):
  - **Save** local state s_i and **send** s_i to P_0.
  - **Attach** t_s to all further messages, which are to be sent to other processes.
  - **Save** t_s and **ignore** all further incoming t_s's.

Distributed states

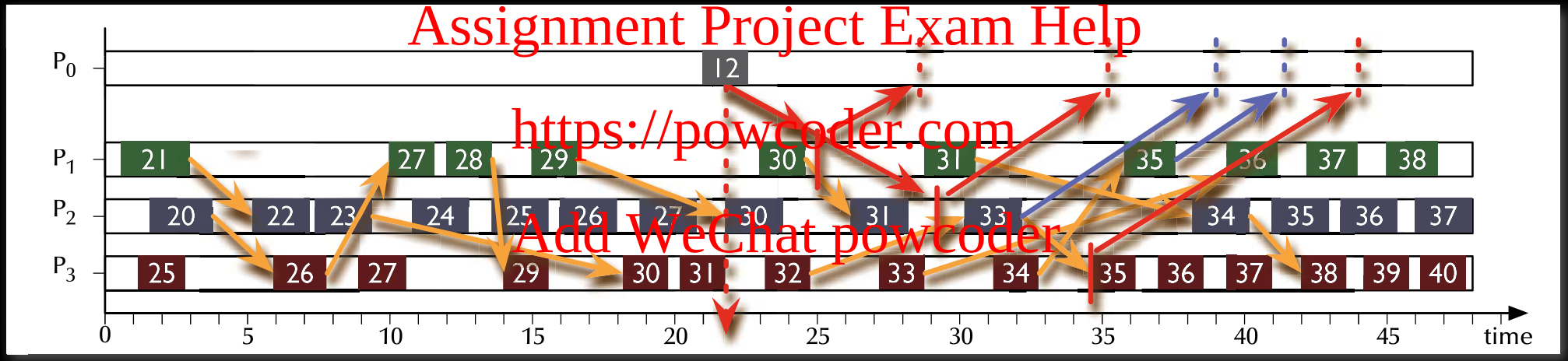
☞ Running the snapshot algorithm:



- $\forall P_i$ which previously received t_s and **receive** a message m without t_s:
  - **Forward** m to P_0 (this message belongs to the snapshot).

Distributed states

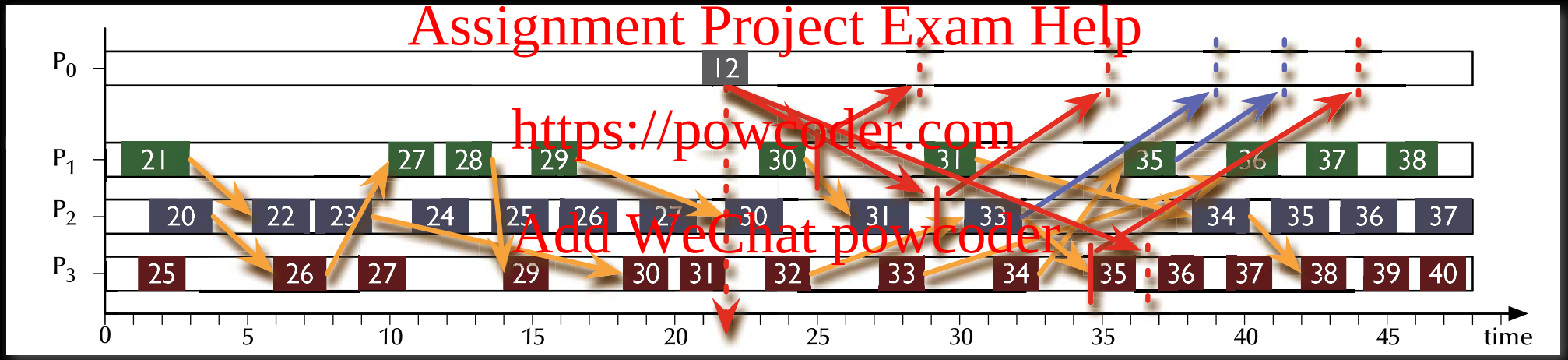
☞ Running the snapshot algorithm:



- $\forall P_i$ which **receive** t_s (as an individual token-message, or as part of another message):
  - **Save** local state s_i and **send** s_i to P_0.
  - **Attach** t_s to all further messages, which are to be sent to other processes.
  - **Save** t_s and **ignore** all further incoming t_s's.

