Distributed Operating Systems

Coverage

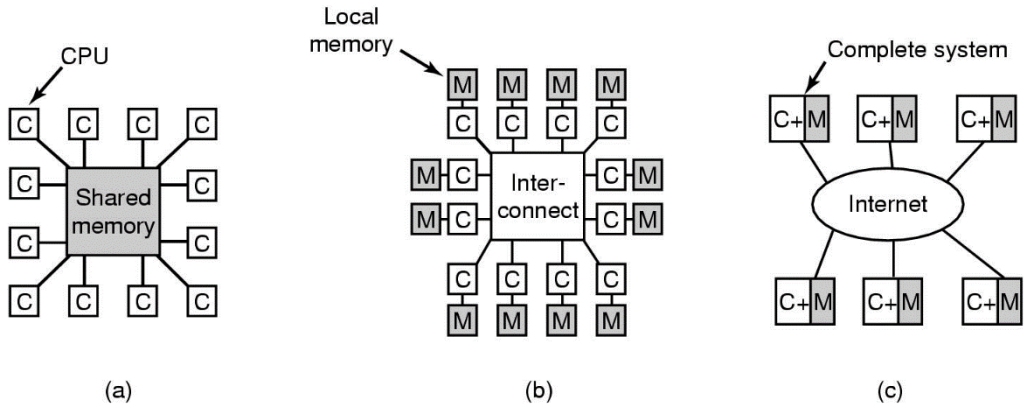
- Distributed Systems (DS) Paradigms
  - DS … NOS, DOS's
  - DS Services: communication, synchronization, coordination, replication…

What is a Distributed System

"A distributed system is the one preventing you from working because of the failure of a machine that you had never heard of"

Leslie Lamport

… collection of autonomous entities appearing to users as a single OS



shared memory multiprocessor

message passing multicomputer

distributed system

→ Multiple computers sharing (same) state and interconnected by a network

Distribution: Example Pro/Cons

The Good Stuff: Resource Sharing (concurrency → performance), Distributed Access (matching spatial distribution of applications), Scalable, Load Balancing (Migration, Relocation), Fault Tolerance.

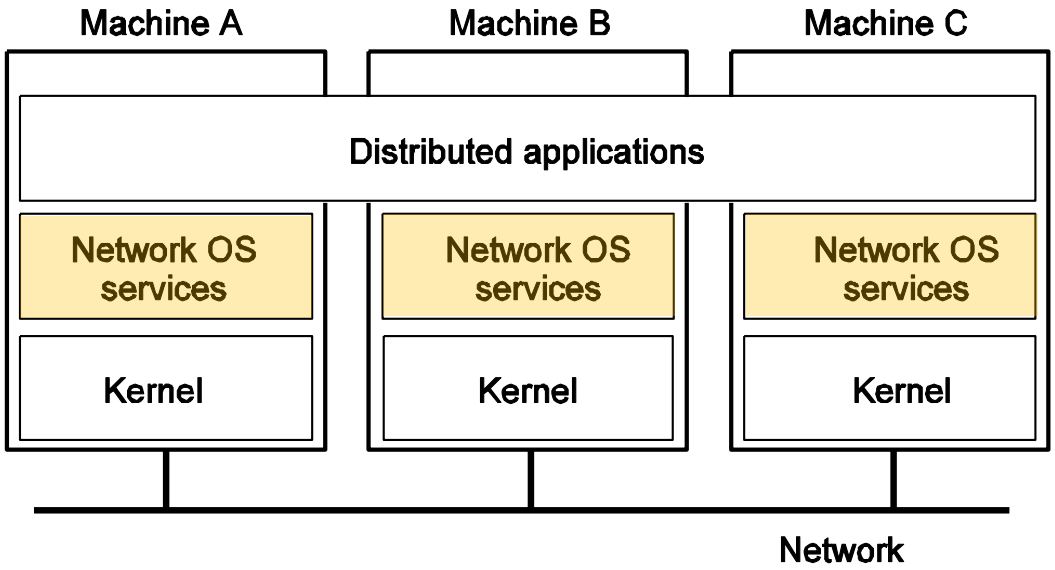
- Bank account database (DB) example
  - Naturally centralized: easy consistency and performance
  - Fragment DB among regions: exploit locality of reference, security & reduce reliance on network for remote access
  - Replicate each fragment for fault tolerance
- But, we now need (additional) DS techniques
  - Route request to right fragment
  - Maintain access/consistency of fragments as a whole database
  - Maintain access/consistency of each fragment's replicas
  - …

OS's for DS's

- Loosely-coupled OS
  - A collection of computers each running their own OS with OS's allowing sharing of resources across machines
  - A.K.A. **Network Operating System (NOS)**
  - Provides local services to remote clients via remote logging
  - Data transfer from remote OS to local OS via FTP (File Transfer Protocols)

