Cloud Computing

Parallel Computing and Distributed Systems

Dell Zhang

Birkbeck, University of London

2018/19

The Path to Cloud Computing

- Cloud computing is based on the ideas and experiences accumulated in many years of research on ***parallel** computing* and ***distributed** systems*.
  - Cloud applications are based on the client-server paradigm with a relatively simple software, a thin-client, running on the user's machine, while the computations are carried out on the cloud [*usually by many machines in parallel*] .

Parallel Computing

- Parallel hardware and software systems allow us to:
  - solve problems demanding resources not available on a single system,
  - reduce the time required to obtain a solution.

Speed-up

- The *speed-up* S measures the effectiveness of parallelization:

$$S(N) = T(1) \,/\, T(N)$$

  - $T(1)$: the execution time of the sequential computation
  - $T(N)$: the execution time when N parallel computations are carried out

Speed-up

- When the problem/dataset size is fixed, …
- **Amdahl's Law:**
  - If α is the fraction of running time a sequential program spends on non-parallelizable segments of the computation then:

$$S(N) = 1 / [\alpha + (1-\alpha)/N] < 1 / \alpha$$

  - The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

To prove this result call σ the sequential time and π the parallel time and start from the definitions of $T(1), T(N)$, and α:

$$T(1) = \sigma + \pi, \qquad T(N) = \sigma + \frac{\pi}{N}, \qquad \text{and} \qquad \alpha = \frac{\sigma}{\sigma + \pi}. \tag{3}$$

Then

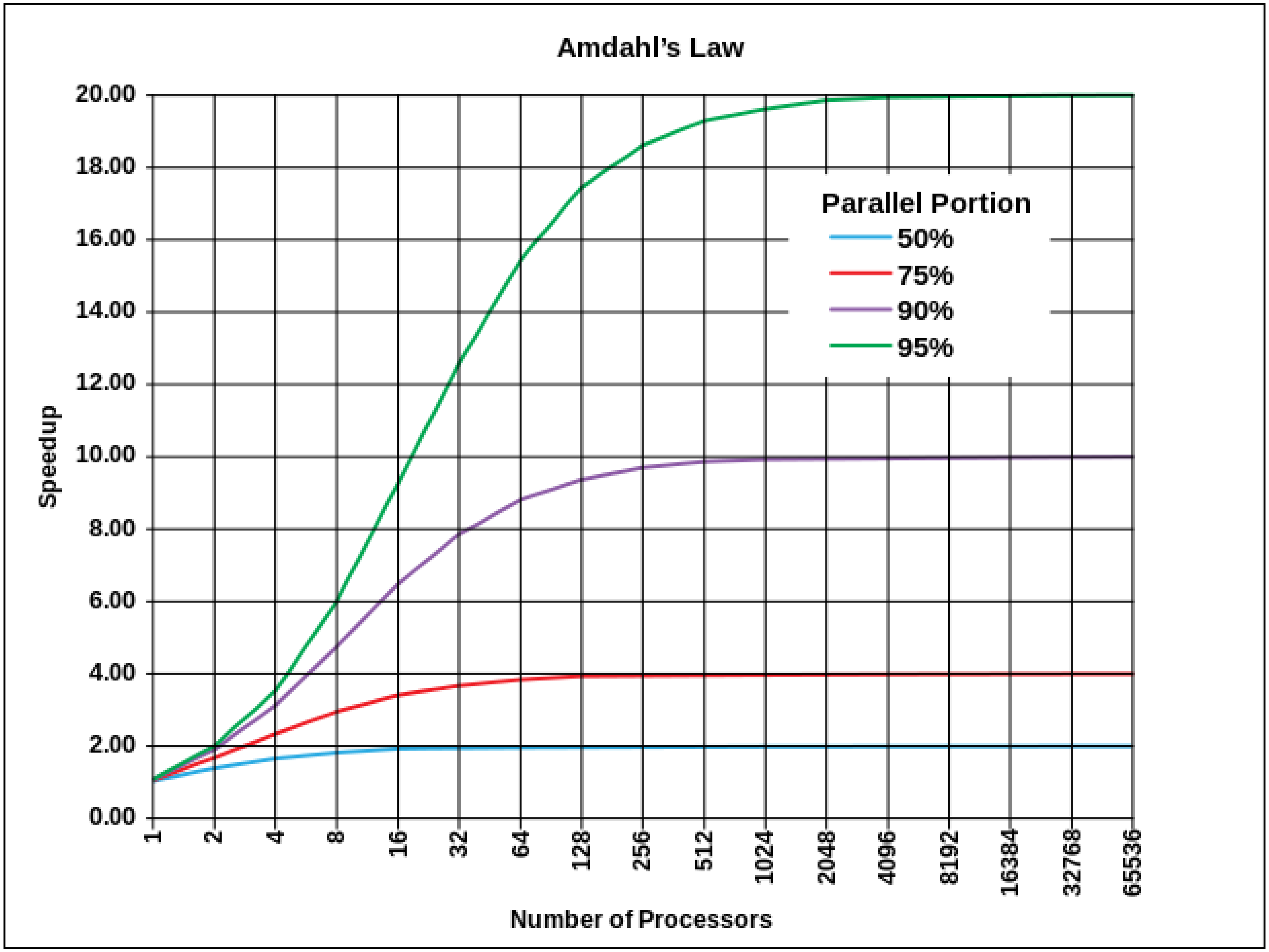
$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \pi/N} = \frac{1 + \pi/\sigma}{1 + (\pi/\sigma) \times (1/N)}. \tag{4}$$

But

$$\pi/\sigma = \frac{1 - \alpha}{\alpha} \tag{5}$$

Thus, for large N

$$S = \frac{1 + (1 - \alpha)/\alpha}{1 + (1 - \alpha)/(N\alpha)} = \frac{1}{\alpha + (1 - \alpha)/N} \approx \frac{1}{\alpha} \tag{6}$$



Amdahl's Law
Speedup
0.00
2.00
4.00
6.00
8.00
10.00
12.00
14.00
16.00
18.00
20.00
Number of Processors
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Parallel Portion
50%
75%
90%
95%

Speed-up

- Relation to *law of diminishing returns*
  - Consider what sort of return you get by adding more processors to a machine, if you are running a fixed-size computation that will use all available processors to their capacity.
  - Each new processor you add to the system will add less usable power than the previous one, as the speed-up heads towards the limit.

