CSci 5105

Introduction to Distributed Systems

Mutual Exclusion

Last Time

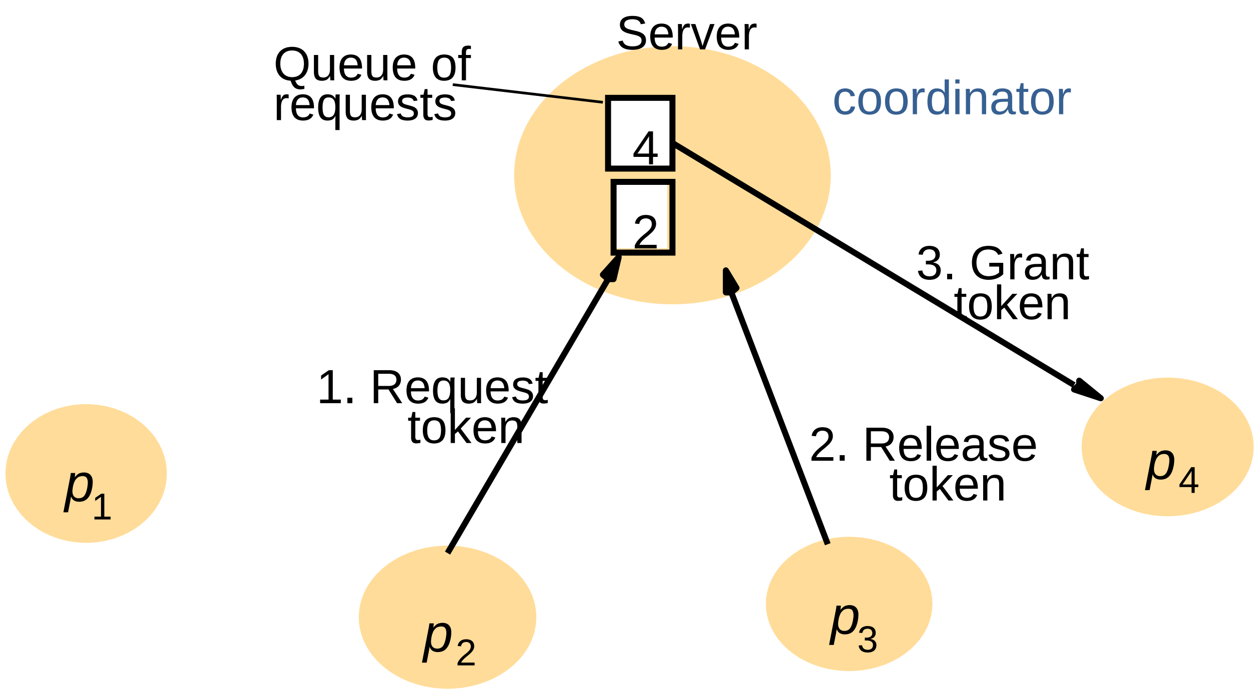
- Synchronization
- Global clocks
- Logical clocks

Mutual Exclusion

- Only 1 process can do something at a time
- Generalize from fine-grained mutual exclusion (shared data access)

Simple Token Based

- Single server grants token
- Tracks who wants token



Options

- Holding
  - Node explicitly acquires, releases
  - Server leases for a fixed time T
- Server
  - Ignore requests, if token is held
  - Or block for to service later