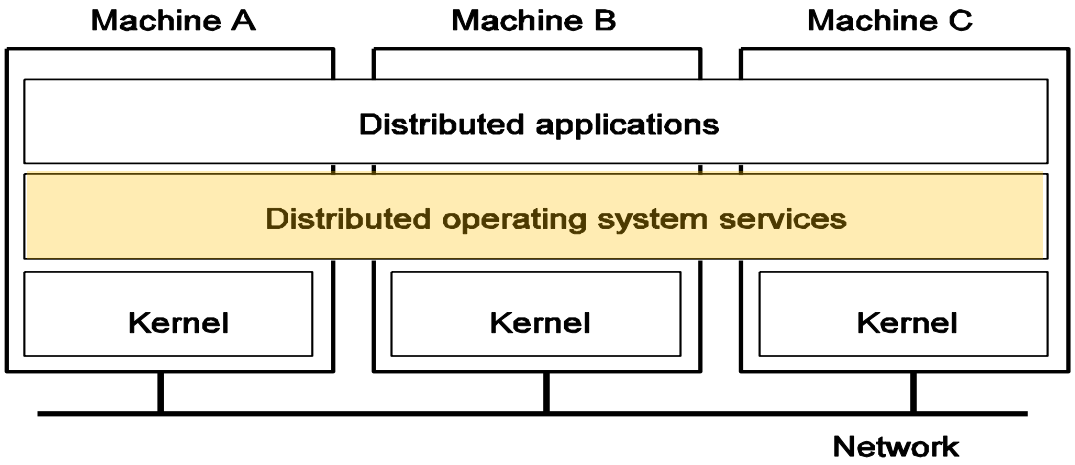
- Tightly-coupled OS
  - OS tries to maintain single global view of resources it manages
  - A.K.A. **Distributed Operating System (DOS)**
  - "Local access feel" as a non-distributed, standalone OS
  - Data migration or computation migration modes (entire process or threads)

Network Operating Systems (NOS)



- Provide an environment where users are (explicitly) aware of the multiplicity of machines.
- Users can access remote resources by
  - logging into the remote machine OR
  - transferring data from the remote machine to their own machine
- Users should know where the required files and directories are and mount them.
- Each machine could act like a server and a client at the same time.
- E.g NFS from Sun Microsystems, etc

Distributed Operating Systems (DOS)



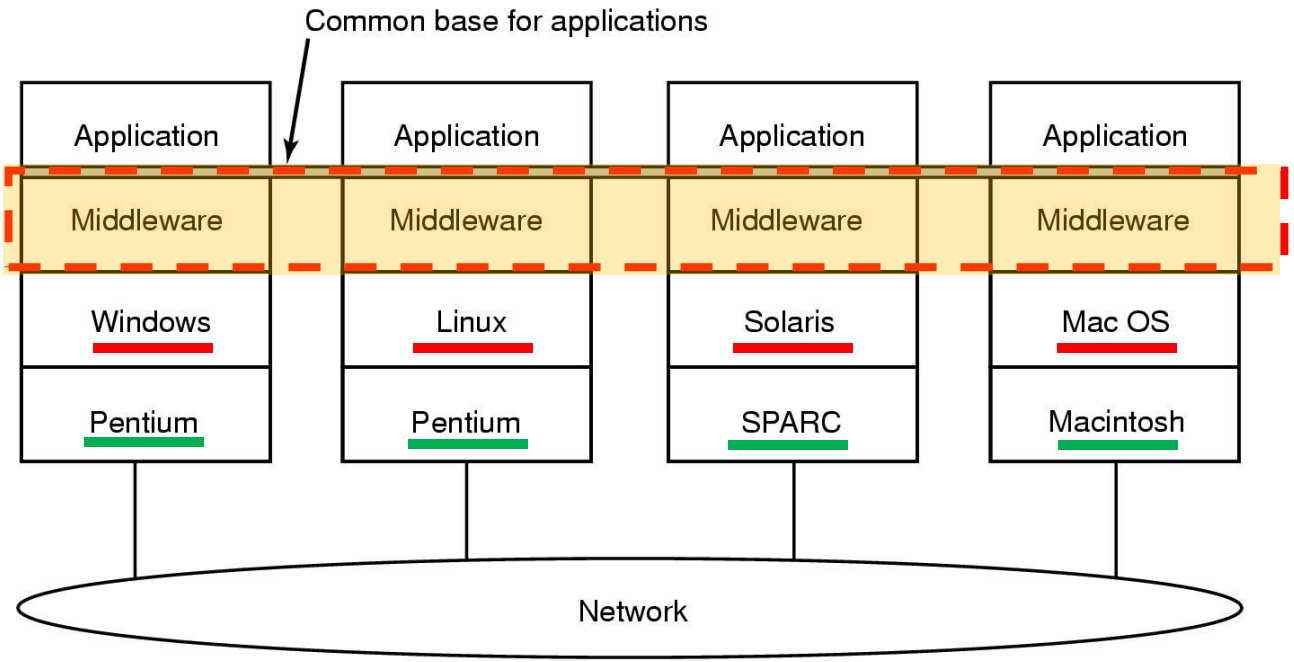
- Runs on a cluster of machines with no shared memory
- Users get the feel of a single processor - virtual uni-processor
- Transparency is the driving force
- Requires
  - A single global IPC mechanism
  - Identical process management and system calls at all nodes
  - Common file system at all nodes
  - State, services and data consistency

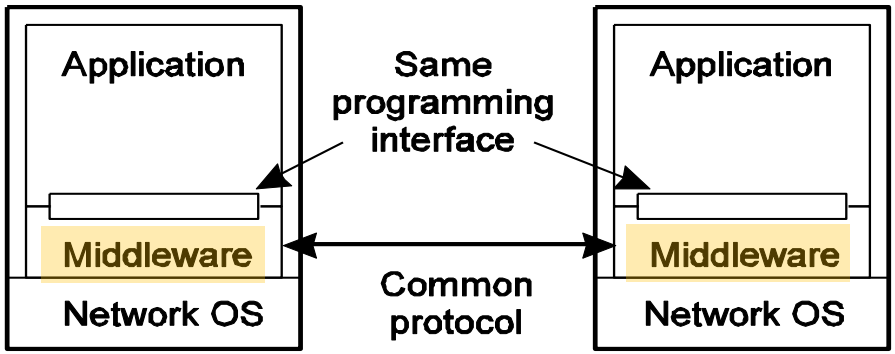
Middleware

- Can we have the best of both worlds?
  - Scalability and openness of a NOS
  - Transparency and common-state of a DOS
- Solution → additional layer of SW above OS (Middleware)
  - Mask heterogeneity
  - Improve distribution transparency (and others)