Speed-up

- *Embarrassingly* Parallel Problems
  - Little or no effort is required to separate the problem into a number of parallel tasks, i.e., $\alpha \approx 0$
  - Often the case where there exists no dependency (or communication) between those parallel tasks

Speed-up

- When the computing time available is fixed, …
  - The problem/dataset size is allowed to change, i.e., can be arbitrarily large.
- **Gustafson's Law**
  - The amount of work assigned to each process keeps the same
  - The *scaled speed-up* with N parallel processes:

$$S(N) = \alpha + N\,(1-\alpha) = N - \alpha\,(N-1)$$

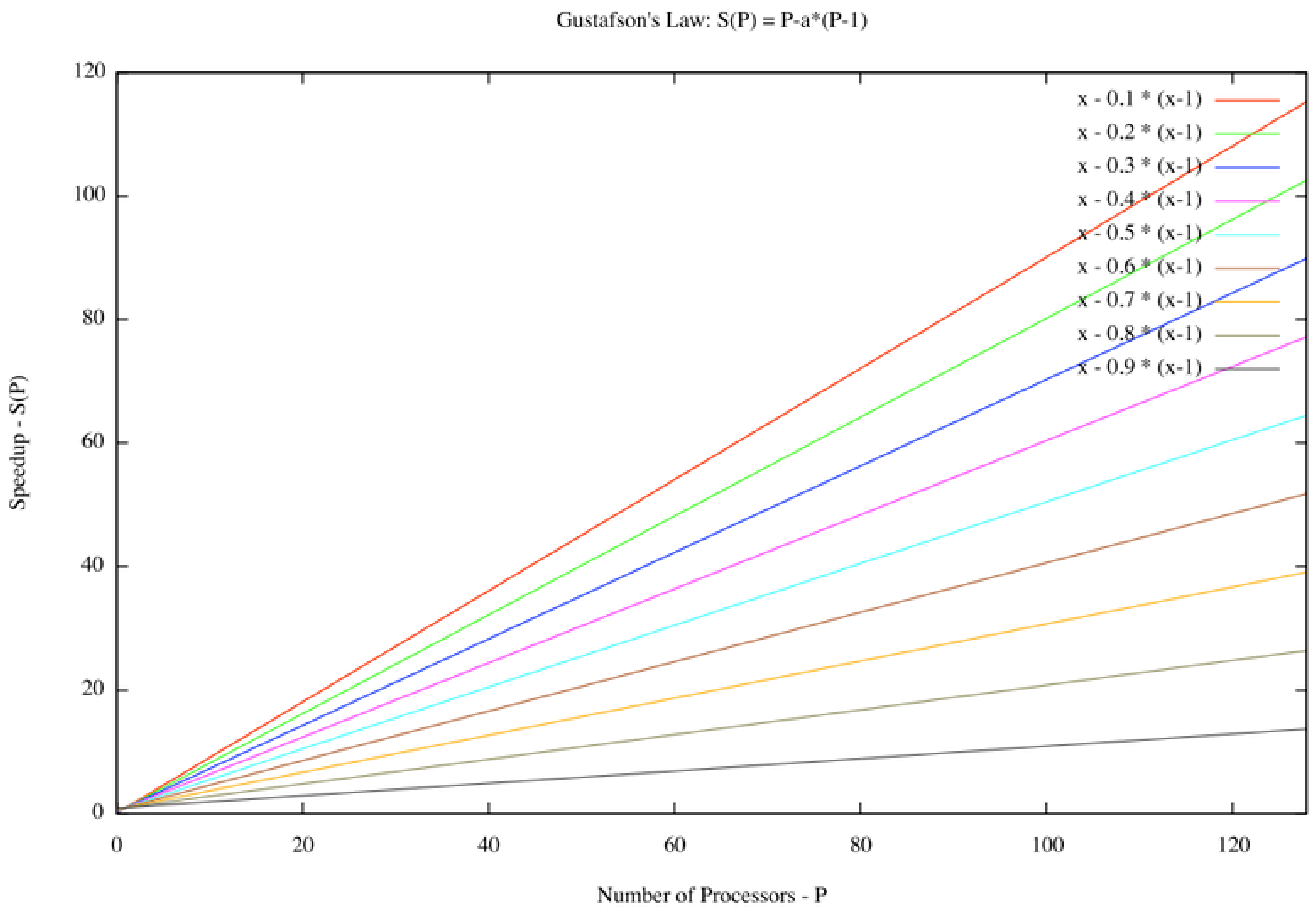
The limitations of the sequential part of a code can be balanced by increasing the problem size.

As before, we call σ the sequential time; now π is the *fixed parallel time per process*; α is given by Equation 3. The sequential execution time, $T(1)$, and the parallel execution time with N parallel processes, $T(N)$, are

$$T(1) = \sigma + N\pi \quad \text{and} \quad T(N) = \sigma + \pi. \tag{8}$$

Then the scaled speed-up is

$$S(N) = \frac{T(1)}{T(N)} = \frac{\sigma + N\pi}{\sigma + \pi} = \frac{\sigma}{\sigma + \pi} + \frac{N\pi}{\sigma + \pi} = \alpha + N(1 - \alpha) = N - \alpha(N - 1). \tag{9}$$


Gustafson's Law: S(P) = P-a*(P-1)
Speedup - S(P)
Number of Processors - P
0
20
40
60
80
100
120
x - 0.1 * (x-1)
x - 0.2 * (x-1)
x - 0.3 * (x-1)
x - 0.4 * (x-1)
x - 0.5 * (x-1)
x - 0.6 * (x-1)
x - 0.7 * (x-1)
x - 0.8 * (x-1)
x - 0.9 * (x-1)

Speed-up

- Impacts
  - Amdahl’s law argues that
    more computing power will make an analysis of the same dataset ***faster***.
  - Gustafson’s law argues that
    more computing power will make ***bigger*** datasets be analysed or ***deeper*** analysis be performed.

Parallelism

- Data parallelism: the data is partitioned into several blocks and the blocks are processed in parallel
  – Single Program Multiple Data (SPMD): multiple copies of the same program run concurrently, each one on a different data block

Divide the Data

Parallelism

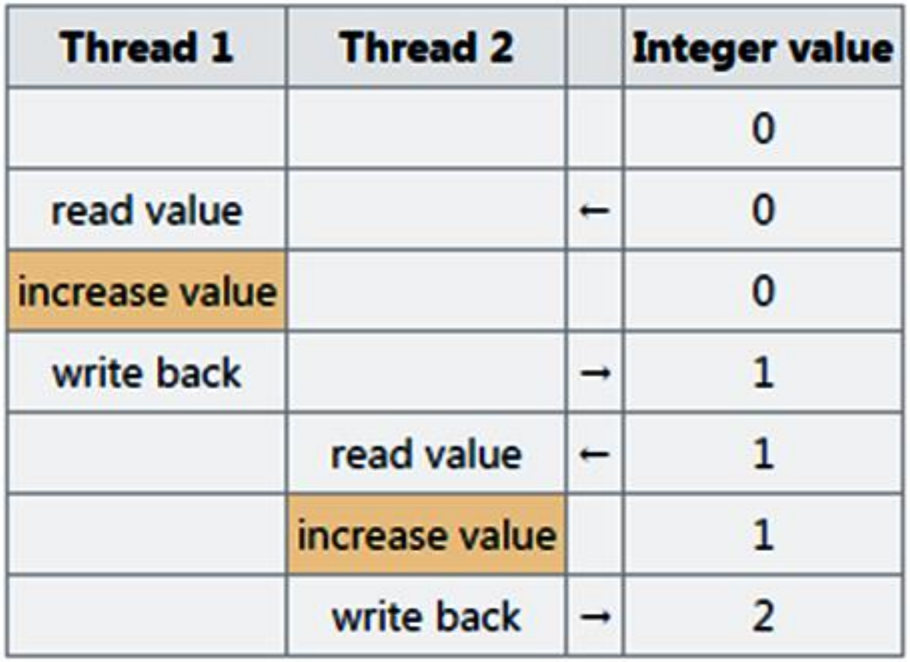
- Task parallelism: the problem is decomposed into tasks that can be carried out concurrently

