COMP4057 Distributed and Cloud Computing

Chapter 04

Technical Issues in Distributed Systems (I)

Reading: Textbook Chap 14/15

Learning Outcomes

- Be able to understand the following technical issues in distributed systems and provide basic solutions
  - Time synchronization (part I)
  - Coordination and agreement (part I)
  - Transactions and concurrency control (part II)

Why is Timing Important?

- We often want to measure time accurately.
  - E.g., an e-Commerce transaction involves events at a merchant's computer and at a bank's computer. For auditing purposes, we need to timestamp the events accurately.

- Some basic time unit
  - 1 millisecond: 10^{-3} second
  - 1 microsecond: 10^{-6} second
  - 1 nanosecond: 10^{-9} second
  - 1 picosecond: 10^{-12} second

Clocks

- How to timestamp the events?
  - Assign a date and time of day to each event

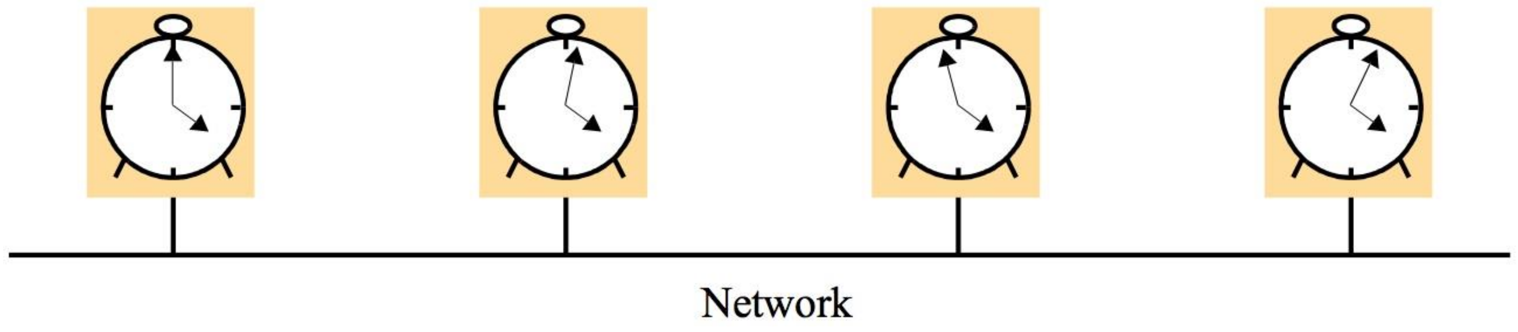
- Computers each have their own physical clocks
  - Electronic devices that count oscillations occurring in a crystal at a frequency, and divide this count and store the result in a register

- Operating system reads the hardware clock value $H(t)$, scales it and adds an offset to produce a software clock $C(t) = \alpha H(t) = \beta$.
  - $C(t)$ is an approximation of the real time t

Clock Skew and Clock Drift

- Computer clocks tend not to be in perfect agreement.

- Clock skew: the instantaneous difference between the readings of any two clocks



Clock Drift

- Clock drift: different crystal-based clocks count time at different rates
  - Even the same clock’s frequency varies with temperature.

- Drift rate: the change in the offset between the clock and a nominal perfect reference clock per unit of time, measured by the reference clock
  - For ordinary clocks based on a quartz crystal, the drift rate is about 10^{-6} seconds/second, i.e., a difference of 1 second every 1,000,000 second (i.e., 11.6 days), or half a minute per year.
  - For high-precision quartz clocks, the drift rate can be 10^{-7} or 10^{-8}.

Coordinated Universal Time

- We may synchronize computer clocks to some external sources with highly accurate time.
  - E.g., atomic clocks use atomic oscillators to achieve drift rate about 10^{-13}.
- UTC: Coordinated Universal Time
  - An international standard for timekeeping
  - UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites (including GPS)
- GPS receivers can have a time accuracy (i.e., clock skew with UTC time) about 1 microsecond.

Synchronizing Physical Clocks

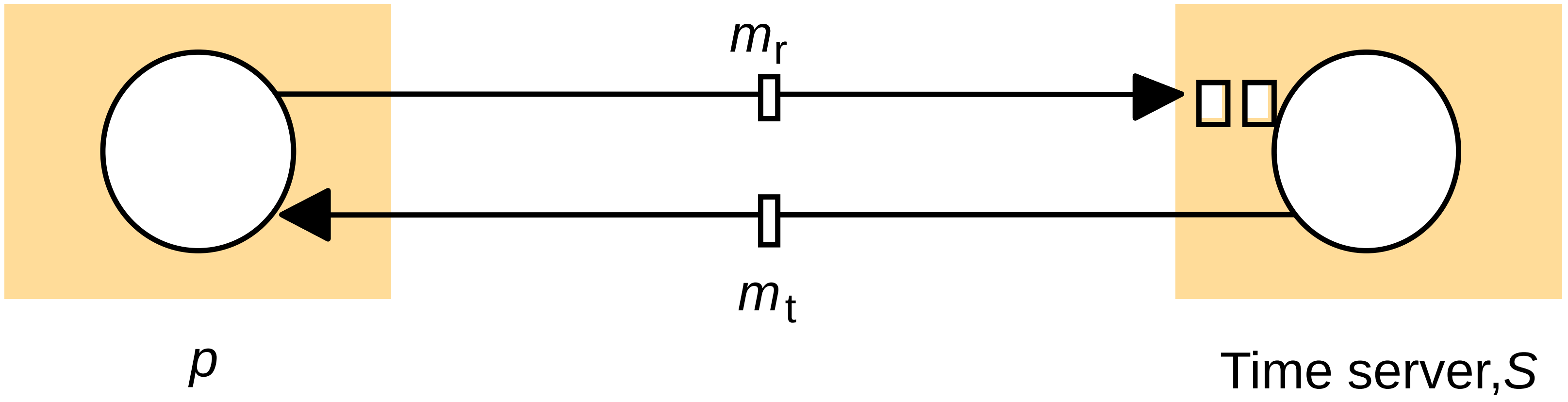
- External synchronization:
  - Synchronize a group of clocks C_i ($1 \leq i \leq N$) with an authoritative external source of time
  - Given source S of UTC time and a synchronization bound $D > 0$, $|S(t) - C_i(t)| < D$ for all i and t

- Internal synchronization:
  - Synchronize a group of clocks C_i ($1 \leq i \leq N$) without any external source of time
  - Given a synchronization bound D > 0, $|C_i(t) - C_j(t)| < D$ for all i, j, t.

External Synchronization by Cristian's Method

- A time server is connected to a device that receives signals from a source of UTC.

1. A client process *p* requests the time in a message m_r.
2. The time server returns the time *t* in a message m_t.
3. Process *p* records the total round-trip time T_{round}.
4. Process *p* sets its local time to $t + T_{round}/2$.

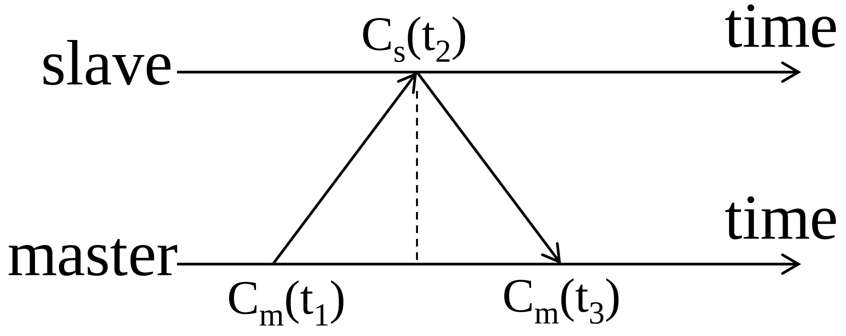


Internal Synchronization by Berkeley Algorithm

- In an intranet, a coordinator computer is chosen to be the **master**. Other computers are **slaves**.
  - — The master will maintain a "**network time**" agreed by all.
- The master **periodically** polls the slaves, and the slaves send back their clock values.
- The master estimates the differences between its own clock and the slaves' clocks by observing the round-trip times, and averages the differences.
  - — Any values far outside the values of the others will be ignored.
  - — This average cancels out the individual clocks' tendencies to run fast or slow.
- The master sets the "network time" as its local time adjusted by the average difference.
- The master adjusts its local time, and informs each slave about its amount of adjustment.
- The slaves adjust their clocks accordingly.