Next Option: Distributed Algorithm

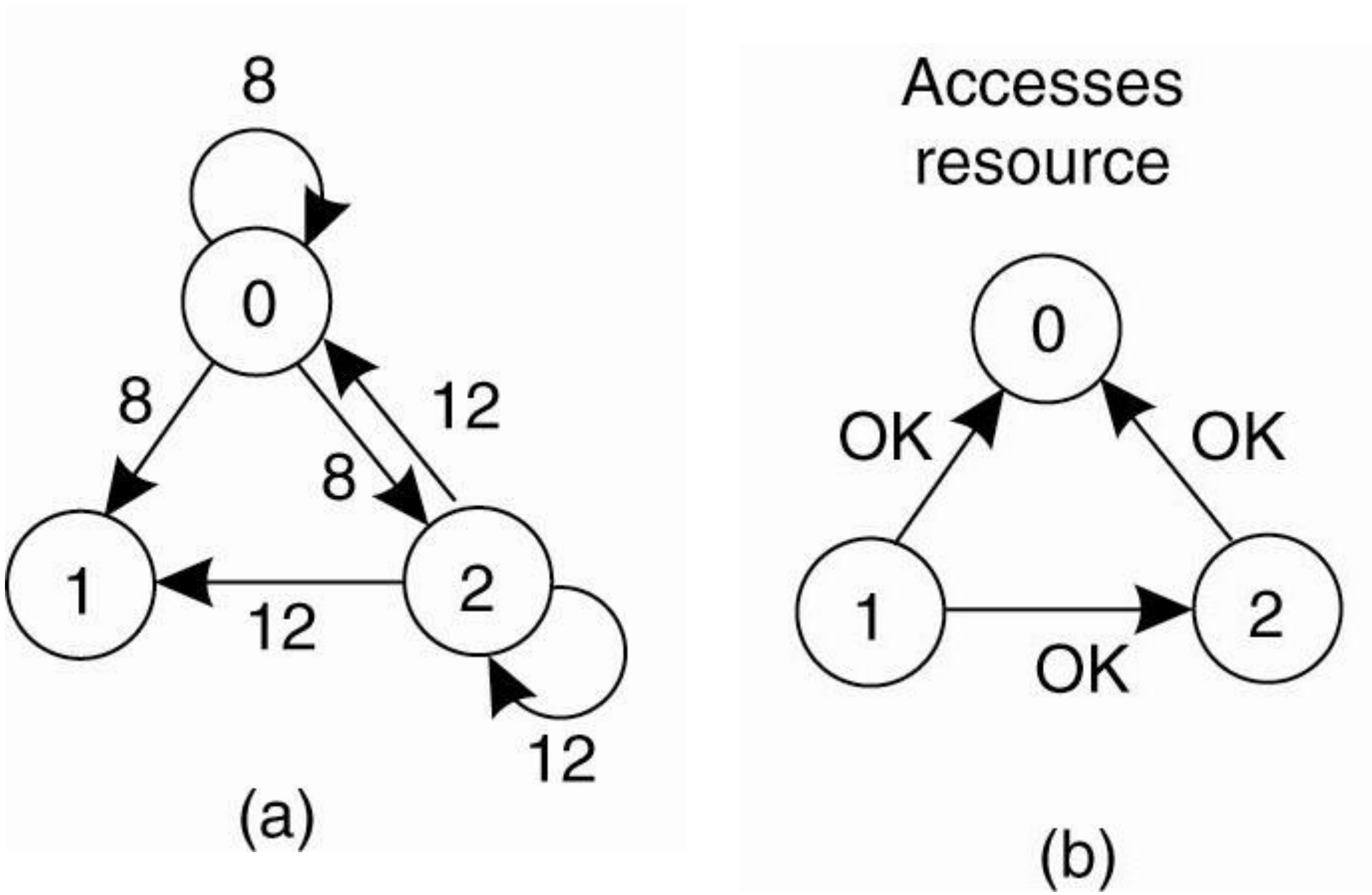
Based on multicast to a group: consensus

1. Sender multicasts interest in the resource with TS

1. Receiver receives request
    a. If the receiver is already using resource, it simply does not reply. Instead, it queues the request

    b. If the receiver is already trying access the resource, it compares the TS of the incoming request with the TS in its earlier request. The lowest TS wins.