Middleware → Openness Basis

- Document-based middleware (e.g. WWW)
- Coordination-based MW (e.g., Linda, publish subscribe, Jini etc.)
- File system based MW (upload/download, remote access)
- Shared object based MW
- Data structures (DHTs)





Global Access →Transparency

- Illusion of a single computer across a DS

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk
Fragmentation	Hide whether the resource is fragmented or not

Distribution transparency: All of above + performance + flexibility (modification, enhancements for kernel/devices), balancing/scheduling, & scaling (allowing systems to expand without disrupting users) + …

Reliability, Performance, Scalability

- Faults (Fail stop, transient, Byzantine)
- Fault Avoidance (rejuvenate)
- Fault Tolerance
  - + Redundancy techniques (**k** failures??)
  - + Distributed control
- Fault Detection & Recovery
  - + Atomic transactions
  - + Stateless servers
  - + Acknowledgements and timeout-based retransmissions of messages

- Batch if possible
- Cache whenever possible
- Minimize copying of data
- Minimize network traffic
- Take advantage of fine-grain parallelism for multiprocessing

- Avoid centralized entities
  - Provides no/limited fault tolerance
  - Leads to system bottlenecks
  - Issues of network traffic capabilities with centralized entity
- ➢ Avoid centralized algorithms
- ➢ Perform most operations on client workstations

Design Issues

- Resource management
  - Hard to obtain consistent information about utilization or availability of resources.
  - To be calculated (costly!!) approximately using heuristic methods.
- Processor allocation
  - Load balancing
  - Hierarchical organization of processors.
  - If a processor cannot handle a request, ask the parent for help.
  - …BUT crashing of a higher level processor results in isolation of all processors attached to it.
- Process scheduling
  - Communication dependency, Causality, Linearizability… to consider
- Fault tolerance
  - Consider distribution of control and data.

So what services do we need to realize DS?

- Communication
- Coordination & Synchronization
- Replication
- Failure Handling
- Consistency
- Liveness
- Storage

Communication (Group Comm)

One to many communication (blocking or non-blocking?)

- (atomic-)Multicast/Broadcast
- Open group/Closed group

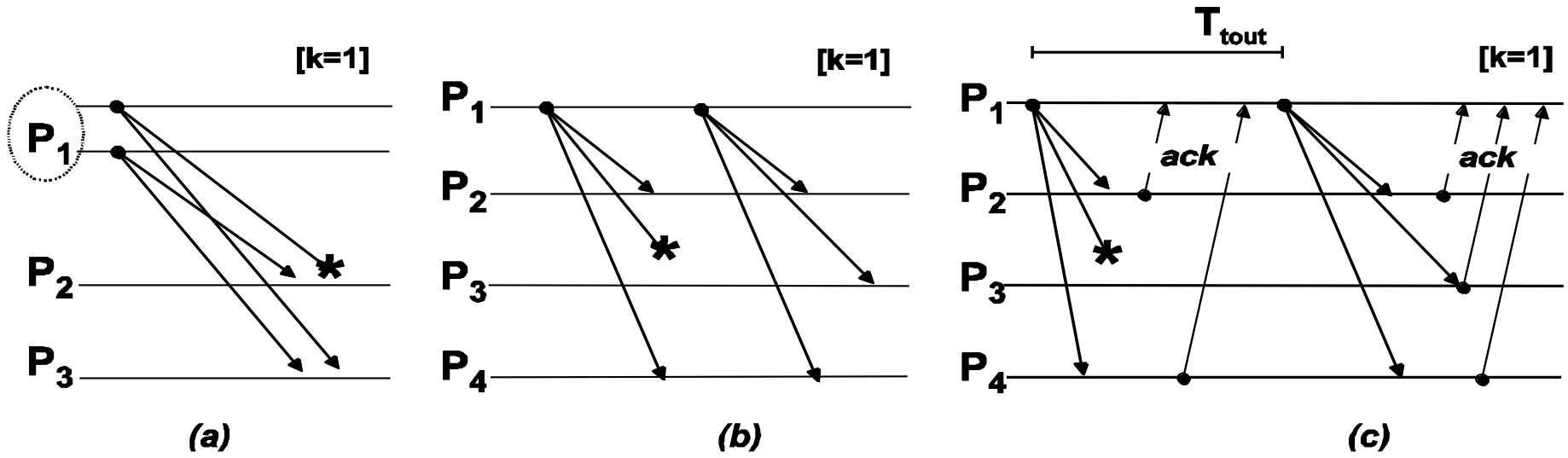
Many to one communication (e.g., WSN)

Many to many Communication

- Absolute Ordering (Global clock)
- Consistent ordering
- Causal ordering

Communication: Reliable Delivery

- Omission failure tolerance (degree k).
- Design choices:
  a) Error masking (spatial): several (> k) links
  b) Error masking (temporal): repeat K+1 times
  c) Error recovery: detect error and recover



Reliable Delivery (cont.)