Illustrative Example of Berkeley Algorithm

- At beginning:
  - Master: 3:05
  - Slave 1: 2:55
  - Slave 2: 3:00
- By polling, master detects that
  - The time difference with Slave 1 is -10 seconds
  - The time difference with Slave 2 is -5 seconds
- The master calculates the average difference as:
  - (0-10-5)/3 = -5
- Master sets the “network time” to 3:05 – 5 = 3:00
- Master informs Slave 1 to adjust its clock by 5 seconds
- Master informs Slave 2 to adjust its clock by 0 second



Time difference is:

$$\frac{c_m(t_1) + c_m(t_3)}{2} - c_s(t_2)$$

Internet Synchronization by Network Time Protocol (NTP)

- The NTP service is provided by a network of servers.
  - Primary servers are connected directly to a time source such as radio clock receiving UTC. They are at **stratum 1**, the root of the hierarchy.
  - Stratum 2 servers directly connected to primary servers; stratum 3 synchronize with stratum 2 and so on.
  - If the source of primary servers fails, it will become stratum 2.
- Three modes of synchronization:
  - Multicast
  - Procedure-call [Cristian’s algo]
  - Symmetric mode

NTP

- NTP estimates offset o_i , delay d_i
- If the true offset between the clocks are o and the transmissions time for m = t, m′ = t′

$$T_{i-2} = T_{i-3} + t + o \text{ and } T_i = T_{i-1} + t' - o$$

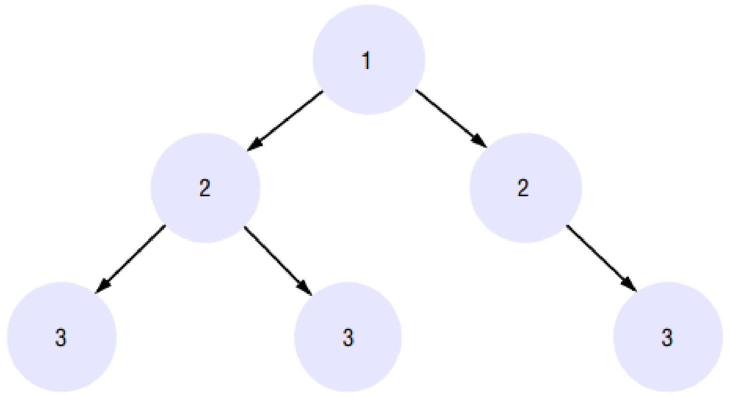
This leads to:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

and:

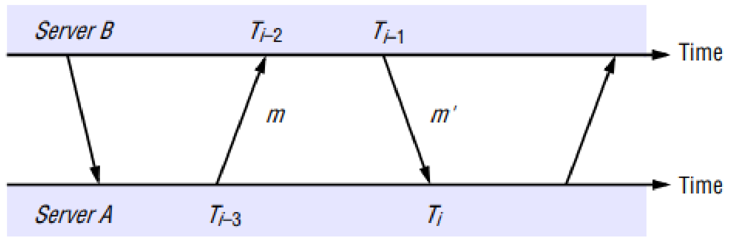
$$o = o_i + (t' - t)/2 \text{, where } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

An example synchronization subnet in an NTP implementation



Arrows denote synchronization control, numbers denote strata.

Messages exchanged between a pair of NTP peers



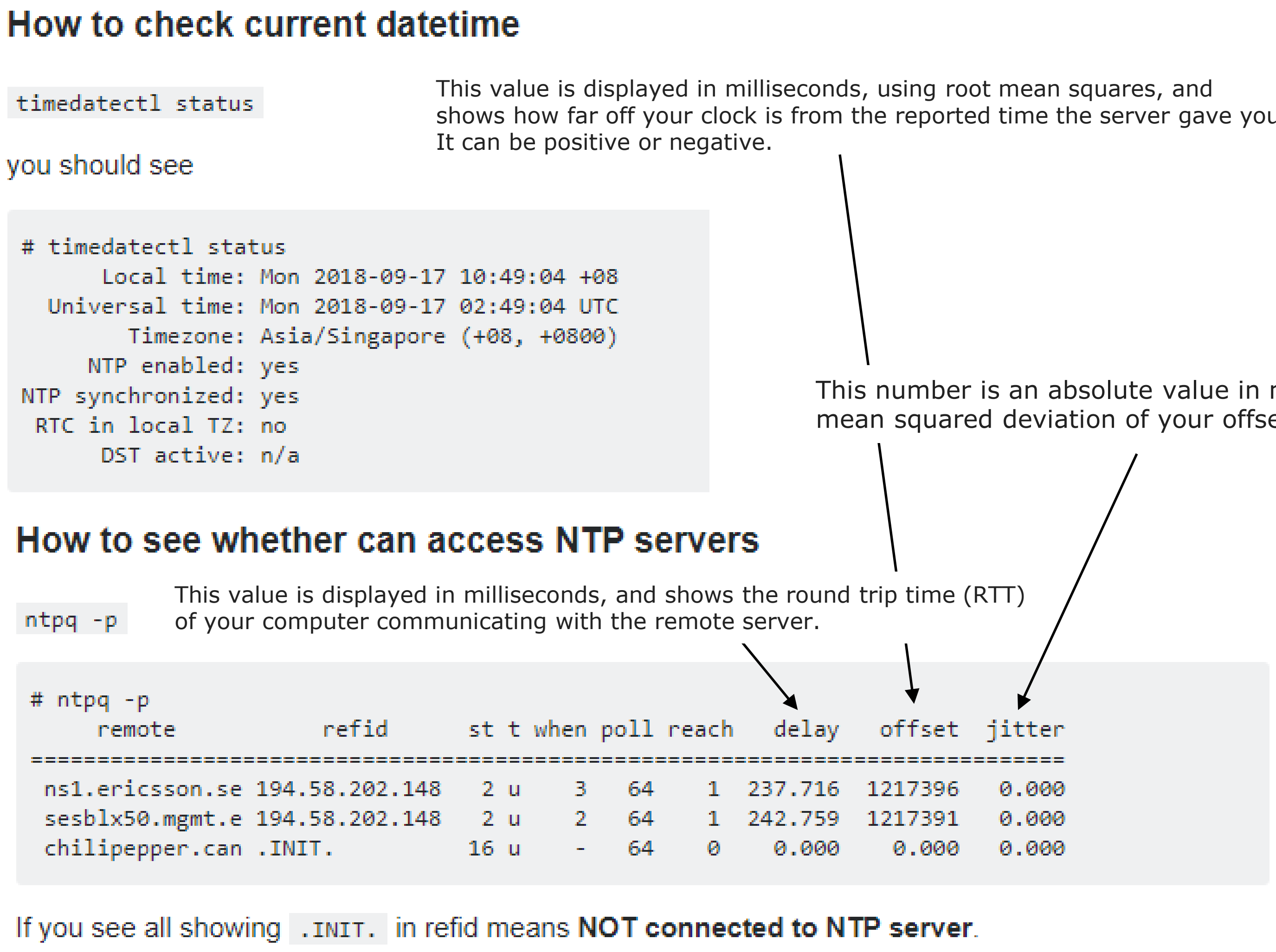
Assume B faster than A by o

Case Study: Fixing System Clock drift in Linux

- Network Time Protocol (NTP) daemon **ntpd** is ran to synchronize the time of the local computer to a NTP server (or other references: radio/satellite receiver/modem).
- Over UDP port 123
- pool.ntp.org has a big virtual cluster of timeservers
- Synchronization is usually done over NTP daemon, but can also be done manually.

```
sudo service ntp stop
sudo date –set="20 Mar 2020" #set a random wrong date as an example
sudo ntpdate -s 0.asia.pool.ntp.org
sudo service ntp start
```

Case Study: Fixing System Clock drift in Linux



How to check current datetime

`timedatectl status`

you should see

```
# timedatectl status
      Local time: Mon 2018-09-17 10:49:04 +08
  Universal time: Mon 2018-09-17 02:49:04 UTC
        Timezone: Asia/Singapore (+08, +0800)
     NTP enabled: yes
NTP synchronized: yes
 RTC in local TZ: no
      DST active: n/a
```

This value is displayed in milliseconds, using root mean squares, and shows how far off your clock is from the reported time the server gave you. It can be positive or negative.