Divide the Task

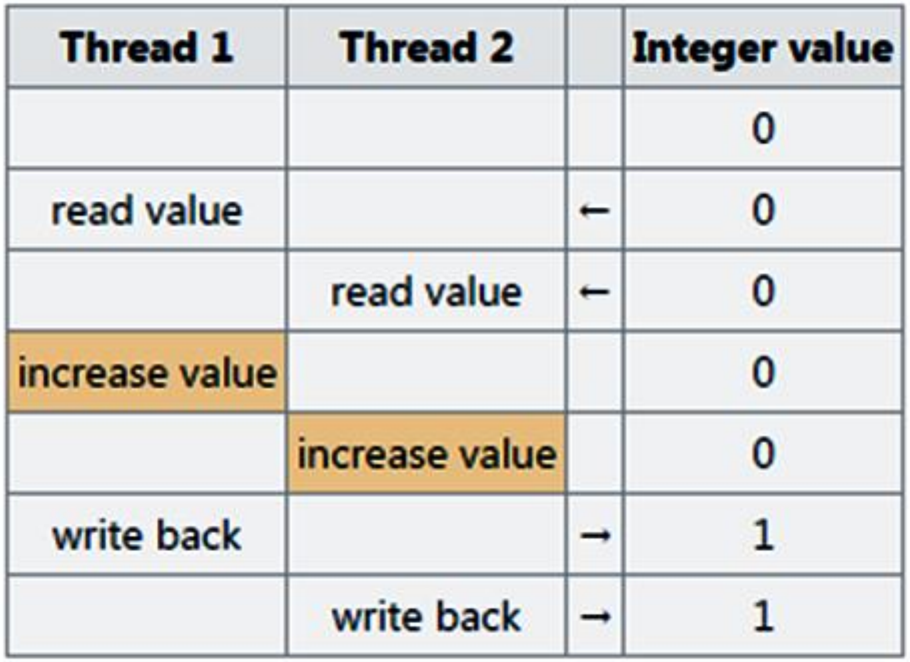
Concurrency

- Concurrency is important
  - Many cloud applications are data-intensive and use a number of instances which run concurrently
- Concurrency is challenging
  - It could lead to <u>race conditions</u>, an undesirable effect when the results of concurrent execution depend on the sequence or timing of events
  - Shared resources must be protected by <u>locks / semaphores / monitors</u> to ensure serial access

Race Conditions



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Deadlock

- A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does
  - “chicken or egg”
  - “Catch-22”

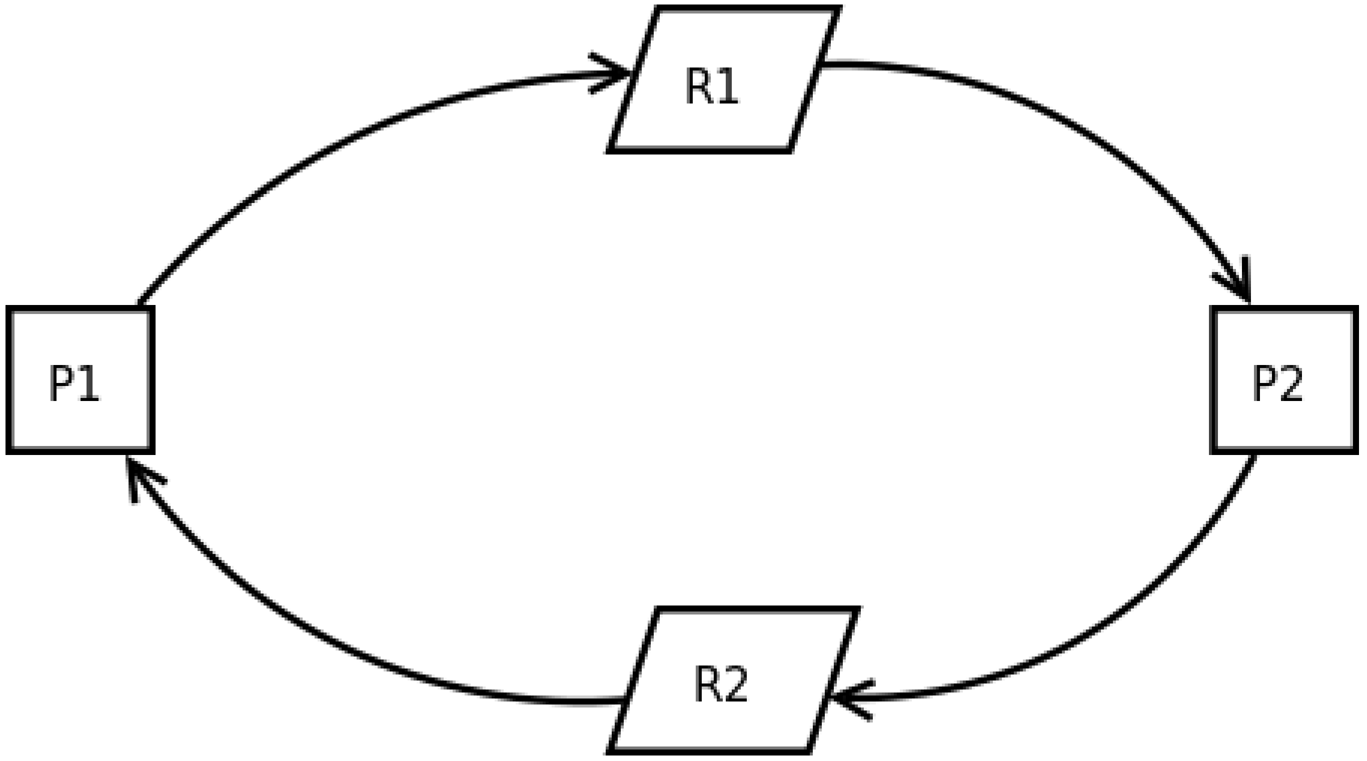
Coffman Conditions

- (1) Mutual exclusion
  - At least one resource must be non-sharable, only one process/thread may use the resource at any given time.
- (2) Hold and wait
  - At least one processes/thread must hold one or more resources and wait for others.

Coffman Conditions

- (3) No preemption
  - The scheduler or monitor should not be able to force a process/thread holding a resource to relinquish it.
- (4) Circular wait
  - Given the set of n processes/threads $\{P_1, P_2, P_3, \ldots, P_n\}$.
    P_1 waits for a resource held by P_2,
    P_2 waits for a resource held by P_3, and so on,
    P_n waits for a resource held by P_1.