Distributed states

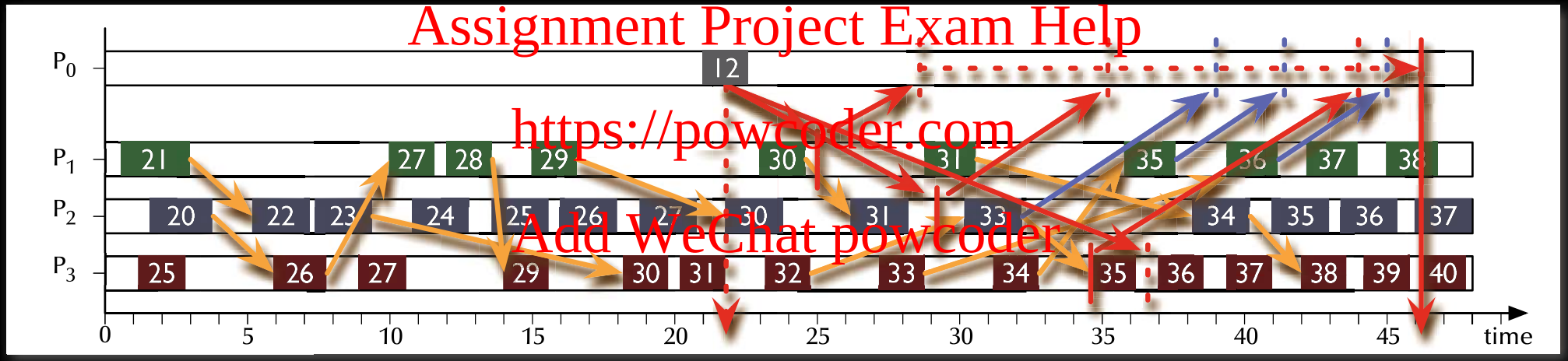
☞ Running the snapshot algorithm:



- **Save** t_s and **ignore** all further incoming t_s's.

Distributed states

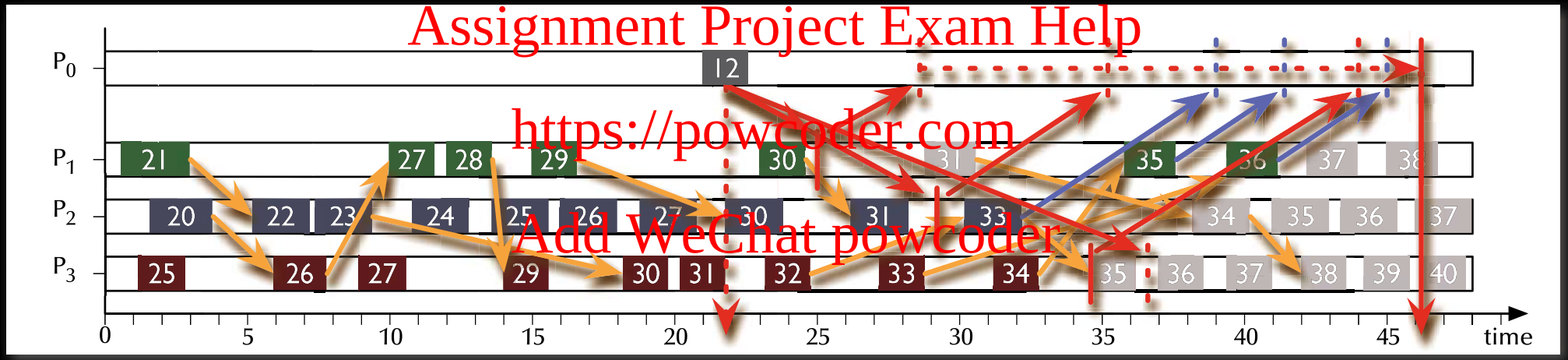
☞ Running the snapshot algorithm:



- **Finalize** snapshot

Distributed states

☞ Running the snapshot algorithm:



☞ Sorting the events into past and future events.

☞ Past and future events uniquely separated ☞ Consistent state

Distributed Systems

Snapshot algorithm

Termination condition?

Either

- Make assumptions about the communication delays in the system.

or

- Count the sent and received messages for each process (include this in the local state) and keep track of outstanding messages in the observer process.