This number is an absolute value in milliseconds, showing the root mean squared deviation of your offsets.

How to see whether can access NTP servers

`ntpq -p`

This value is displayed in milliseconds, and shows the round trip time (RTT) of your computer communicating with the remote server.

```
# ntpq -p
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
 ns1.ericsson.se 194.58.202.148   2 u    3   64    1  237.716  1217396   0.000
 sesblx50.mgmt.e 194.58.202.148   2 u    2   64    1  242.759  1217391   0.000
 chilipepper.can .INIT.          16 u    -   64    0    0.000    0.000   0.000
```

If you see all showing `.INIT.` in refid means **NOT connected to NTP server**.

https://askubuntu.com/questions/942891/unable-to-sync-with-ntp-servers
https://pthree.org/2013/11/05/real-life-ntp/

Process States

- Assume a distributed systems consists of *N* processes, d_i, i = 1, 2, …, *N*
- A process p_i's state s_i includes the values of all its variables.
- A process can take two types of actions:
  - Send/receive operation
  - An operation that transforms its state
- An event *e* is defined as the occurrence of a single action carried out by a process
- $e \rightarrow_i e'$: event *e* occurs before *e'* at p_i
- The history of process p_i: the series of events that take place within it, ordered by $\rightarrow_i$:
  - history(p_i) = $h_i = < e_i^0, e_i^1, e_i^2, \ldots >$

Distributed Mutual Exclusion (Mutex)

- If a collection of processes share a resource (or collection of resources), it is quite often to use **mutual exclusion** to prevent interference and ensure consistency when accessing the resources.
  - E.g., online ticket booking

- Critical section: a part of a multi-process program that may not be concurrently executed by more than one processes

- Distributed mutual exclusion: in distributed system, there is no shared variables and the solution of mutual exclusion is based solely on message passing

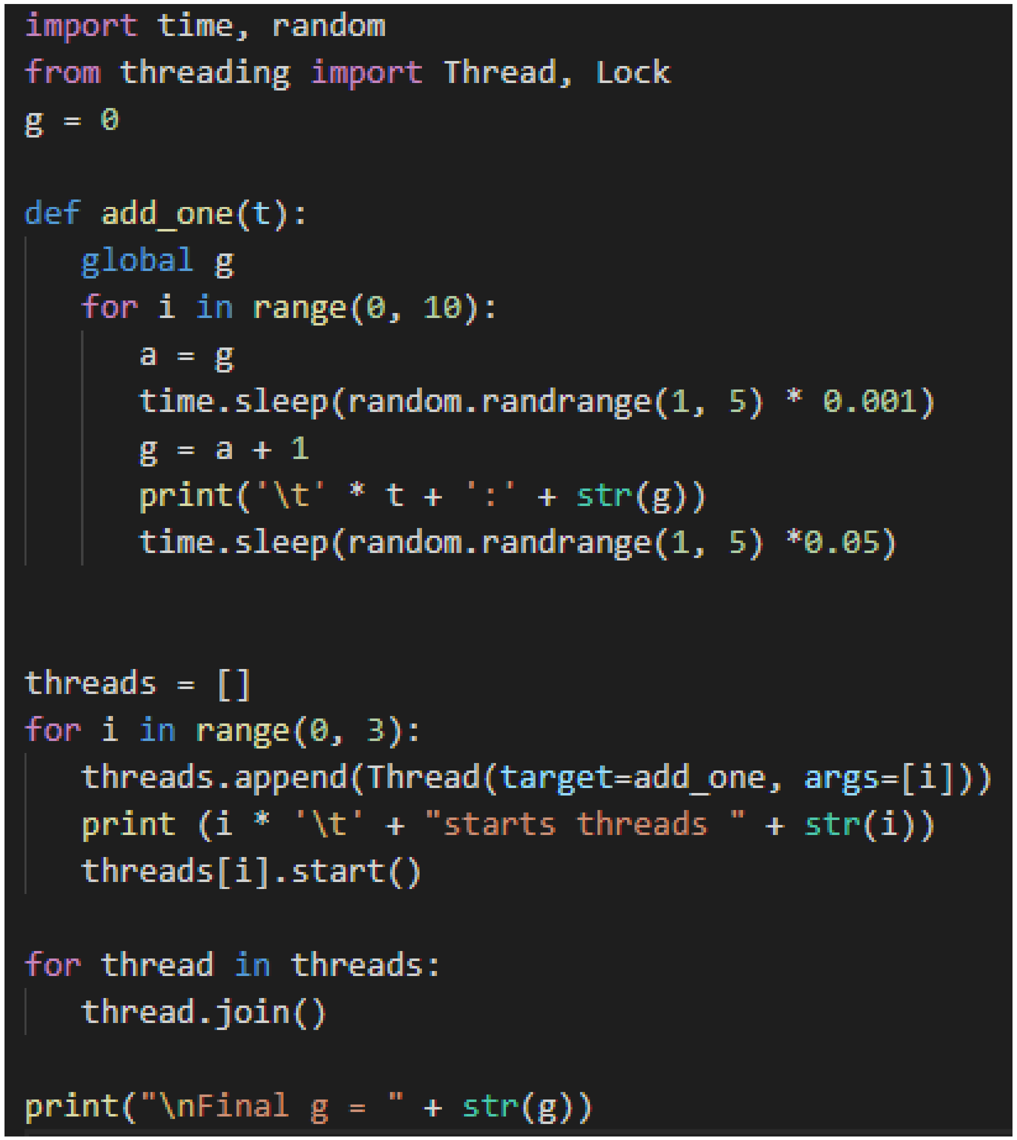
Problem Definition

- Consider a DS of N processes pi, i = 1, 2, …, N that do not share variables. They can access common resources in a critical section as follows:
  1. enter( ); // enter critical section
  2. resourceAccess( ); // access shared resources
  3. exit( ); // leave critical section, other processes may now enter

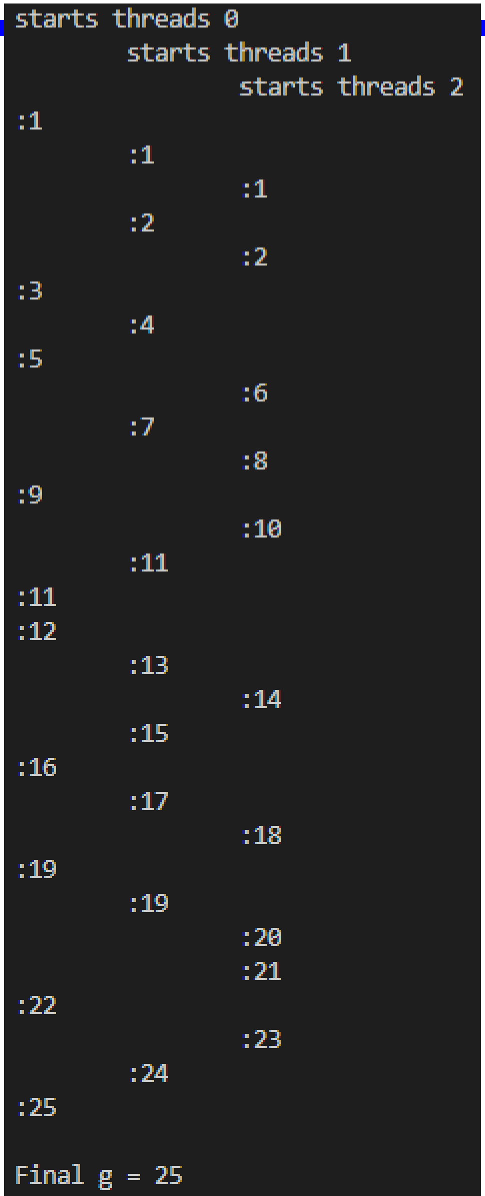
- Essential requirements for mutual exclusion are:
  - ME1: **safety** – at most one process may execute in the critical section at a time
  - ME2: **liveness** – requests to enter and exit the critical section eventually succeed, which implies freedom from both *deadlock* and *starvation*