Error detection and recovery: ACK's and timeouts

- Positive ACK: sent when a message is received
  - Timeout on sender without ACK: sender retransmits
- Negative ACK: sent when a message loss detected
  - Needs sequence #s or time-based reception semantics
- Tradeoffs
  - Positive ACKs faster failure detection usually
  - NACKs : fewer msgs…

Q: what kind of situations are good for

- Spatial error masking?
- Temporal error masking?
- Error detection and recovery with positive ACKs?
- Error detection and recovery with NACKs?

Coverage

- DS Paradigms
  - DS & OS's
  - Services and models
  - Communication

- Coordination
  - Distributed ME
  - Distributed Coordination

Coordination protocols in DOS/DS

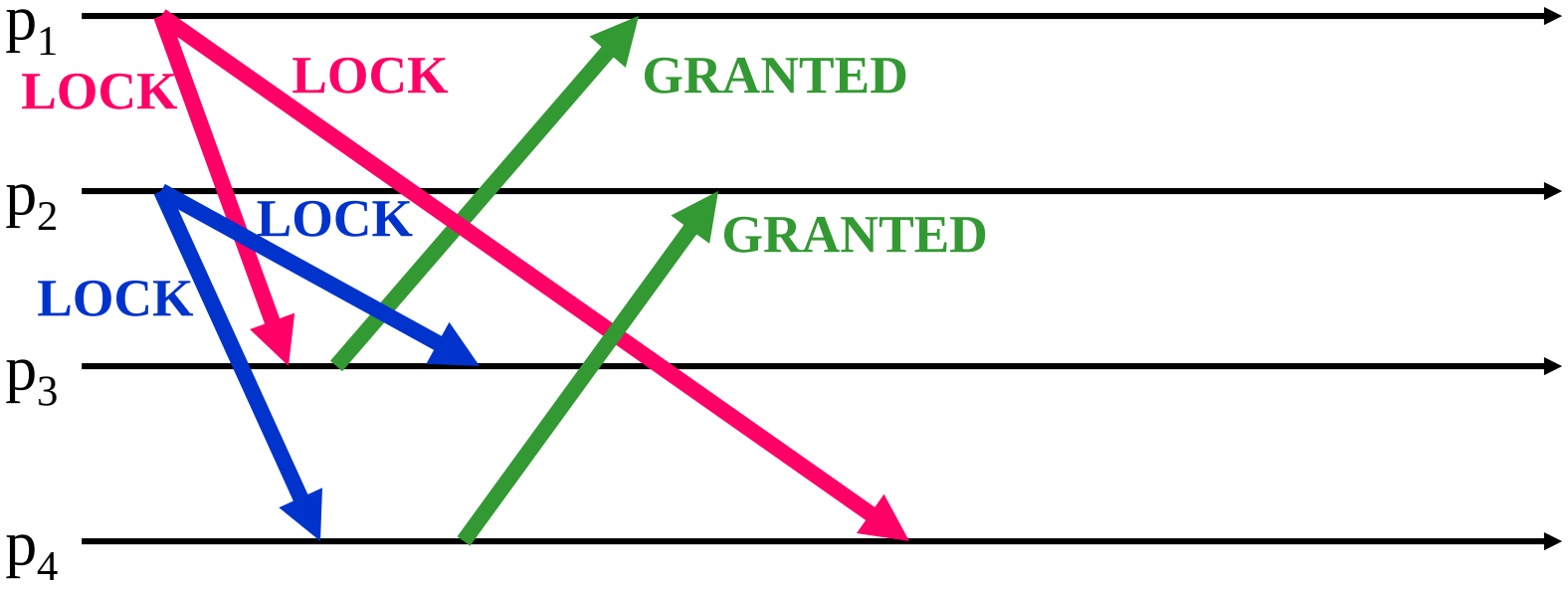
- *Distributed* ME/atomicity
- *Distributed* synchronization & ordering

→How do we coordinate the distributed resources for ME, CS access, consistency etc?

ME

- TSL, Semaphores, monitors… (Single OS)
- Do they work in DS given timing delays, ordering issues +++?

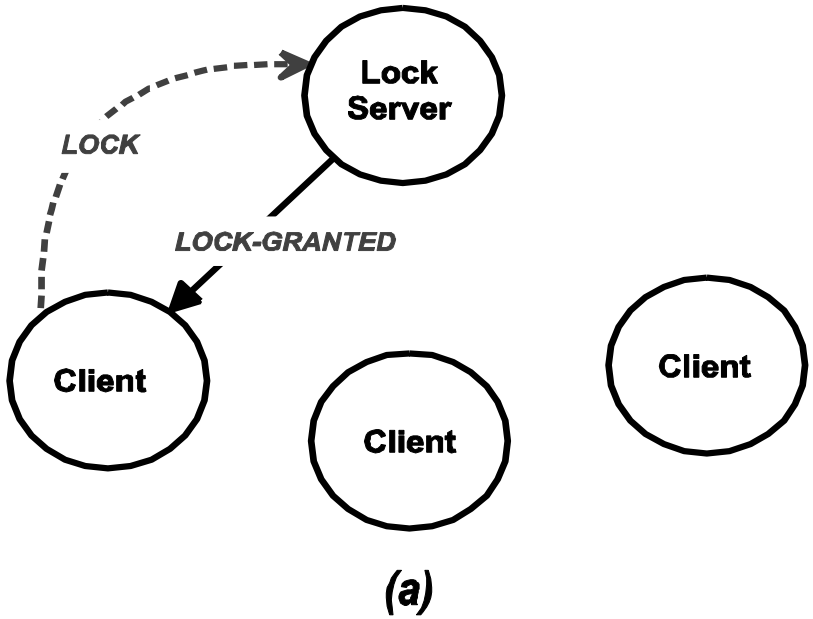
Distributed Lock Problems



What happens? Solutions?