Deadlock



Livelock

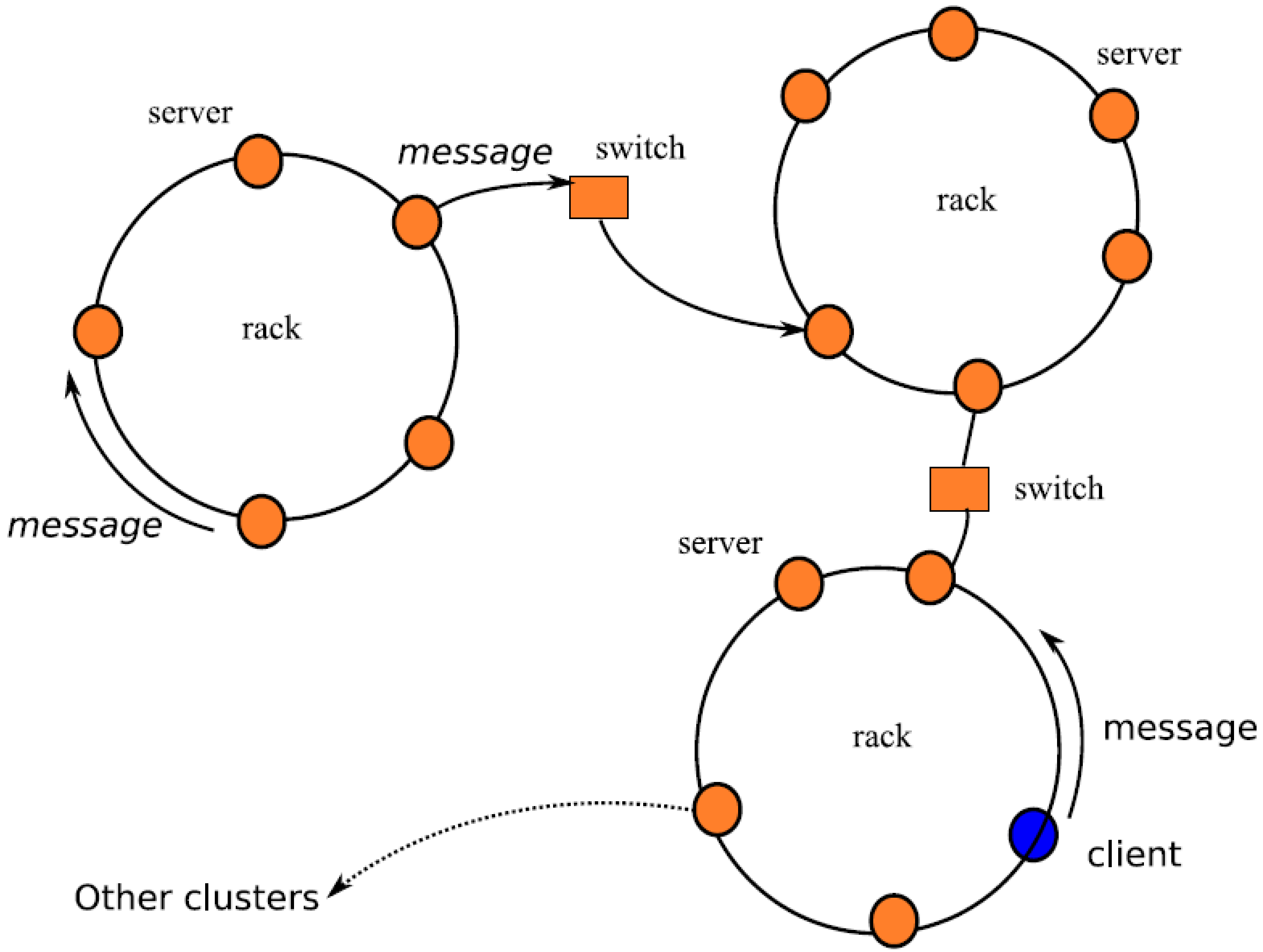
- A *livelock* is similar to a deadlock, except that the states of the processes involved constantly change with regard to one another, none progressing.

Distributed Systems

- A distributed system is an application that *coordinates* the actions of several computers to achieve a specific task
  - By exchanging messages which are pieces of data that convey some information
    - “shared-nothing” architecture: no shared memory/disk
  - Client (nodes) and Server (nodes) are communicating software components
    - we assimilate them with the machines they run on

Distributed Systems

- The system relies on a network that connects the computers and handles the routing of messages
  - Local Area Networks (LAN)
    - 3 communication levels: “racks”, clusters, and groups of clusters
  - Peer to Peer (P2P) Networks

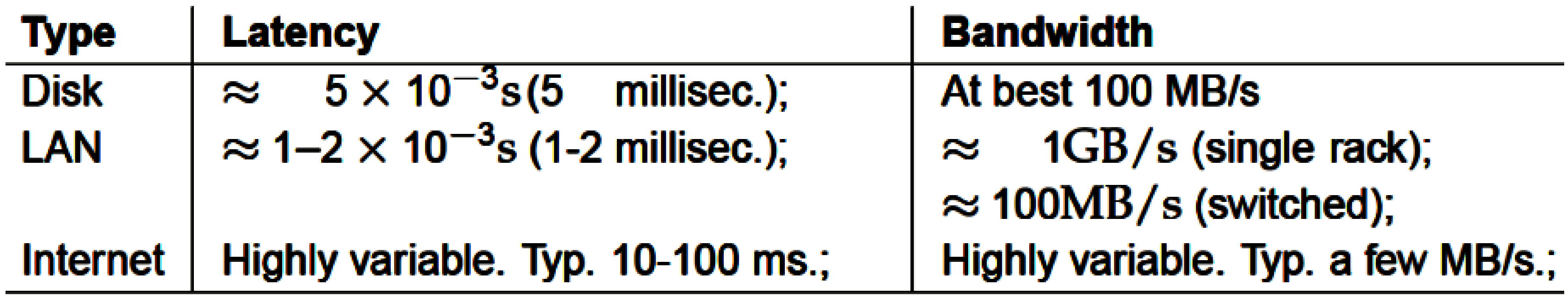

server
message
switch
server
rack
rack
message
switch
server
rack
message
client
Other clusters

Distributed Systems

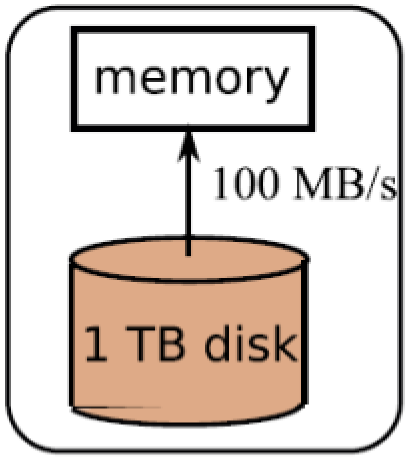
- Typical setting of a Google data centre
  - ≈ 40 servers per rack;
  - ≈ 150 racks per data centre (cluster);
  - ≈ 6,000 servers per data centre;
  - How many data centres? Google's secret, and constantly evolving . . .
  - Rough estimate: 150-200 data centres? 1,000,000 servers?