- Another possible requirement is:
  - ME3: **ordering** – if one request to enter the critical section happened before another, then entry to the critical section is granted in that order

Example of a racing

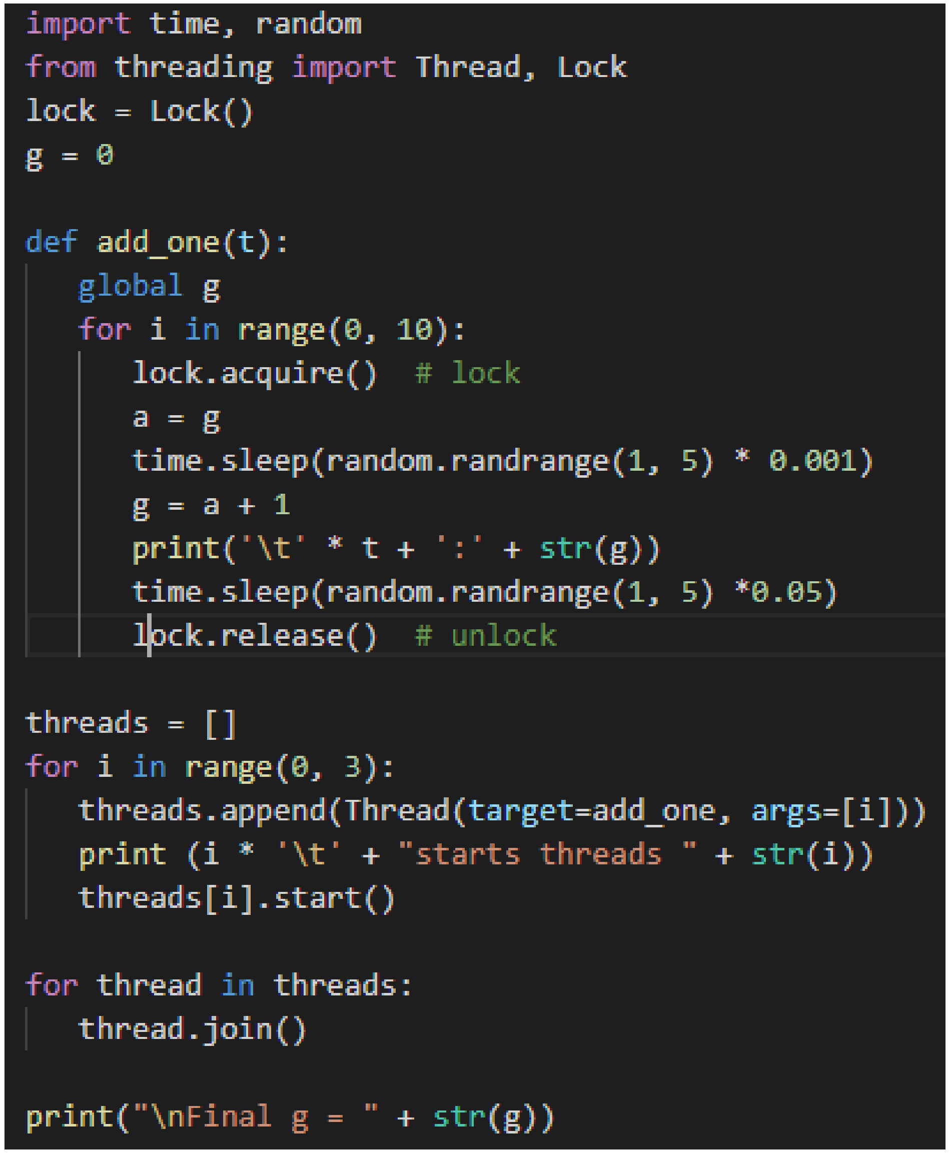


```python
import time, random
from threading import Thread, Lock
g = 0

def add_one(t):
    global g
    for i in range(0, 10):
        a = g
        time.sleep(random.randrange(1, 5) * 0.001)
        g = a + 1
        print('\t' * t + ':' + str(g))
        time.sleep(random.randrange(1, 5) *0.05)


threads = []
for i in range(0, 3):
    threads.append(Thread(target=add_one, args=[i]))
    print (i * '\t' + "starts threads " + str(i))
    threads[i].start()

for thread in threads:
    thread.join()

print("\nFinal g = " + str(g))
```

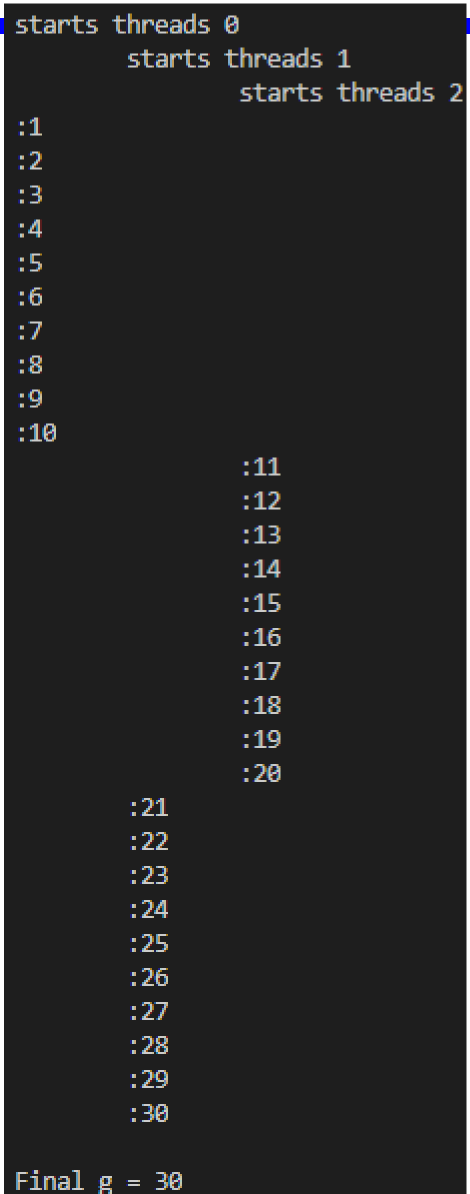


```
starts threads 0
        starts threads 1
                starts threads 2
:1
        :1
                :1
        :2
                :2
:3
        :4
:5
                :6
        :7
                :8
:9
                :10
        :11
:11
:12
        :13
                :14
        :15
:16
        :17
                :18
:19
        :19
                :20
                :21
:22
                :23
        :24
:25

Final g = 25
```