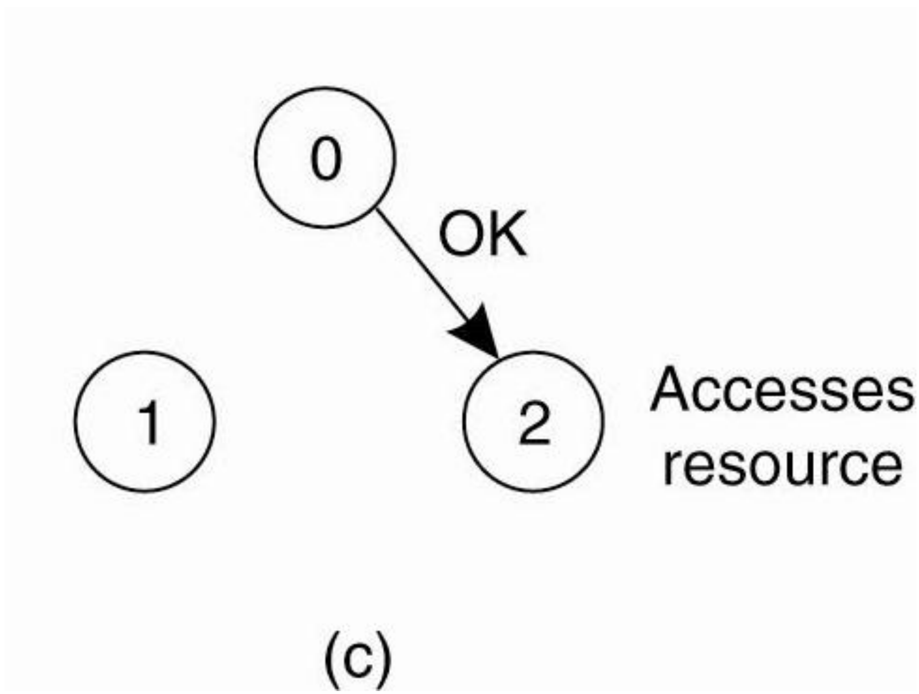
Example



- Process 0 has the lowest TS, so it wins.

Example (con't)

- When process 0 is done,
  it sends an OK also, so 2 can now go ahead.



Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

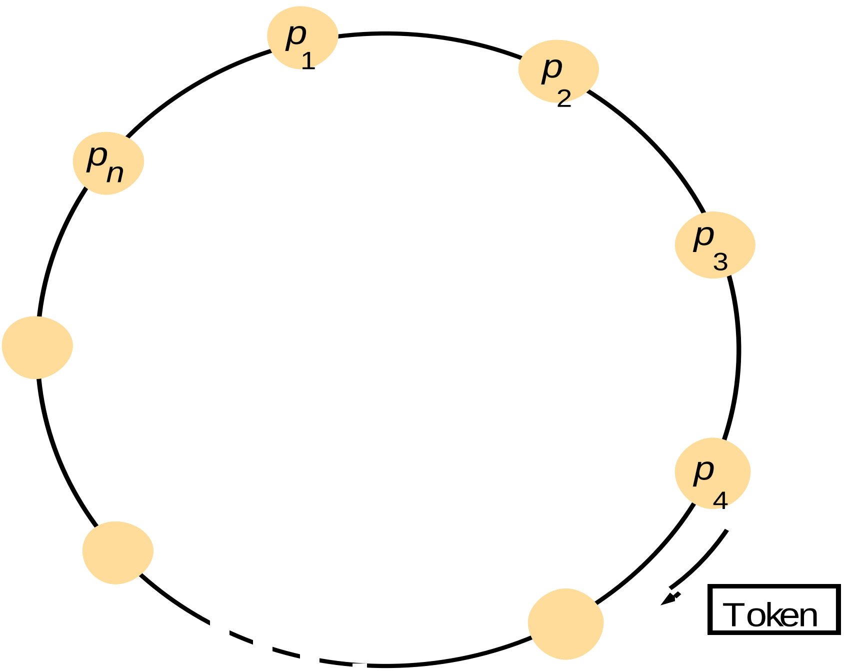
Maekawa's algorithm

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;
On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;
On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Next Option: Ring

- Token circulates around the ring
- If is passes to you
  - keep it (if you want it)
  - don't keep it (pass it along)



A Comparison of the Four Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
[redacted]			
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

Election

- Need to pick a leader
  - Example: coordinator for mutual exclusion
- Assumption
  - Everyone knows everyone but not who is up or down
  - Everyone knows everyone's unique process number
- Bully Algorithm
  - Initiated by a process that detects coordinator is down