Consistent distributed states

Why would we need that?

- Find deadlocks.

- Find termination / completion conditions.

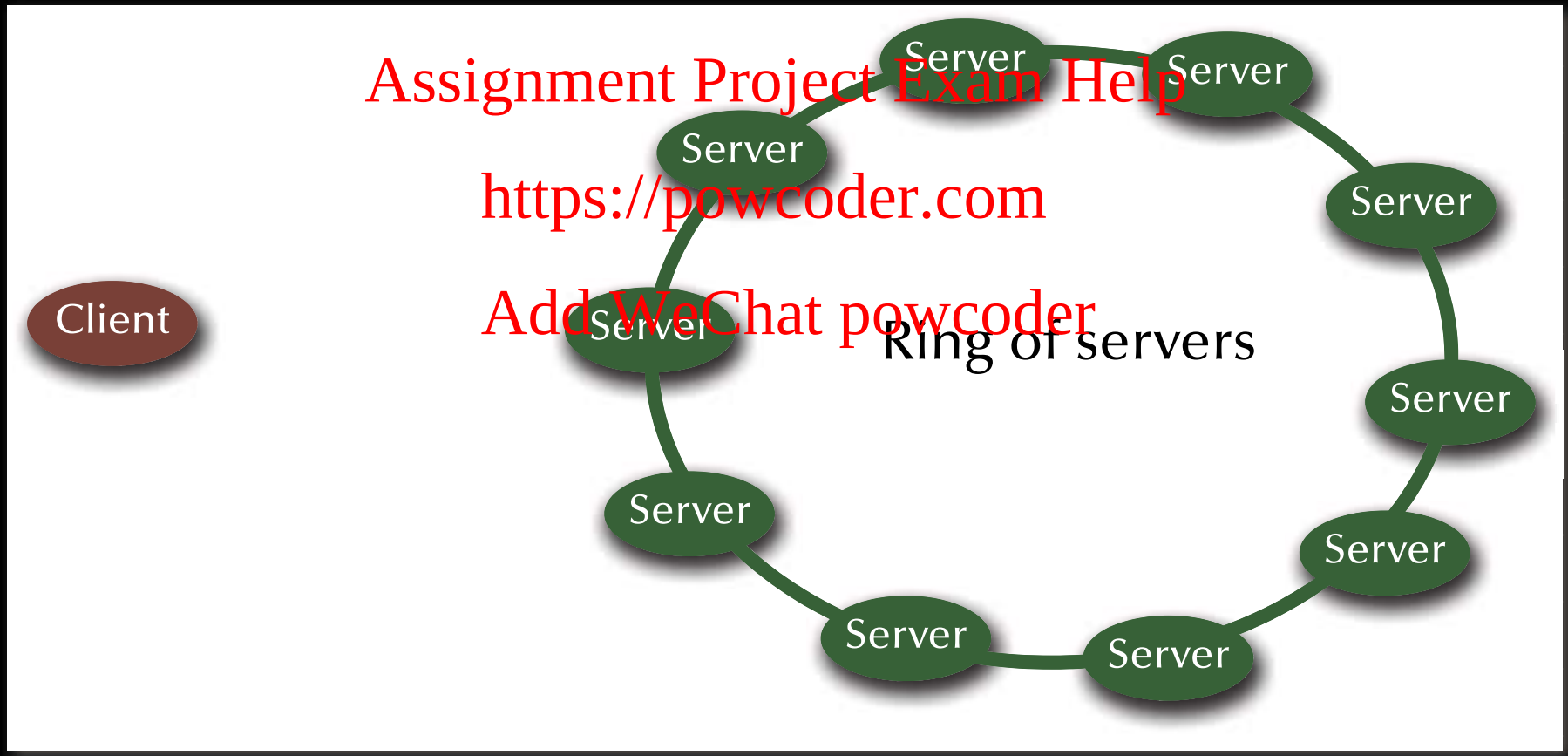
- … any other global safety or liveness property.
- Collect a consistent system state for system backup/restore.
- Collect a consistent system state for further processing (e.g. distributed databases).
- …

Distributed Systems

A distributed server (load balancing)