Example of mutex /w starvation

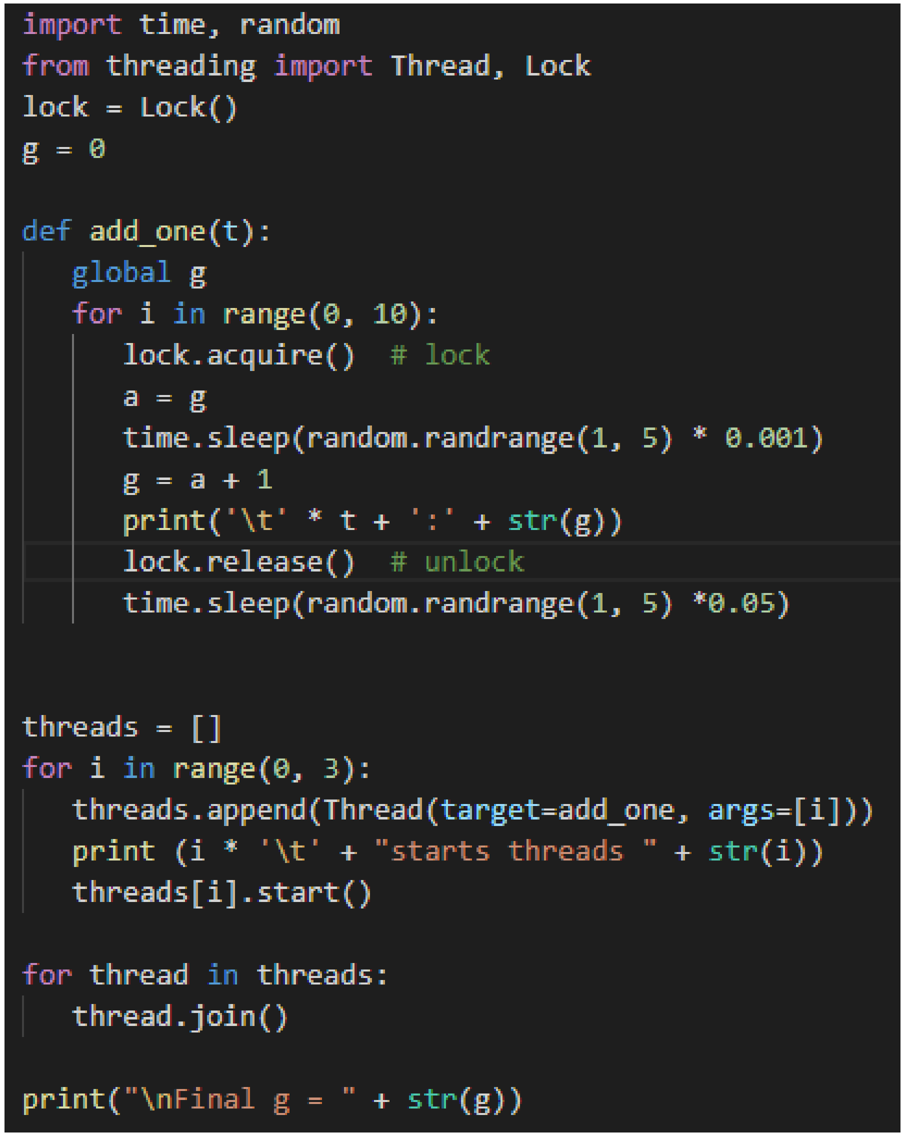


```python
import time, random
from threading import Thread, Lock
lock = Lock()
g = 0

def add_one(t):
    global g
    for i in range(0, 10):
        lock.acquire()  # lock
        a = g
        time.sleep(random.randrange(1, 5) * 0.001)
        g = a + 1
        print('\t' * t + ':' + str(g))
        time.sleep(random.randrange(1, 5) *0.05)
        lock.release()  # unlock

threads = []
for i in range(0, 3):
    threads.append(Thread(target=add_one, args=[i]))
    print (i * '\t' + "starts threads " + str(i))
    threads[i].start()

for thread in threads:
    thread.join()

print("\nFinal g = " + str(g))
```

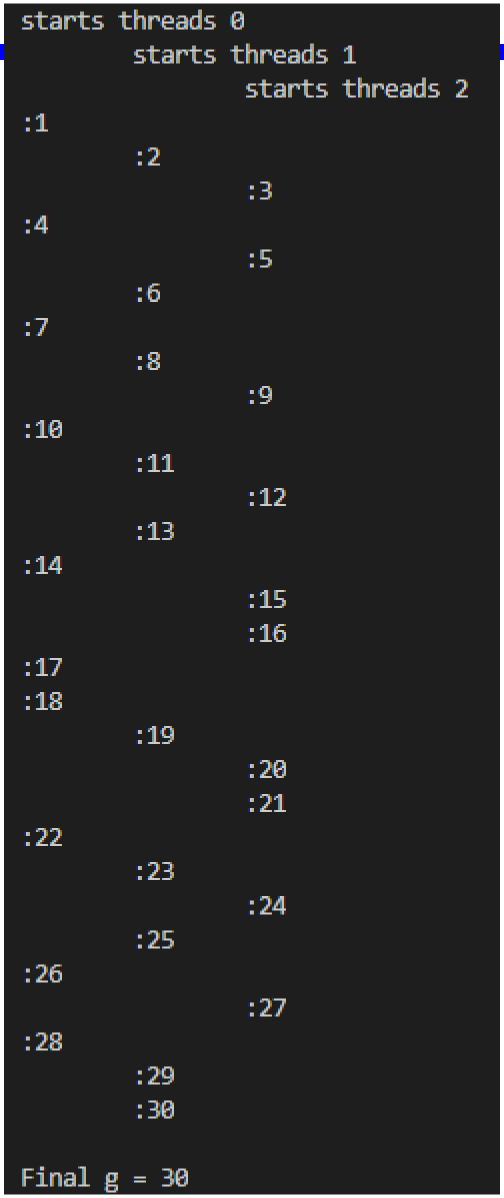


```
starts threads 0
        starts threads 1
                starts threads 2
:1
:2
:3
:4
:5
:6
:7
:8
:9
:10
                :11
                :12
                :13
                :14
                :15
                :16
                :17
                :18
                :19
                :20
        :21
        :22
        :23
        :24
        :25
        :26
        :27
        :28
        :29
        :30

Final g = 30
```