Distributed Mutual Exclusion

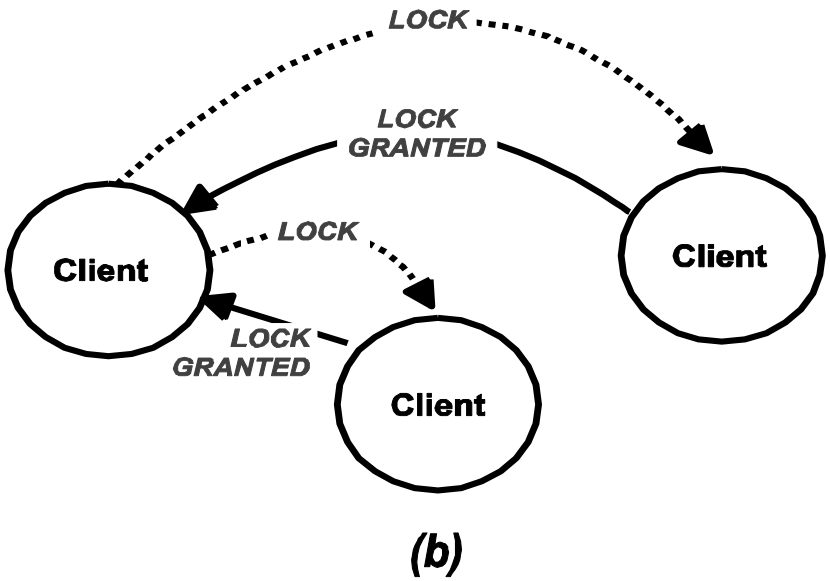
• Solution #1: Build a lock server in a centralized manner (generally simple)



- Lock server solution problems
  - Server is a single point of failure
  - Server is performance bottleneck
  - Failure of client holding lock also causes problems: No unlock sent
    - Similar to garbage collection problems in DSs … validity conditions etc

Distributed Mutual Exclusion (cont.)

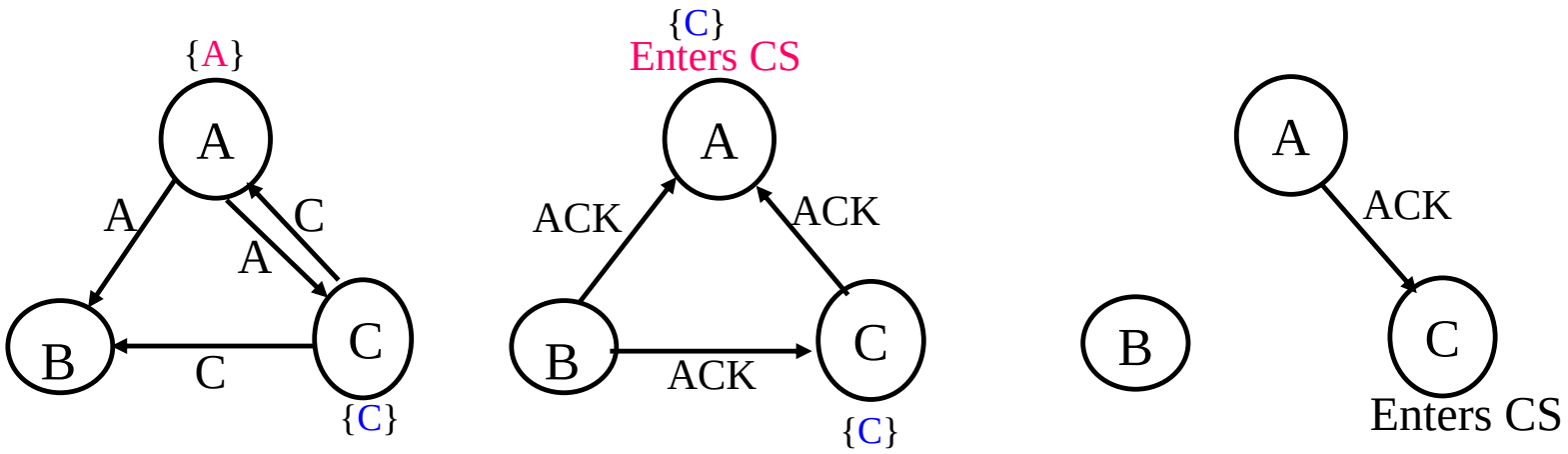
- Solution #2: Decentralized alg
  - Replicate state in central server on all processes
  - Requesting process sends LOCK message to all others
  - Then waits for LOCK_GRANTED from all
  - To release critical region, send UNLOCK to all



(b)

Distributed ME (all ACK)

- To request CS: send REQ msg. M to ALL; enQ M in local Q
- Upon receiving M from P_i
  - if it does not want (or have) CS, send ACK
  - if it has CS, enQ request P_i : M
  - if it wants CS, enQ/ACK based on lowest ID (time-stamp would be so much nicer but lack of time ➔ no time basis for timestamps)
- To release CS: send ACK to all in Q, deQ
- To enter CS: enter CS when ACK received from all