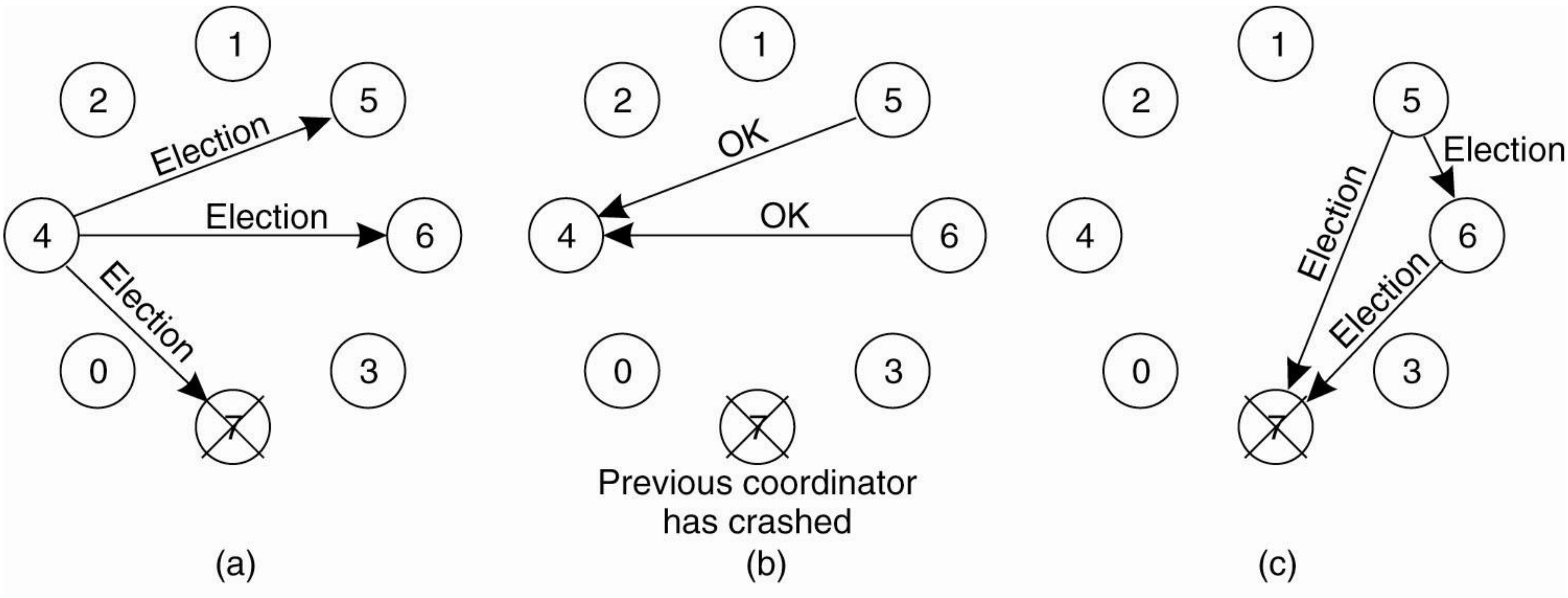
The Bully Algorithm

1. *P* sends an *ELECTION* message to all processes with higher numbers

2. If no one responds, *P* wins the election and becomes coordinator