Example of a mutex

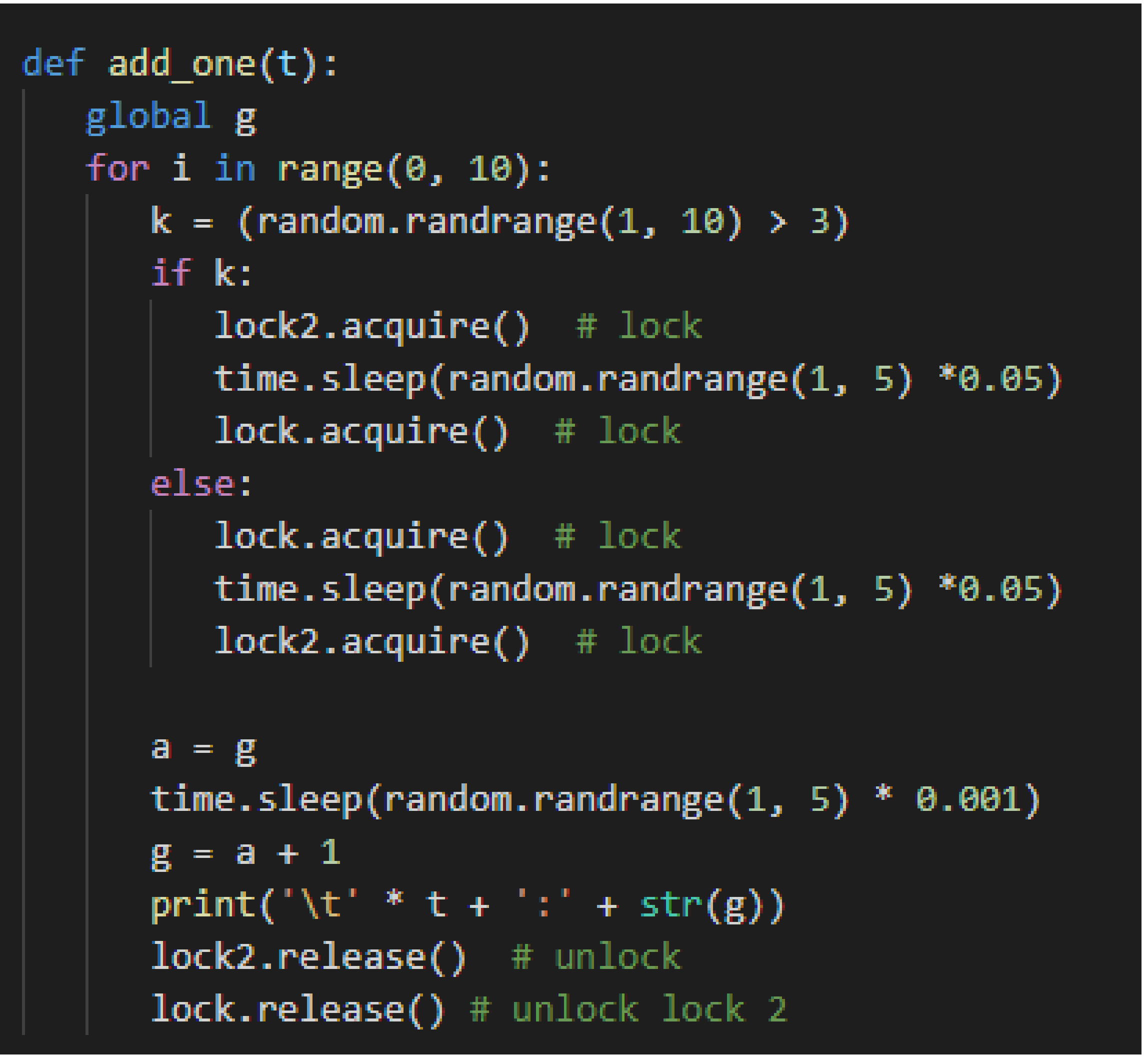


```python
import time, random
from threading import Thread, Lock
lock = Lock()
g = 0

def add_one(t):
    global g
    for i in range(0, 10):
        lock.acquire()  # lock
        a = g
        time.sleep(random.randrange(1, 5) * 0.001)
        g = a + 1
        print('\t' * t + ':' + str(g))
        lock.release()  # unlock
        time.sleep(random.randrange(1, 5) *0.05)


threads = []
for i in range(0, 3):
    threads.append(Thread(target=add_one, args=[i]))
    print (i * '\t' + "starts threads " + str(i))
    threads[i].start()

for thread in threads:
    thread.join()

print("\nFinal g = " + str(g))
```

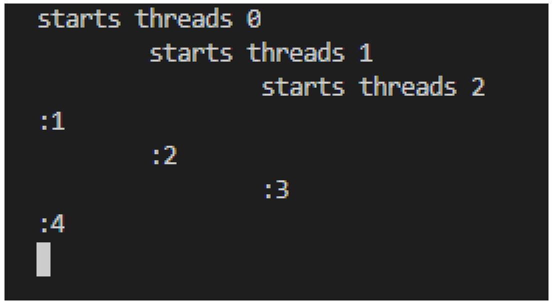


```
starts threads 0
        starts threads 1
                starts threads 2
:1
        :2
                :3
:4
                :5
        :6
:7
        :8
                :9
:10
        :11
                :12
        :13
:14
                :15
                :16
:17
:18
        :19
                :20
                :21
:22
        :23
                :24
        :25
:26
                :27
:28
        :29
        :30

Final g = 30
```

Example of a deadlock



```python
def add_one(t):
    global g
    for i in range(0, 10):
        k = (random.randrange(1, 10) > 3)
        if k:
            lock2.acquire()  # lock
            time.sleep(random.randrange(1, 5) *0.05)
            lock.acquire()  # lock
        else:
            lock.acquire()  # lock
            time.sleep(random.randrange(1, 5) *0.05)
            lock2.acquire()  # lock

        a = g
        time.sleep(random.randrange(1, 5) * 0.001)
        g = a + 1
        print('\t' * t + ':' + str(g))
        lock2.release()  # unlock
        lock.release() # unlock lock 2
```

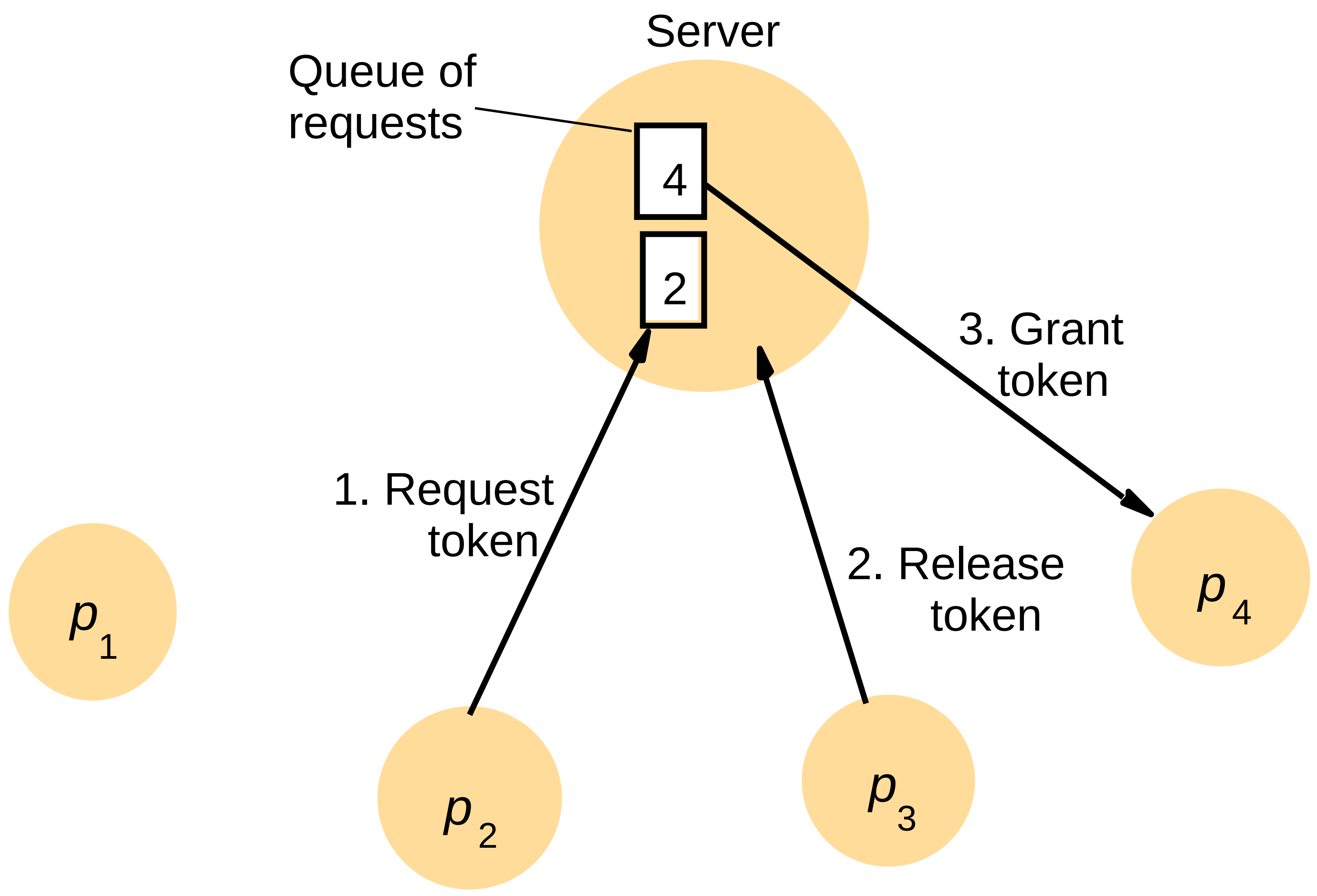


```
starts threads 0
        starts threads 1
                starts threads 2
:1
        :2
                :3
:4
█
```

Performance Concerns

- Algorithms for distributed mutual exclusion can be evaluated by the following criteria:
  - Consumed ***bandwidth***: the number of messages sent in each enter() and exit() operation
  - ***Client delay***: the incurred delay by a process at each enter() and exit() operation
  - System ***throughput***: the rate at which the collection of processes as a whole can access the critical section. This can be measured by ***synchronization delay*** between one process existing the critical section and the next process entering it.