Distributed Systems

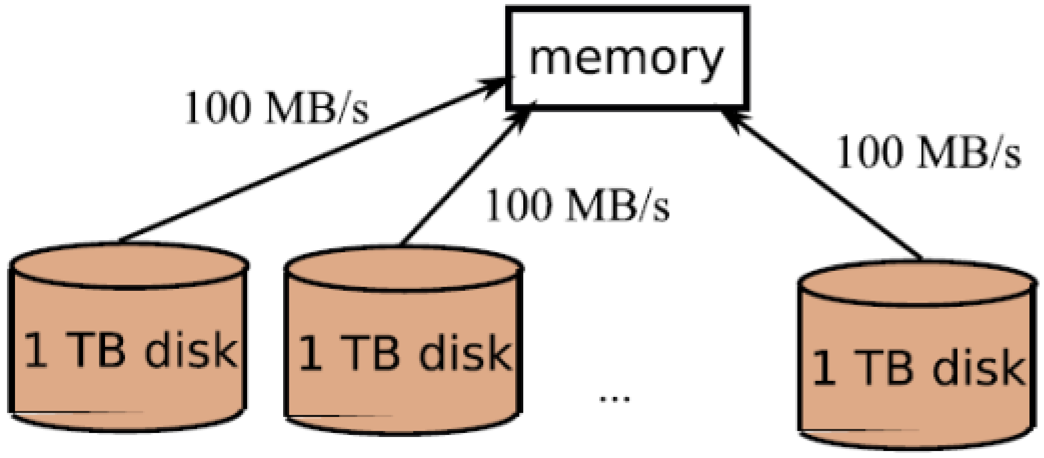
- Performance



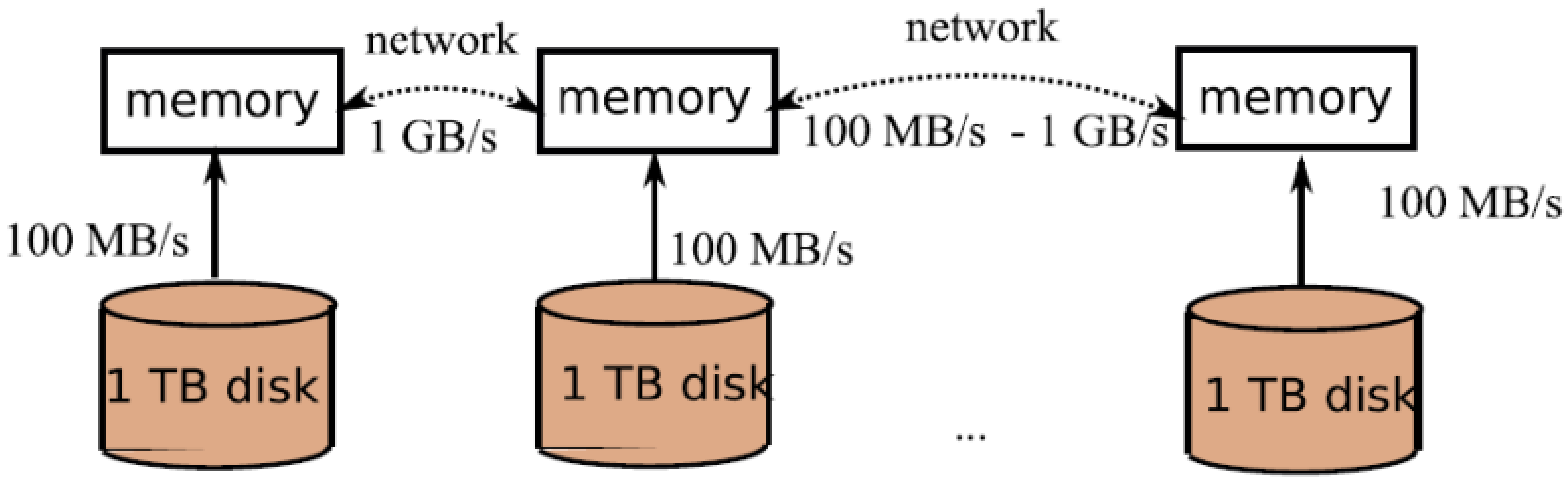
Type	Latency	Bandwidth
Disk	$\approx 5 \times 10^{-3}$s (5 millisec.);	At best 100 MB/s
LAN	$\approx$ 1–2 $\times 10^{-3}$s (1-2 millisec.);	$\approx 1\mathrm{GB/s}$ (single rack);
		$\approx 100\mathrm{MB/s}$ (switched);
Internet	Highly variable. Typ. 10-100 ms.;	Highly variable. Typ. a few MB/s.;



a. Single CPU, single disk



b. Parallel read: single CPU, many disks



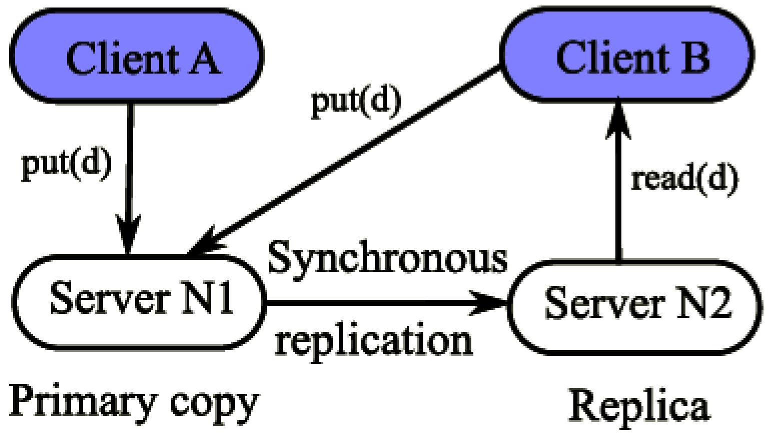
c. Distributed reads: an extendible set of servers

Distributed Systems

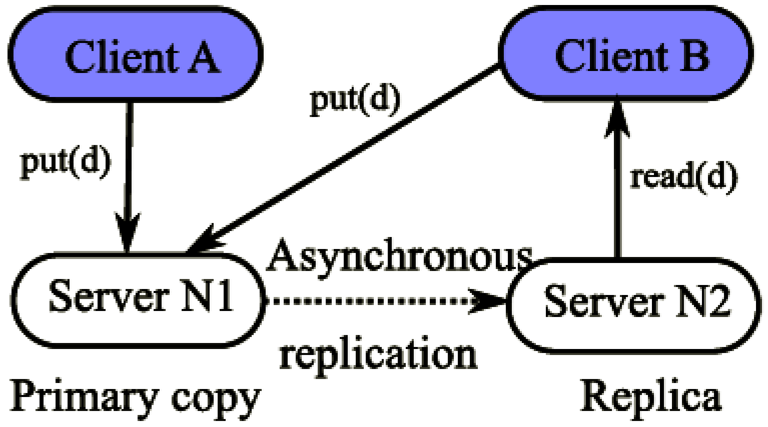
- For **data-centric** application
  - *disk transfer rate is a bottleneck* for large scale data management: parallelization and distribution of the data on many machines is a means to eliminate this bottleneck
  - *write once, read many*: distributed storage is appropriate for large files that are written once and then repeatedly scanned
  - *data locality*: bandwidth is a scarce resource, so program should be "pushed" near the data they must access to