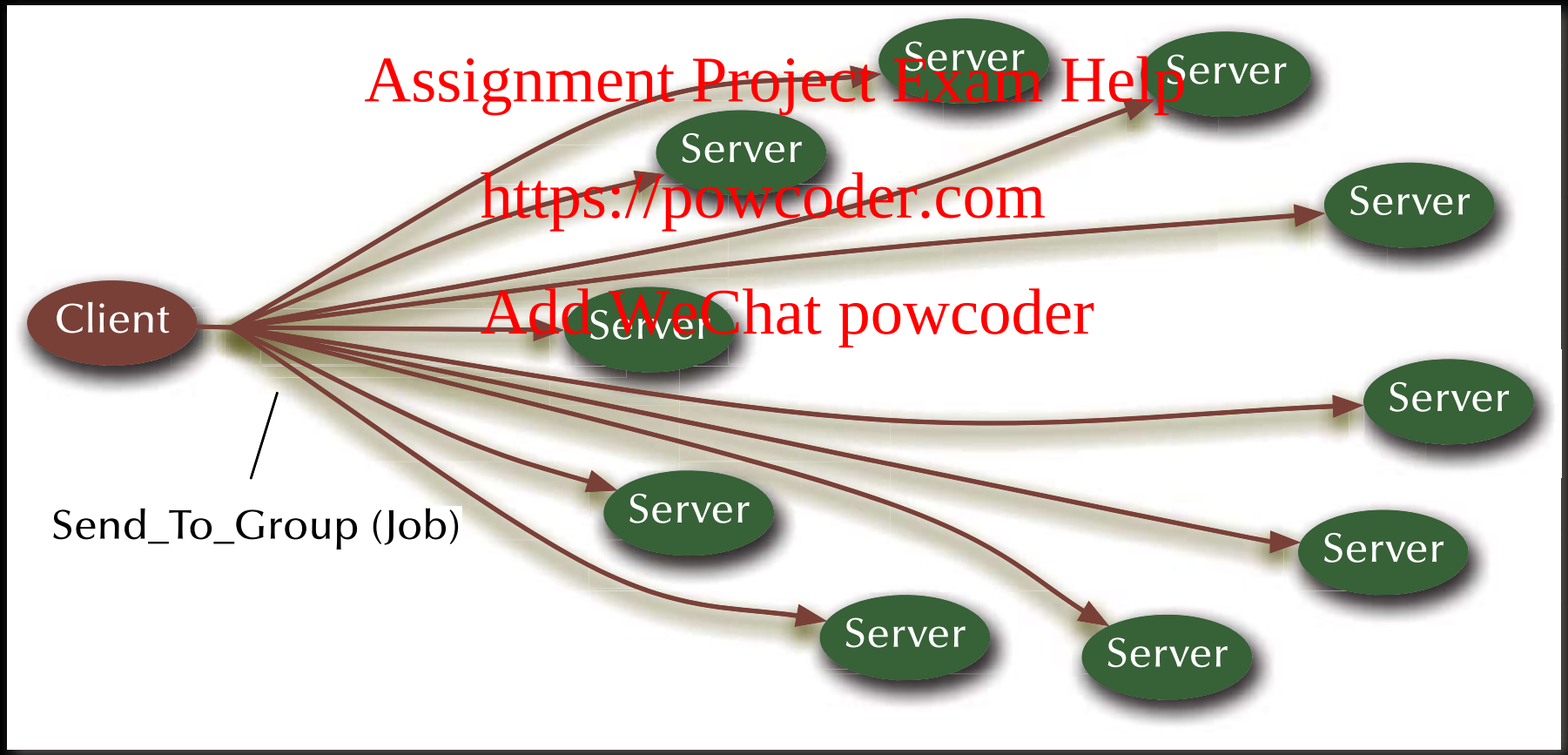
Distributed Systems

A distributed server (load balancing)