Central Server Algorithm



REF1 Fig. 15.2 Server managing a mutual exclusion token for a set of processes

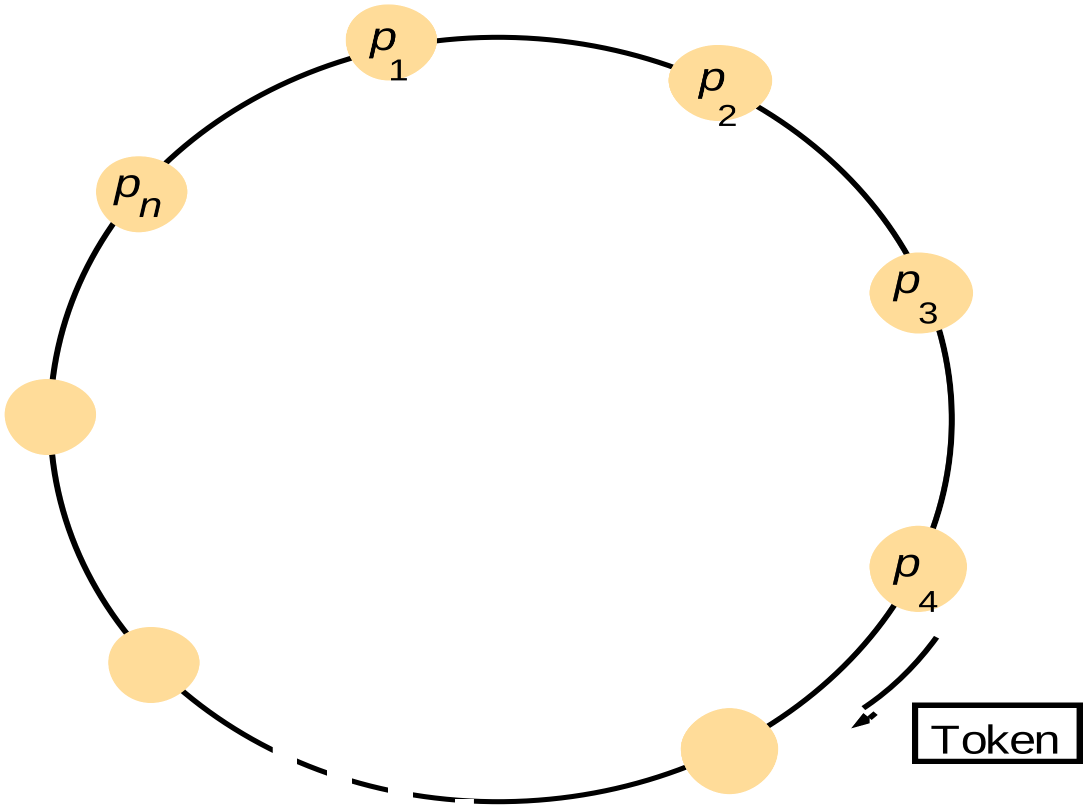
Central Server Algorithm

- Idea: employ a server that grants permission to enter the critical section
- A client process's actions:
  - To enter a critical section, a process sends a request message to the server and awaits a reply.
  - It a client process receives the token from the server, it can enter the critical section.
  - If a process exits the critical section, it sends a message to the server, giving it back the token.
- The server's actions:
  - 1. Upon receiving a token request:
    - If no other process has the token, then replies immediately, granting the token;
    - If the token is currently held by another process, the server does not reply, but queues the request
  - 2. Upon receiving a token release message:
    - If the queue is not empty, removes the oldest entry from the queue, and replies to the corresponding process, granting the token

Performance Analysis

- Bandwidth cost:
  - It requires two messages to enter the critical section: a request message and a grant message
  - It requires one message to exit the critical section: a release message
- Client delay:
  - a round-trip delay for entering the critical section
  - No delay for exiting process, assuming asynchronous message passing
- Synchronization delay: the time for a release message to the server and a grant message to the next process

A Ring-based Algorithm



REF1 Fig. 15.3 A ring of processes transferring a mutual exclusion token

A Ring-based Algorithm

- The *N* processes are arranged as a logical ring:
  - Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$
  - Exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction around the ring
- If a process doesn’t required to enter the critical section when it receives the token, it simply forwards the token to the next process in the ring.
- A process that requires the token waits until it receives it, then enters the critical section, exits, and sends the token to the next.

Performance Analysis

- Bandwidth cost:
  - The algorithm continuously consumes network bandwidth, even when no process requires entry to the critical section
- Client delay:
  - A delay of 0~*N* message transmissions for entering the critical section
  - One message delay for exiting the critical section
- Synchronization delay:
  - Can be from 1 to *N* message transmissions

Elections

- An **election algorithm** is used for choosing a unique process from a collection of processes to play a particular role.
  - E.g., in the server based mutual exclusion algorithm, how to select the server?
- A process *calls the election* if it initiates a particular run of the election algorithm.
  - In principle, *N* processes could call *N* concurrent elections.
- A process is a *participant* if it is engaged in some run of the election algorithm, or a *non-participant* if it is not engaged in any election.

Problem Definition

- The elected process is unique, even if several processes call elections concurrently.
  - Consider the scenario that two processes detect the failure of a coordinator process around the same time.
- Without loss of generality, we want to elect the process that has the largest identifier.
  - We assume each process has a unique identifier.
- Each process uses variable ***$elected_i$*** to represent the identifier of the elected process.

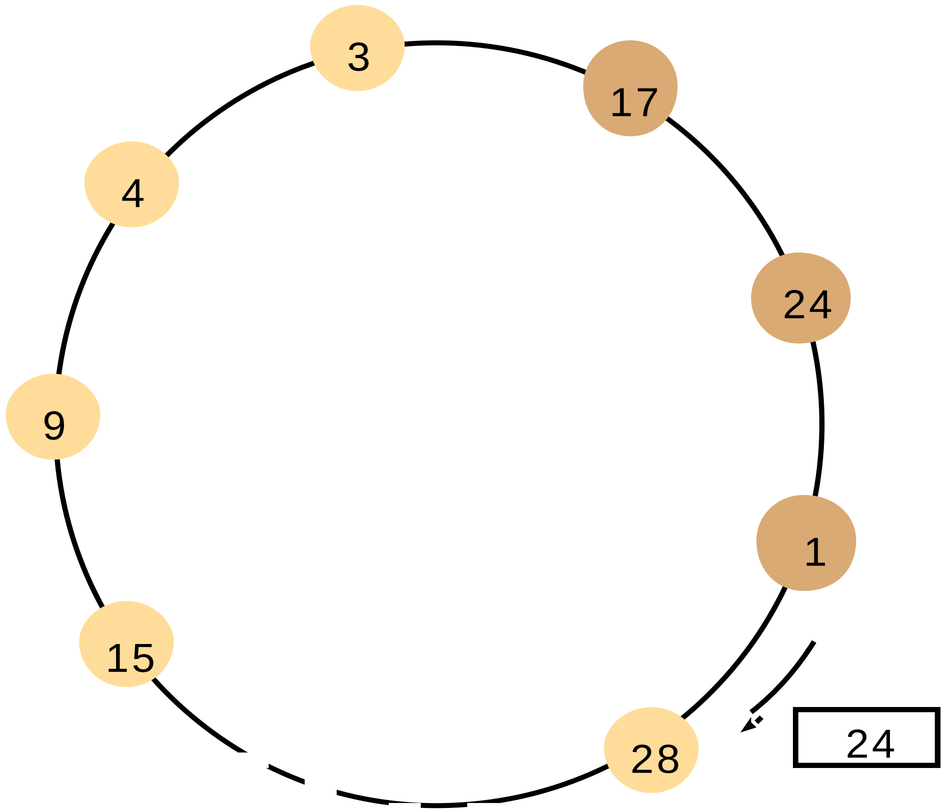
A Ring-based Election Algorithm (Cont.)