Synchronization

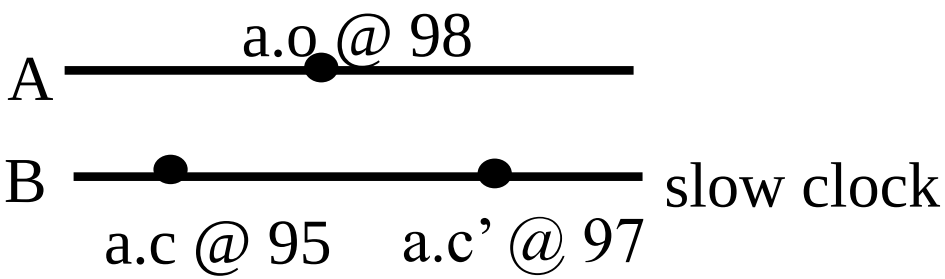
- Blocking (send primitive is blocked until receive acknowledgment)
  - Time out
- Nonblocking (send and copy to buffer)
  - Polling at receive primitive
  - Interrupt

- Synchronous (Send and receive primitives are blocked)
- Asynchronous

Coordination in Distributed Systems

- Event ordering
  - centralized system: ~ easy (common clock & memory)
  - distributed system: hard (convergent/consistent dist. time)

- Example: Unix “make” program
  - source files/object files ➔ make [compiles & links based on last version]
  - a.o @99 & a.c @100 ➔ re-compile a.c [assuming a common time base]
  - “make” in a DS?