Replication and Consistency

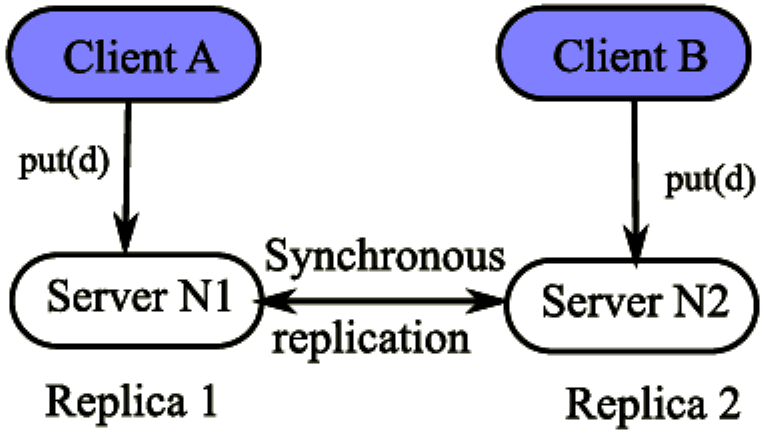
- Distribution also gives an opportunity to reinforce the security of data with *replication*
  - Replication: a mechanism that copies data item located on a machine *A* to a remote machine *B*
  - Consistency: ability of a system to behave as if the transaction of each user always run in isolation from other transactions, and never fails
    - Difficult in centralized systems because of multi-users and concurrency
    - Even more difficult in distributed systems because of replicas



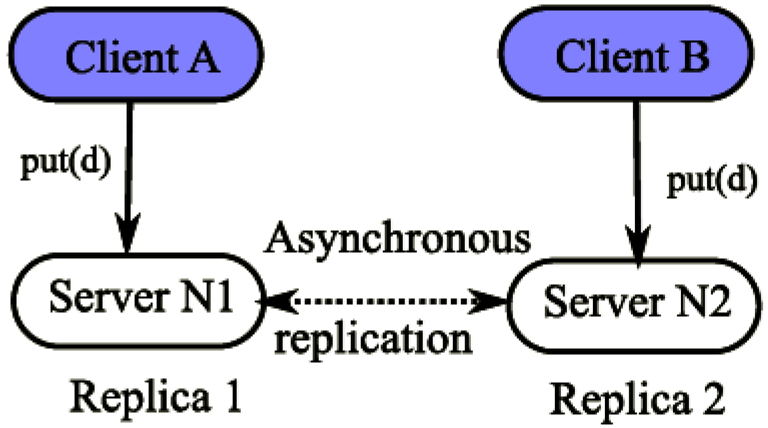
a) Eager replication with primary copy



b) Lazy replication with primary copy
(a.k.a Master-Slave replication)



c) Eager replication, distributed



d) Lazy replication, distributed
(a.k.a. Master-Master replication)

Consistency Management

- Strong consistency: requires a (slow) synchronous replication, and possibly heavy locking mechanisms
  - SQL databases favour consistency over availability
    - The two phase commit (2PC) protocol is the algorithm of choice to ensure strong consistency (ACID properties) in a distributed setting
    - One of the reasons of the ’NoSQL’ trend

Consistency Management

- Weak consistency: accepts to serve some requests with outdated data
  - **Eventual consistency**: the system is guaranteed to converge towards a consistent state based on the last version
  - Data reconciliation: in a system that is not eventually consistent, conflicts occur and the application must determine the new current one given the two conflicting copies

Properties of a Distributed System

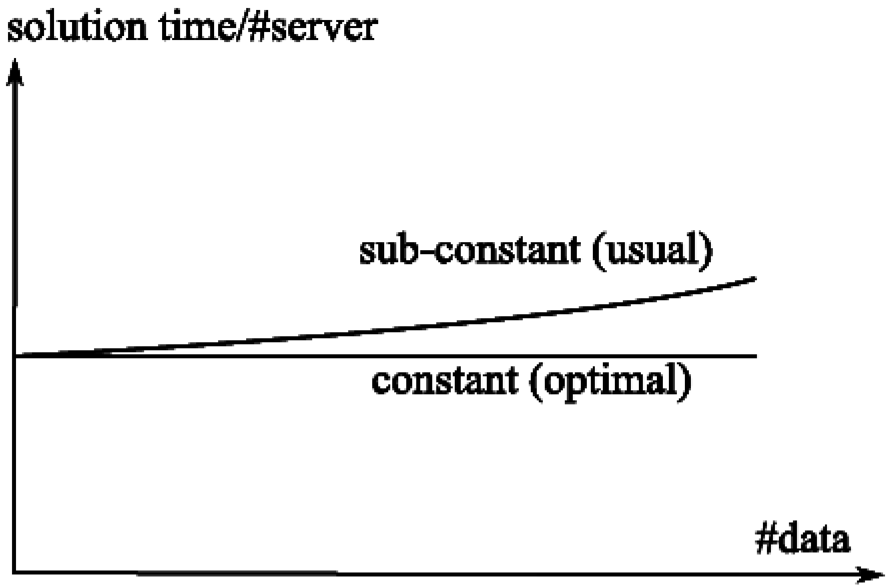
- (1) Reliability
- (2) Scalability
- (3) Availability
- (4) Efficiency

(1) Reliability

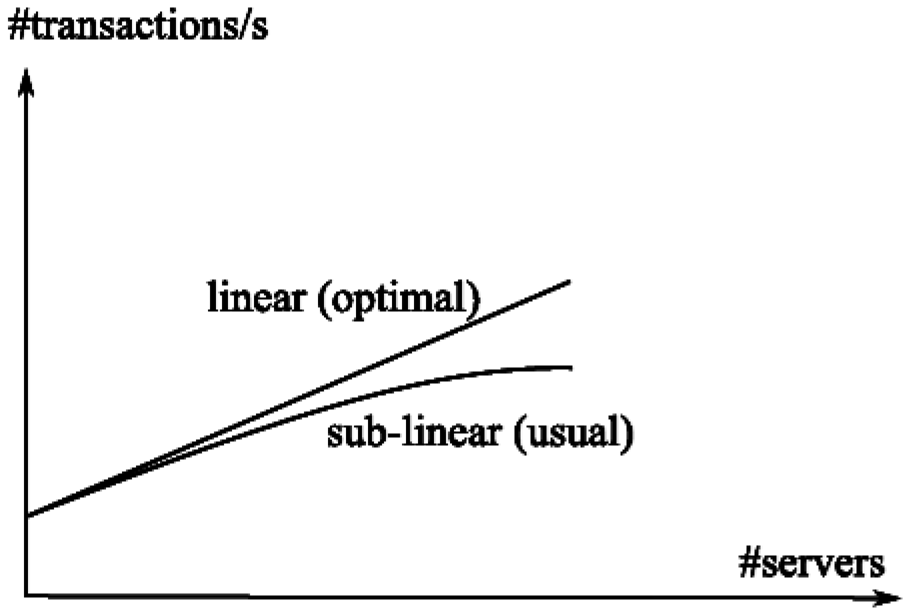
- The ability of a distributed system to deliver its services even when one or several of its software/hardware components fail
  - Based on the assumption that a participating machine affected by a failure can always be replaced by another one, and not prevent the completion of a requested task
  - Relies on *redundancy* of both software and data

(2) Scalability

- The ability of a distributed system to continuously evolve in order to support an ever-growing amount of tasks
  - Distribute evenly the task load to all participants
  - Ensure a negligible distribution management cost



Weak scaling: the solution time per server remains constant as the dataset grows



Strong scaling: the global throughput raises linearly with the number of servers (fixed problem size)