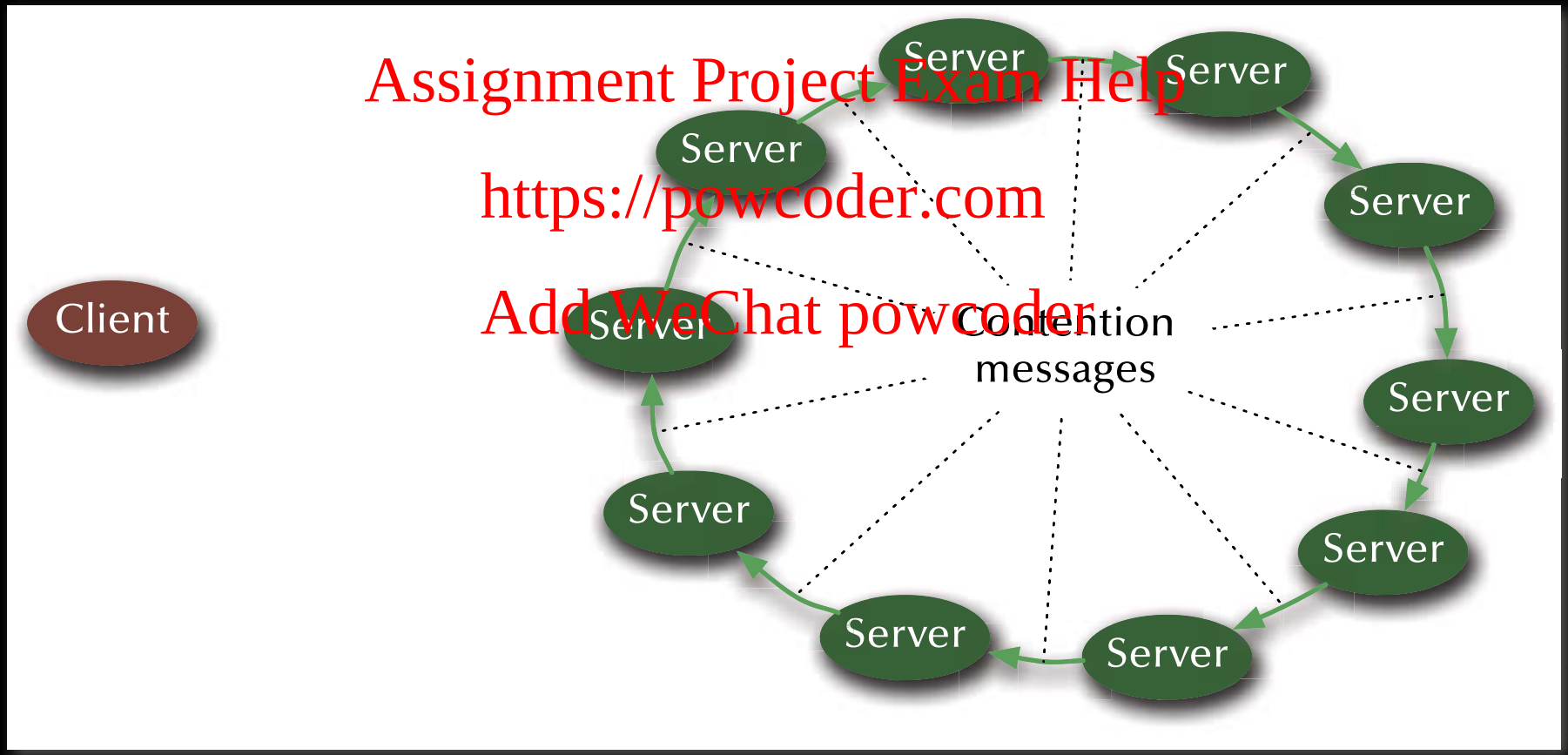
Distributed Systems

A distributed server (load balancing)