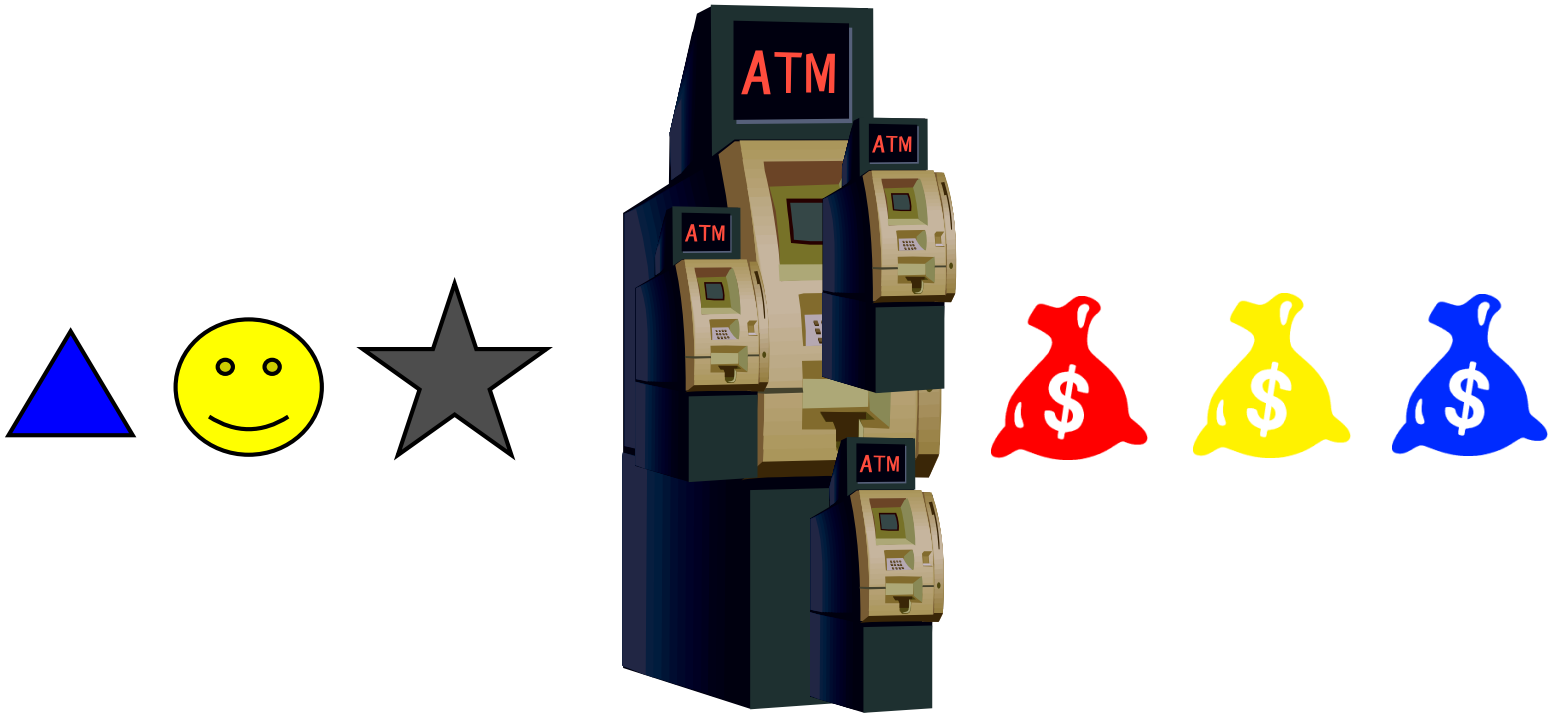


Coordination in Distributed Systems

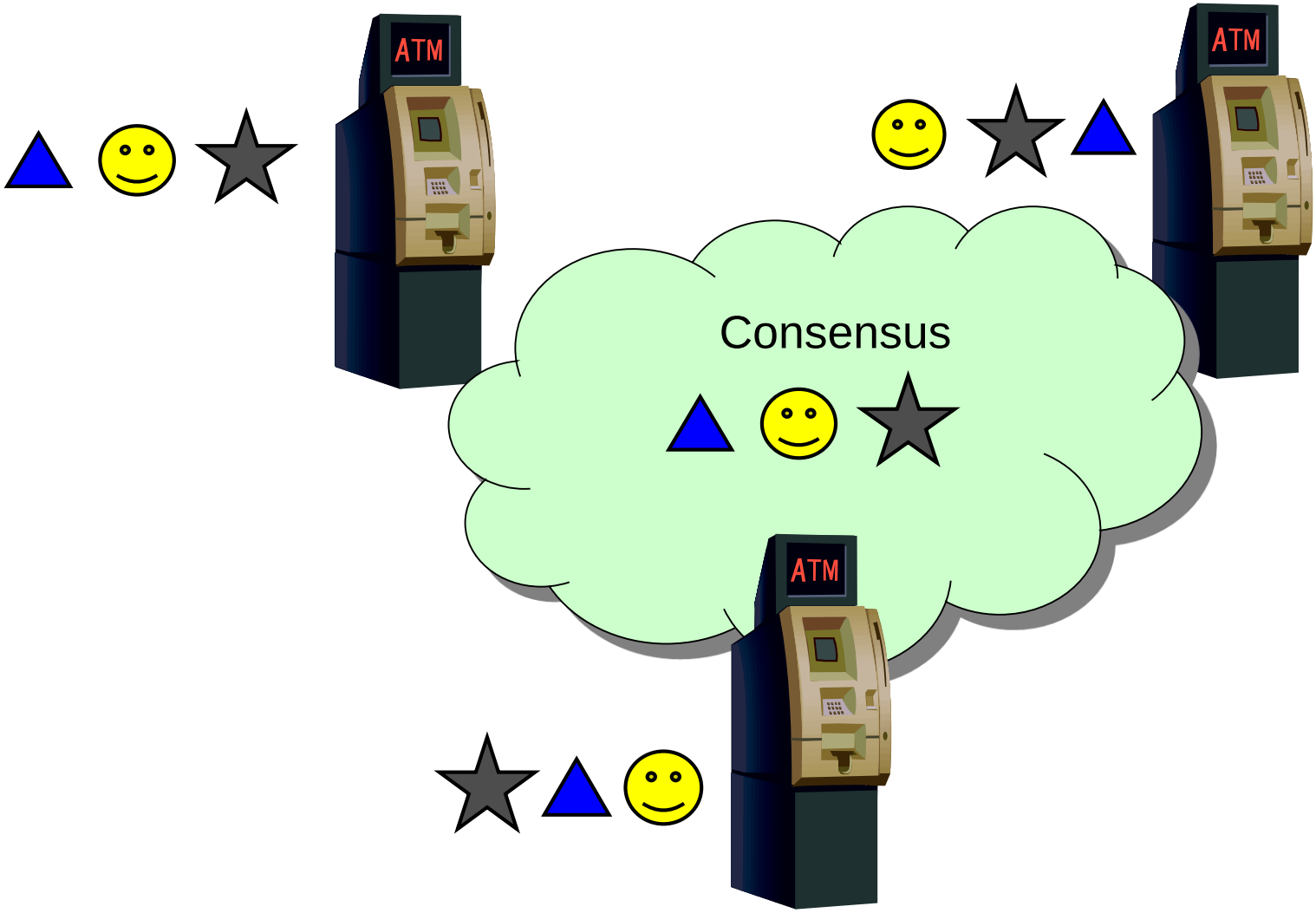
1. Given distributed entities, how does one obtain resource "**coordination**" to result in an agreed action (such as CS access/ME, shared memory "writes", producer/consumer modes or decisions)?
2. How are distributed tasks/requests "**ordered**"?
3. Given distributed resources, how do they "all" **agree** on a "single" course of action?

- Asynchronous Coordination
  -2PC, leader elections, etc…
- Synchronous Coordination
  - clocks, ordering, serialization, etc …

Consistency & Distributed State Machines



Distributed State Machine



Asynch: “Single” Decision - Commit

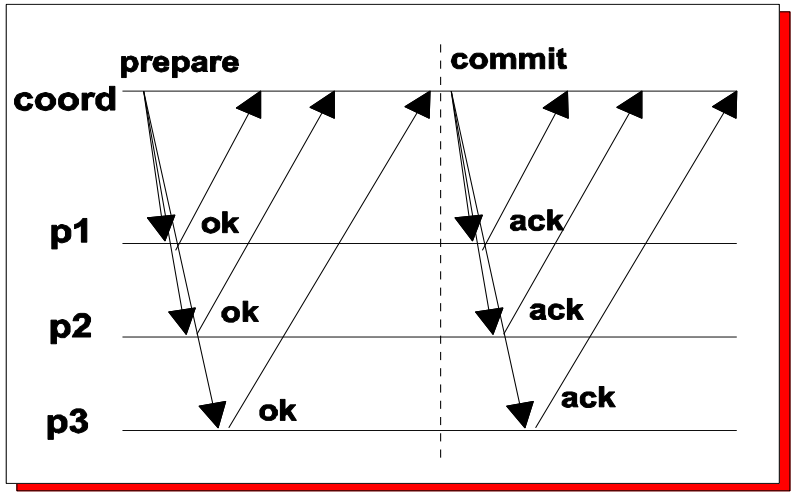
2PC: Two Phase Commit Protocols

- coordinator (pre-specified or selected dynamically)
- multiple secondary sites (“cohorts”)