(3) Availability

- The capacity of a distributed system to limit as much as possible its latency
  - Failure detection
    - Monitor the participating nodes to detect failures as early as possible (usually via “heartbeat” messages)
    - Design quick restart protocols
  - Replication on several nodes
    - May be synchronous or asynchronous

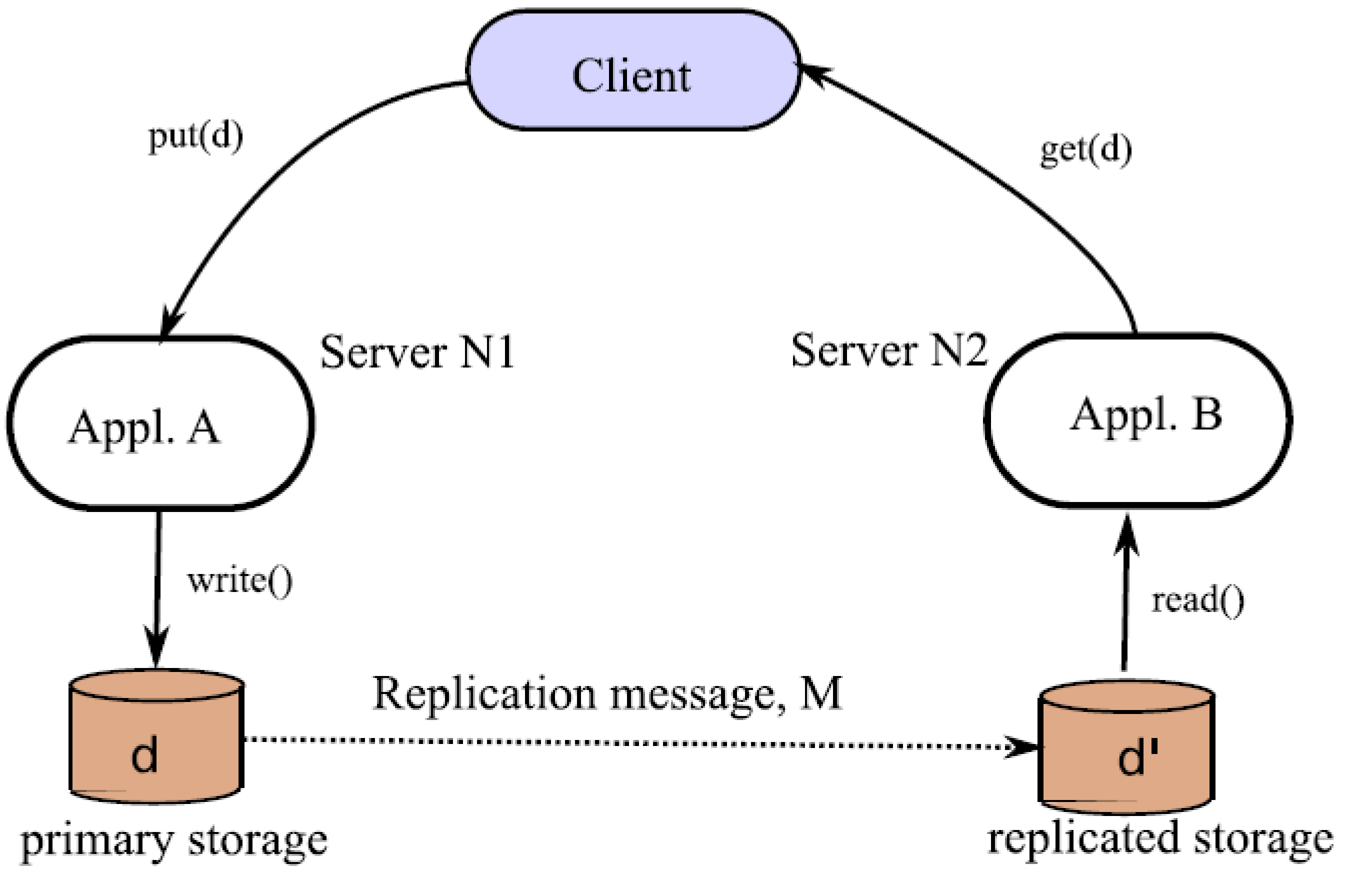
(4) Efficiency

- Two usual measures:
  - The response time (or latency) which denotes the delay to obtain the first item
    - unit cost: the *number of messages* globally sent by the nodes of the system, regardless of the message size;
  - The throughput (or bandwidth) which denotes the number of items delivered in a given period unit (e.g., a second).
    - unit cost: the *size of messages* representing the volume of data exchanges.

The CAP Theorem

- No distributed system can simultaneously provide all three of the following properties
  - **C**onsistency : all nodes see the same data at the same time
  - **A**vailability: node failures do not prevent survivors from continuing to operate
  - **P**artition tolerance: the system continues to operate despite arbitrary message loss

The CAP Theorem



Failure Management

- Failure recovery in <u>centralized</u> systems
  - The state of a (data) system is the set of item committed by the application.
  - Updating “in place” is considered as inefficient because of disk seeks. Instead, update are written in main memory and in a sequential log file.
  - Failure? The main memory is lost, but all the committed transactions are in the log: a REDO operations is carried out when the system restarts.

As implemented in all DBMSs

Failure Management

- Failure recovery in <u>distributed</u> systems
  - When do we know that a component is in failed state? Periodically send message to each participant.
  - Does the centralized recovery still hold? Yes, providing the log file is accessible ...

Leslie Lamport

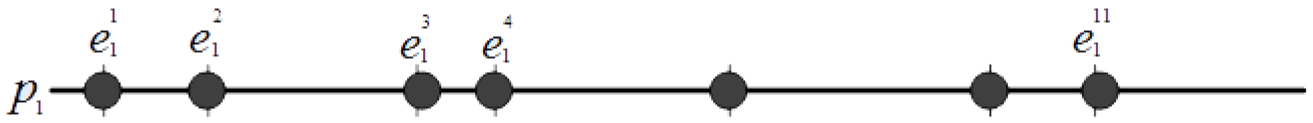
Space-Time Diagram

- An abstract model of distributed systems
  - Processes
    - A process is characterized by its *state* (the ensemble of information we need to restart a process after it was suspended).
    - An *event* is a change of state of a process.
  - (Communication) Channels
    - A communication channel provides the means for processes or threads to communicate with one another and coordinate their actions by exchanging messages.

Space-Time Diagrams

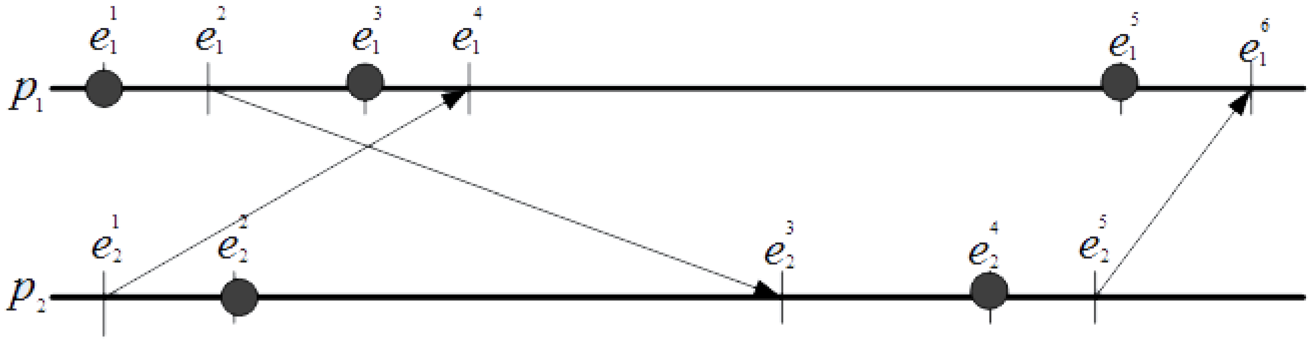
- A space-time diagram displays local and communication events during a process lifetime.
  - Local events are small black circles.
  - Communication events in different processes are connected by lines from the send event and to the receive event.