- Arrange a collection of N processes as a logical ring.
  - Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$
  - All messages are sent clockwise around the ring.
  - Assume no failures occur.
- Initially, every process is marked as ***non-participant***.
- Any process can begin an election, by marking itself as a ***participant***, and placing its identifier in an ***election message***, sending it to its clockwise neighbour.

A Ring-based Election Algorithm (Cont.)

- When a process receives an ***election message***, it compares the identifier in the message with its own.
  - If the arrived identifier is greater, it forwards the message to its neighbour. It also marks itself as a participant.
  - If the arrived identifier is smaller, two cases:
    - if the receiver is not a participant, then it substitutes its own identifier in the message and forwards it. It also marks itself as a participant.
    - if the receiver is already a participant, it does nothing.
  - If the arrived identifier is that of the receiver itself, it becomes the coordinator. It marks itself as a non-participant and sends an ***elected message*** to its neighbour.
- When a process receives an ***elected message***, it marks itself as non-participant, sets its variable $\boldsymbol{elected_i}$ to the identifier in the message, and forwards the message to its neighbour unless it is the new coordinator.

A Ring-based Election Algorithm



Note: the election was started by process 17. The highest process identifier encountered so far is 24.
Participant processes are shown in a dark tint.

REF1 Fig. 15.7: A ring-base election in progress

Performance Analysis

- Consider that only a single process starts an election.

- How many messages are required in the worst-case?
  - If the anti-clockwise neighbour of the initial process has the highest identifier, it takes $3N-1$ messages: $2N-1$ election messages and N elected messages

- Turnaround time: since the messages are sent sequentially, the worst-case turnaround time is also $3N-1$

The Bully Algorithm

- The ring-based algorithm assumes no process failure.
- The bully algorithm assumes that process failure can be detected by timeout.
  - Assume maximum message transmission delay is T_{trans}, the maximum message processing delay is $T_{process}$, then $T=2T_{trans}+T_{process}$ can be used as a timeout bound to detect process failure.
- The bully algorithm also assumes that each process knowns which processes have higher identifiers, and it can communicate with all such processes.
  - Basic idea: a process tries to find out whether there is any other non-crashed process with higher identifier.

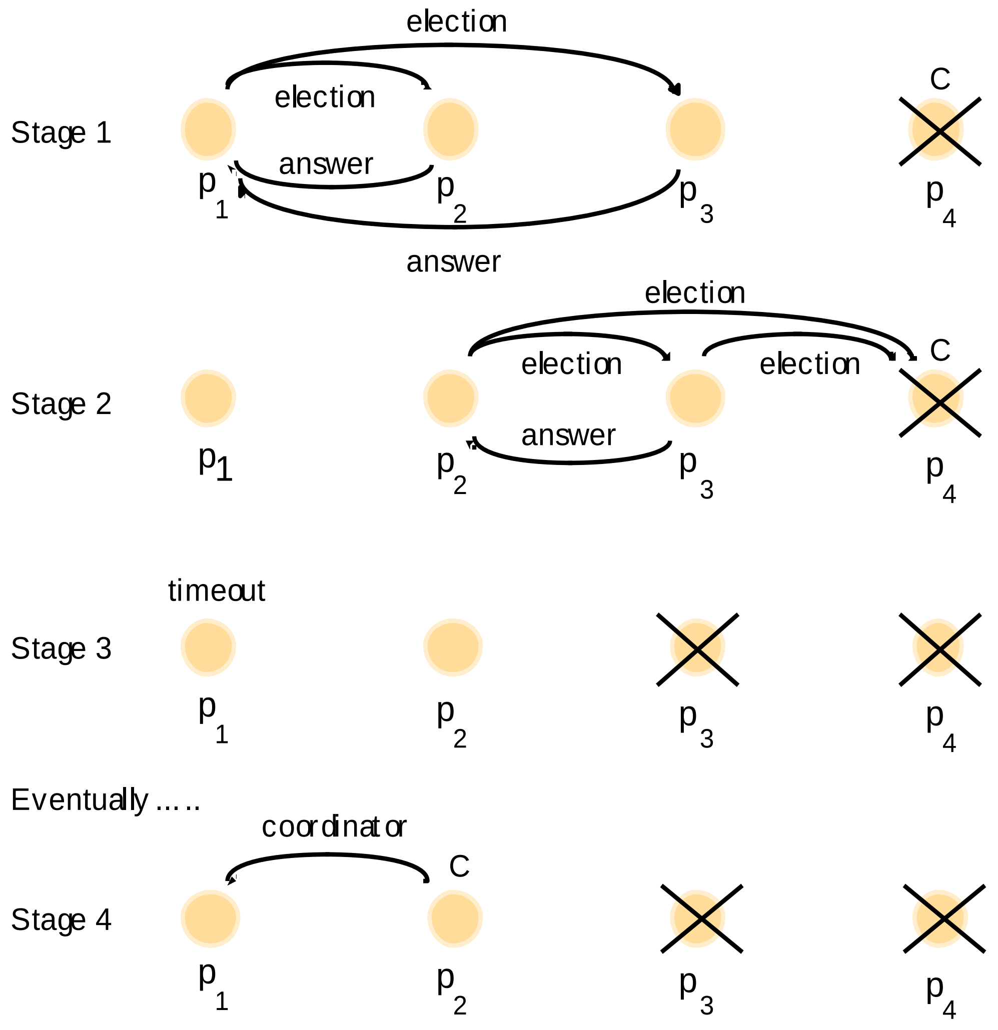
The Bully Algorithm (Cont.)

- There are three types of messages
  - Election message: to announce an election
  - Answer message: in response to an election message
  - Coordinator message: to announce the identity of the elected process (i.e., the new coordinator)
- If any process detects that the existing coordinator has failed (by timeout), it may start the election process.
  - Case 1: If this process happens to have the highest identifier, it will directly send "***coordinator message***" to all processes.
  - Case 2: Otherwise, it sends "***election message***" to all processes that have higher identifiers than itself, and awaits answer messages in response.

The Bully Algorithm (Cont.)

- In Case 2:
  - If no response within time *T*, it considers itself the coordinator and sends "***coordinator message***" to all processes.
  - Otherwise, it waits a further period *T′* for a coordinator message from the new coordinator. If none arrives, it begins another round of election.
- If a process receives a ***coordinator message***, it sets its variable *elected*$_i$ accordingly.
- If a process receives an ***election message***, it sends back an ***answer message*** and begins another election.

The Bully Algorithm (Cont.)



Stage 1: p_1 detects the failure of coordinator p_4 and begins an election. p_2 and p_3 both respond to p_1 and begin their own election.

Stage 2: p_2 gets an answer from p_3, but p_3 gets no response from p_4. So p_3 sets itself to coordinator. But it crashes before it informs others.

Stage 3: When p_1's timeout period *T'* expires, it begins a new round of election.

Stage 4: p_2 determines itself as coordinator, and sends a coordinator message to p_1.

REF1 Fig. 15.8: The bully algorithm

Performance Analysis

- In the best case, the process with the second-highest identifier notices the coordinator’s failure and begins the election.
  - Only N-2 coordinator messages are sent.
  - The turnaround time is one message.

- In the worst case, the process with the lowest-identifier begins the election
  - O(N^2) messages are required.