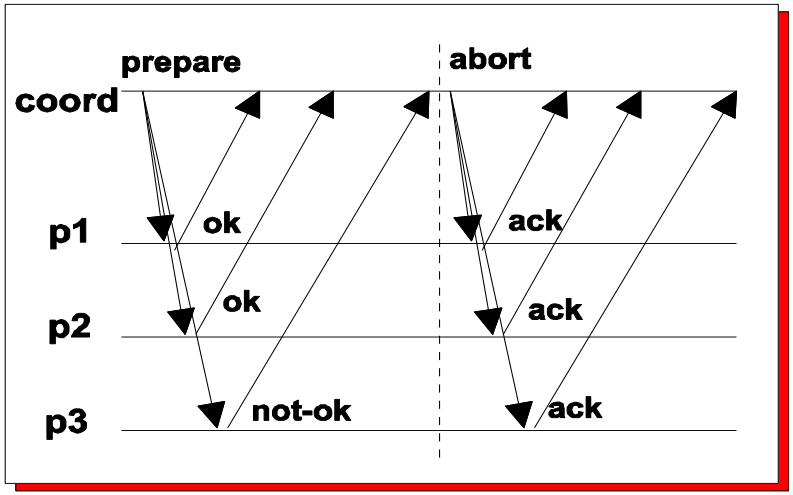
Objective: All nodes agree and execute a single decision

[all agree or no action taken...banking transactions]

Two – Phase Commit (2PC) Protocol



(a)



(b)

Coord. Actions

1. send PREPARE to **all**

.....”**bounded waiting**”.....

---

4. receive OK from **all**
   put COMMIT in log &
   send COMMIT to **all**

4’. receive ABORT
   → send ABORT to **all**

5. ACK from **all**? DONE

2. get msg (PREPARE)

3. if ready, send OK
(write undo/redo logs)
else, send NOT-OK

---

4 → receive COMMIT
release resources,
send ACK

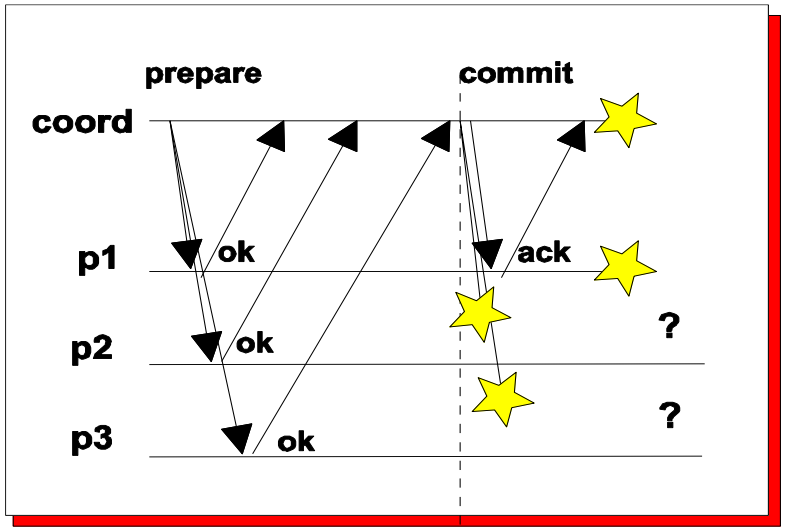
4’ → receive ABORT, undo actions

Each Client Actions

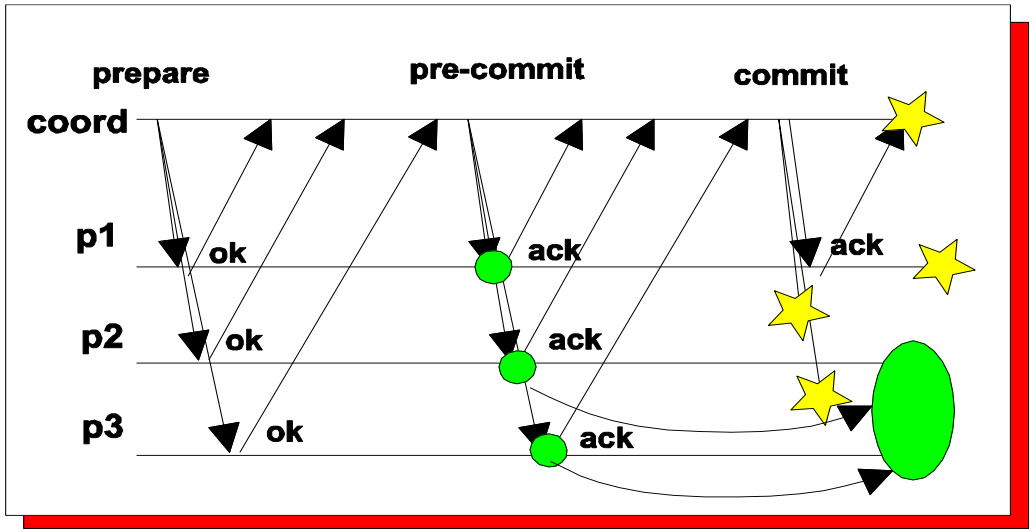
Two-phase commit (cont.)

Problem: coordinator failure, after PREPARE & before COMMIT, blocks participants waiting for decision (a)

- **Three-phase commit** overcomes this (b) … slowwwwww
  - delay final decision until enough processes "know" which decision will be taken



(a)



(b)