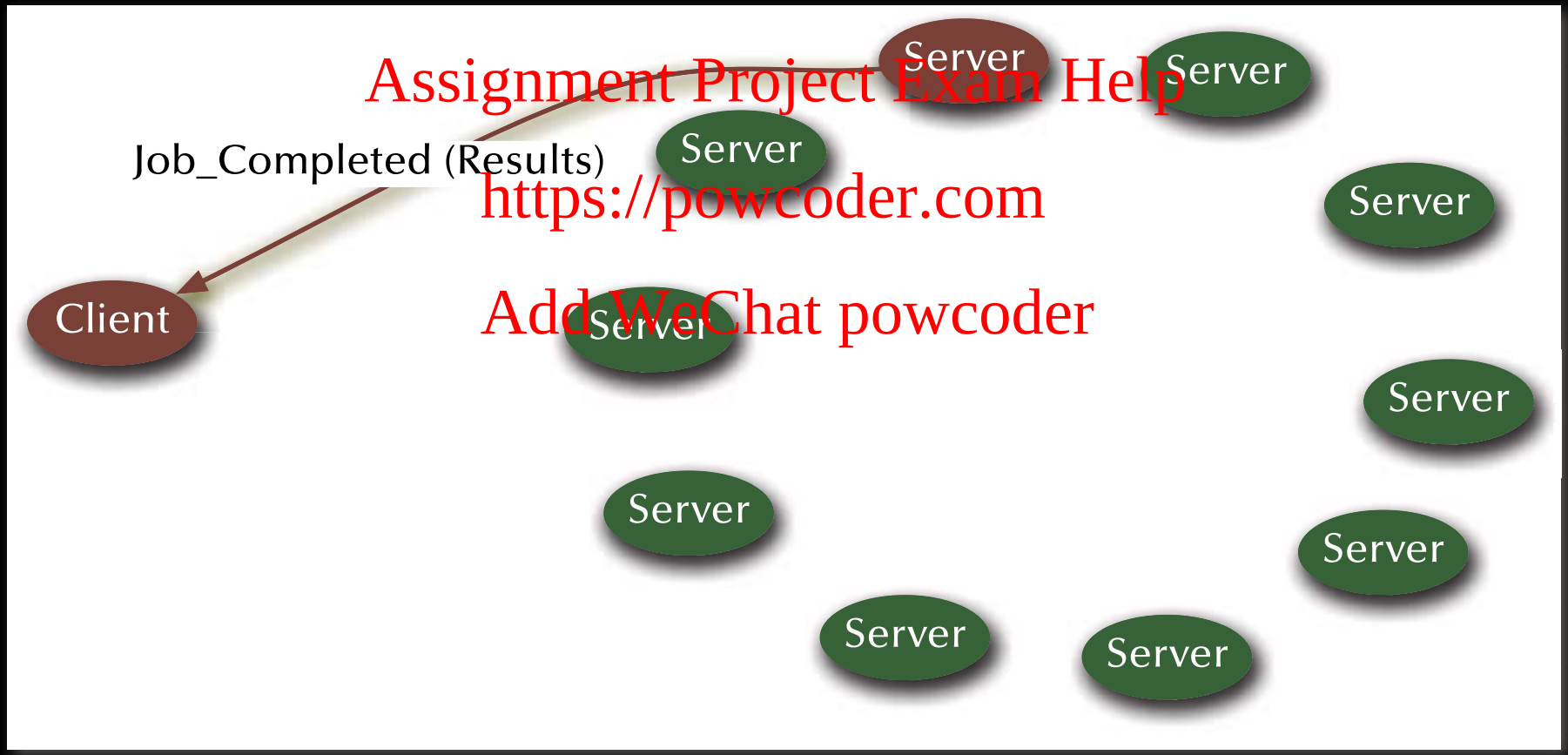
Distributed Systems

A distributed server (load balancing)



Distributed Systems

A distributed server (load balancing)



A distributed server (load balancing)

```
with Ada.Task_Identification; use Ada.Task_Identification;

task type Print_Server is
   entry Send_To_Server (Print_Job : in Job_Type; Job_Done  : out Boolean);
   entry Contention     (Print_Job : in Job_Type; Server_Id : in  Task_Id);
end Print_Server;
```

Distributed Systems

A distributed server (load balancing)

```
task body Print_Server is
   begin
      loop
         select
            accept Send_To_Server (Print_Job : in Job_Type; Job_Done : out Boolean) do
               if not Print_Job in Turned_Down_Jobs then
                  if Not_Too_Busy then
                     Applied_For_Jobs := Applied_For_Jobs + Print_Job;
                     Next_Server_On_Ring.Contention (Print_Job, Current_Task);
                     requeue Internal_Print_Server.Print_Job_Queue;
                  else
                     Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
                  end if;
               end if;
            end Send_To_Server;
```


(...)

Distributed Systems


```
      or
         accept Contention (Print_Job : in Job_Type; Server_Id : in Task_Id) do
            if Print_Job in AppliedForJobs then
               if Server_Id = Current_Task then
                  Internal_Print_Server.Start_Print (Print_Job);
               elsif Server_Id > Current_Task then
                  Internal_Print_Server.Cancel_Print (Print_Job);
                  Next_Server_On_Ring.Contention (Print_Job; Server_Id);
               else
                  null; -- removing the contention message from ring
               end if;
            else
               Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
               Next_Server_On_Ring.Contention (Print_Job; Server_Id);
            end if;
         end Contention;
      or
         terminate;
      end select;
   end loop;
end Print_Server;
```