3. If one of the higher-ups answers, it takes over. *P*'s job is done

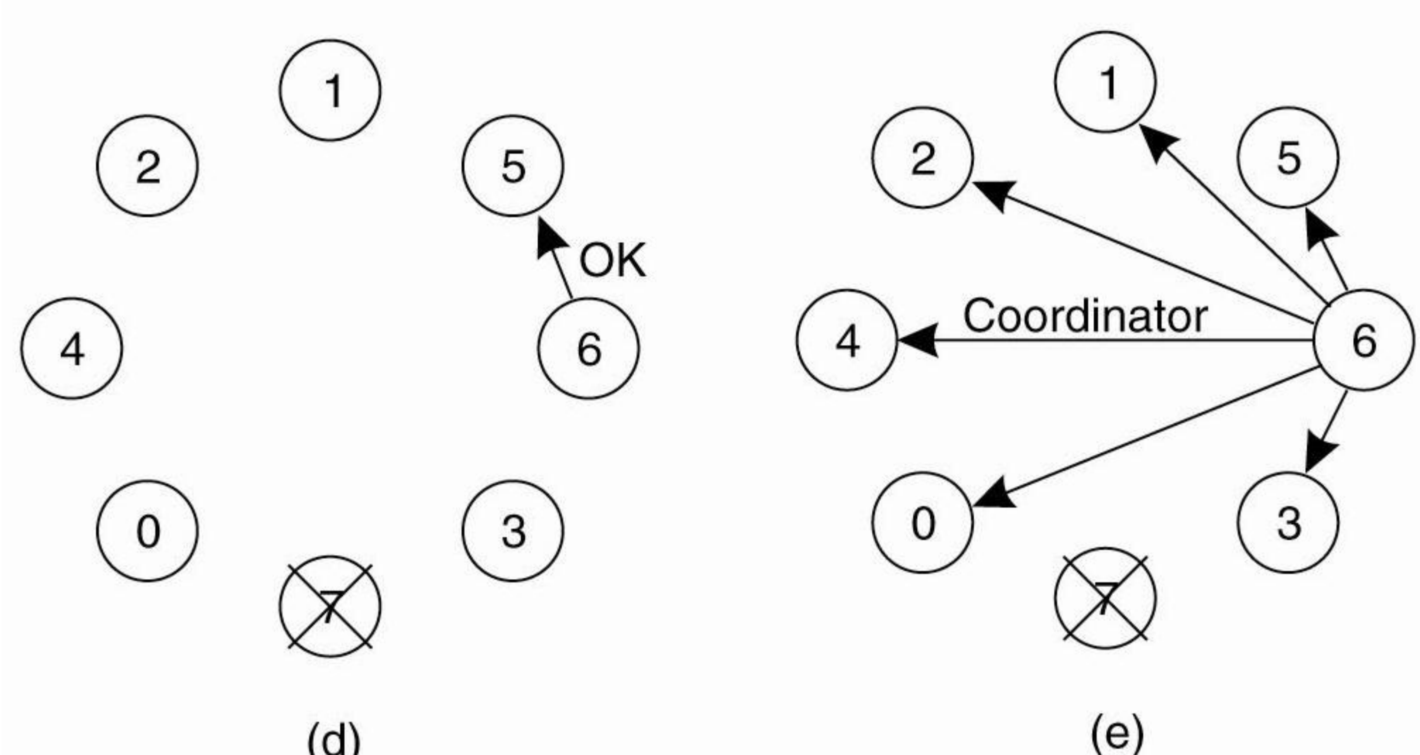
The Bully Algorithm



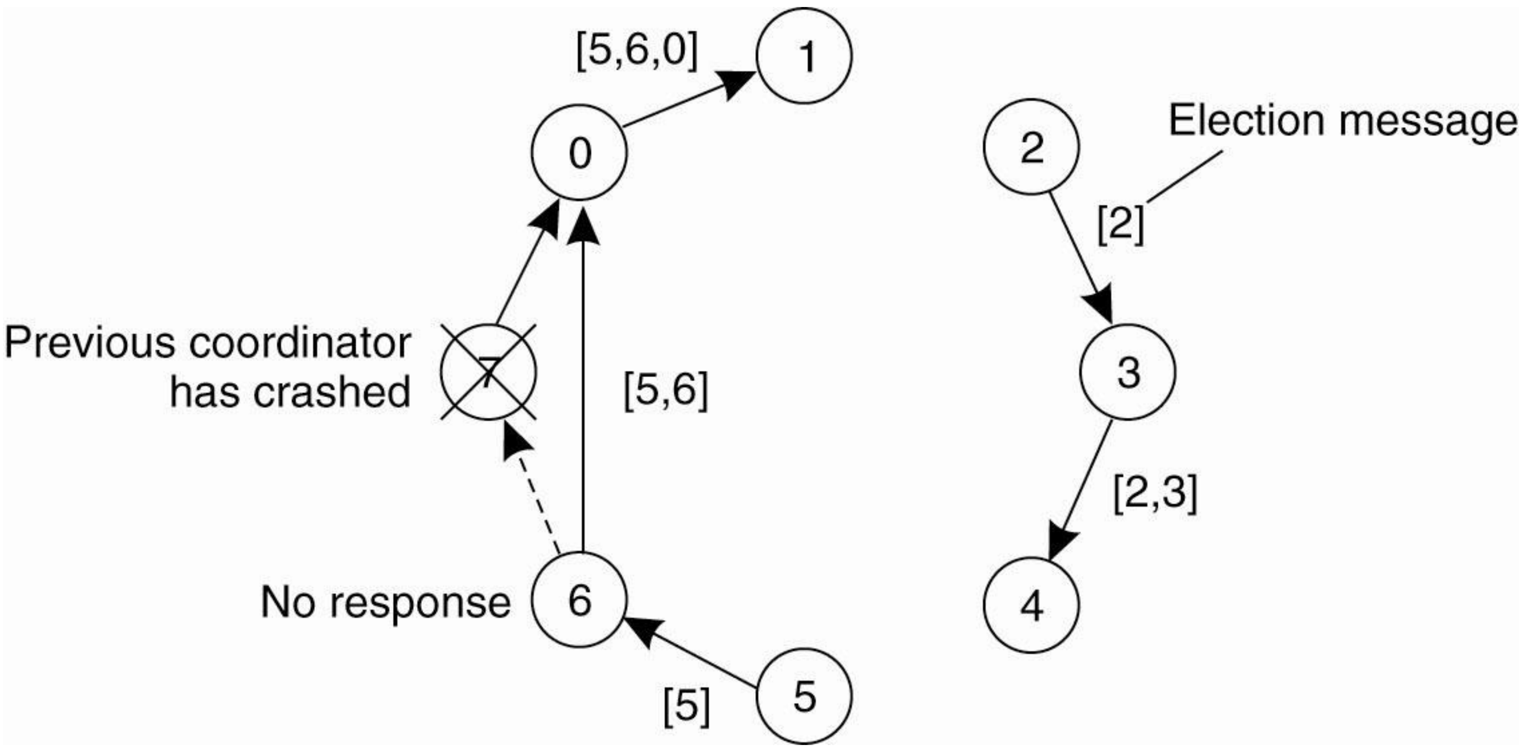
- Process 4 holds election
- Processes 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm

- Process 6 tells 5 to stop
- Process 6 wins and tells everyone

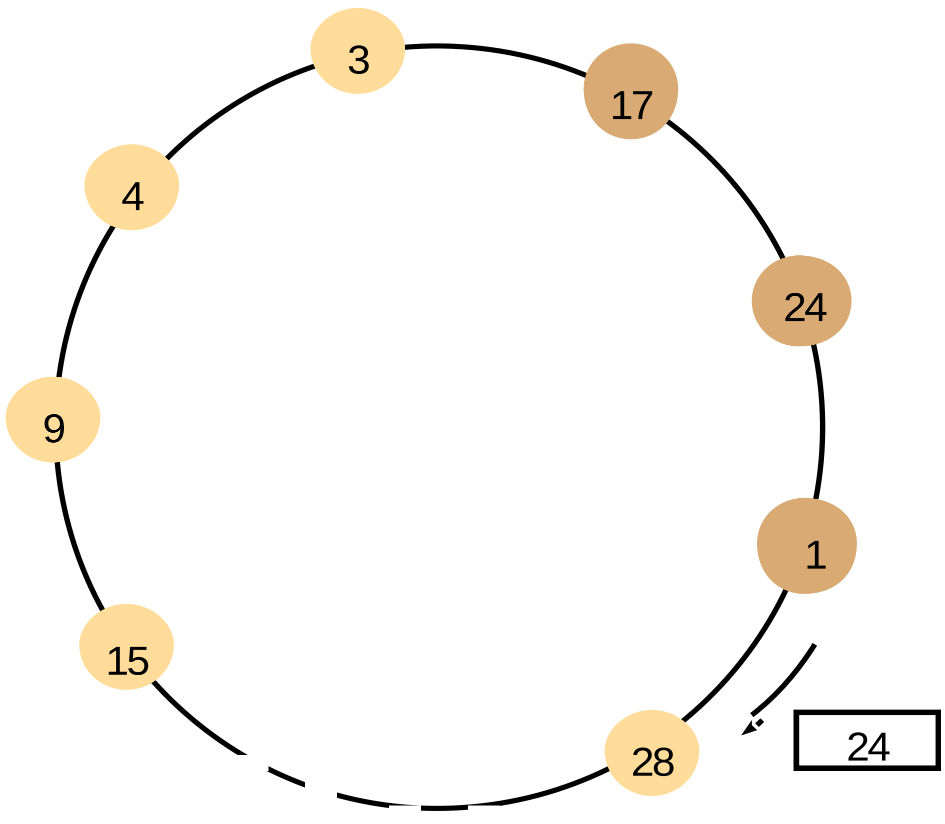


Ring Algorithm



Another

- Election started by process 17



Next Time

Next topic: Replication and Consistency

Read Chapter 7 TVS

Have a great weekend!