Space-Time Diagrams



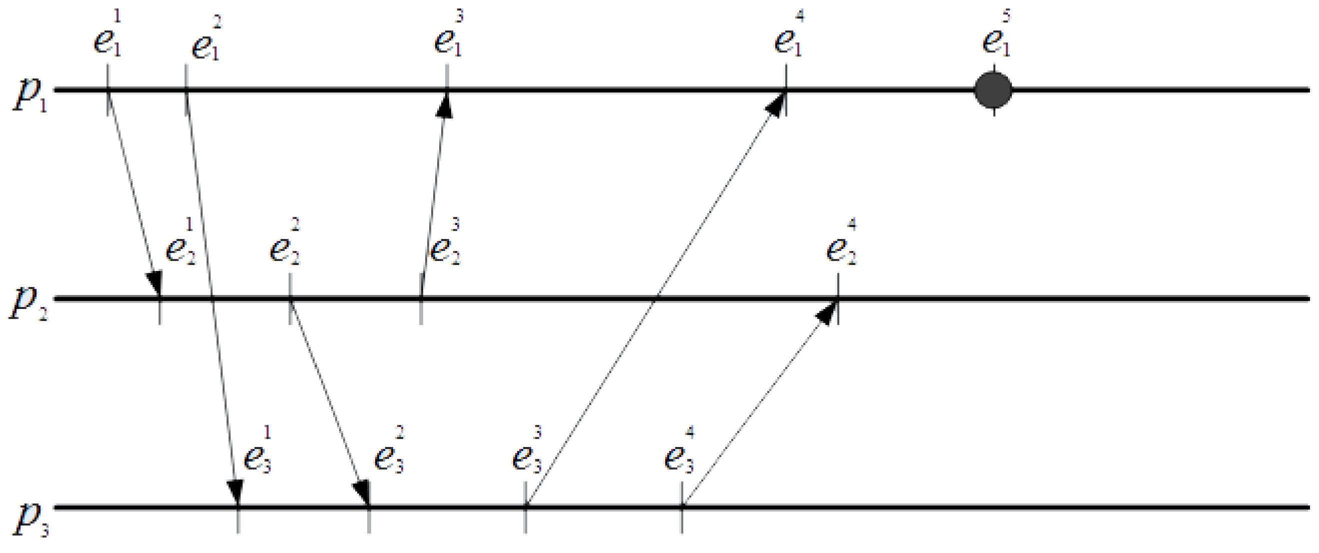
(a)

Space-Time Diagrams



(b)

Space-Time Diagrams



(c)

Happened-Before Relation

- $a \rightarrow b$: **logically** a ***must*** have happed before b
  - For local events

    if $e_i^k, e_i^l \in h_i$ and $k < l$ then $e_i^k \rightarrow e_i^l$.
  - For communication events
    - The cause should precede its effect

    if $e_i^k = send(m)$ and $e_j^l = receive(m)$ then $e_i^k \rightarrow e_j^l$.
  - Transitivity

    if $e_i^k \rightarrow e_j^l$ and $e_j^l \rightarrow e_m^n$ then $e_i^k \rightarrow e_m^n$.

Logical Clocks

- A logical clock is an abstraction necessary to ensure the clock condition in the absence of a global clock.

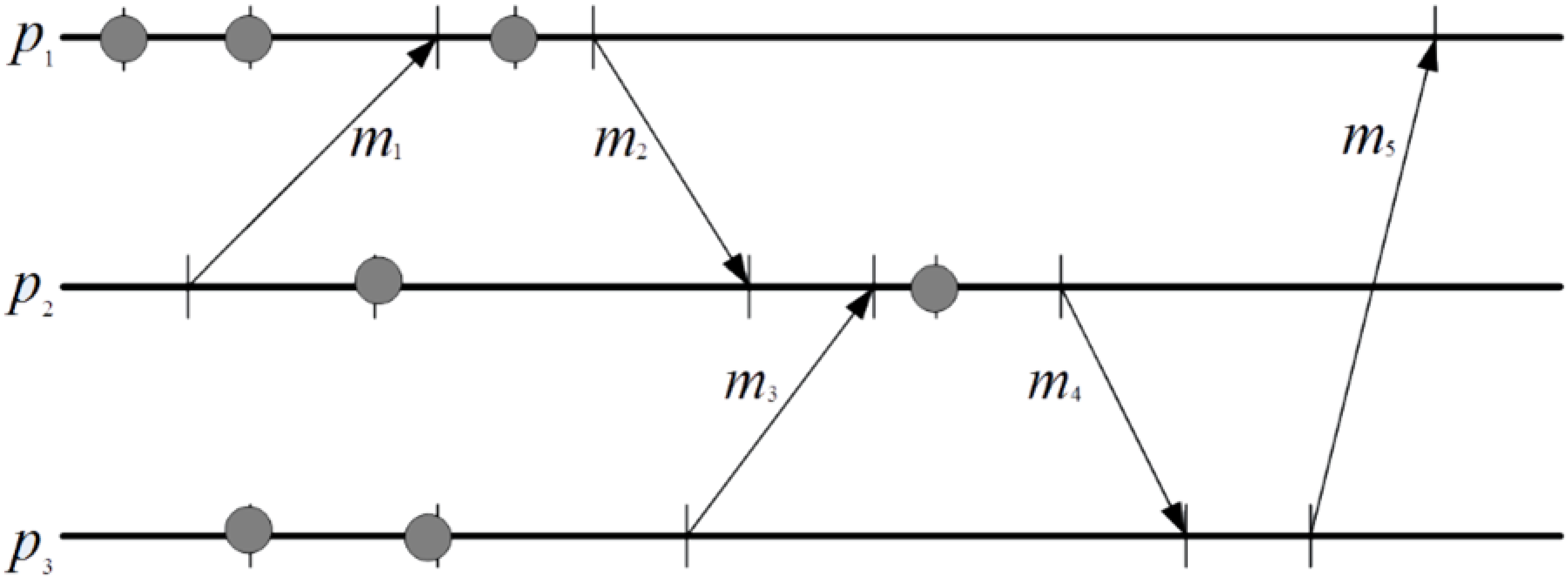
Lamport Timestamps

- A process maps events to positive integers.
  - $\mathrm{LC}(e)$ the local variable associated with event e.
- Each process time-stamps each message m it sends with the value of the logical clock at the time of sending:
  - $\mathrm{TS}(m) = \mathrm{LC}(\mathrm{send}(m))$.

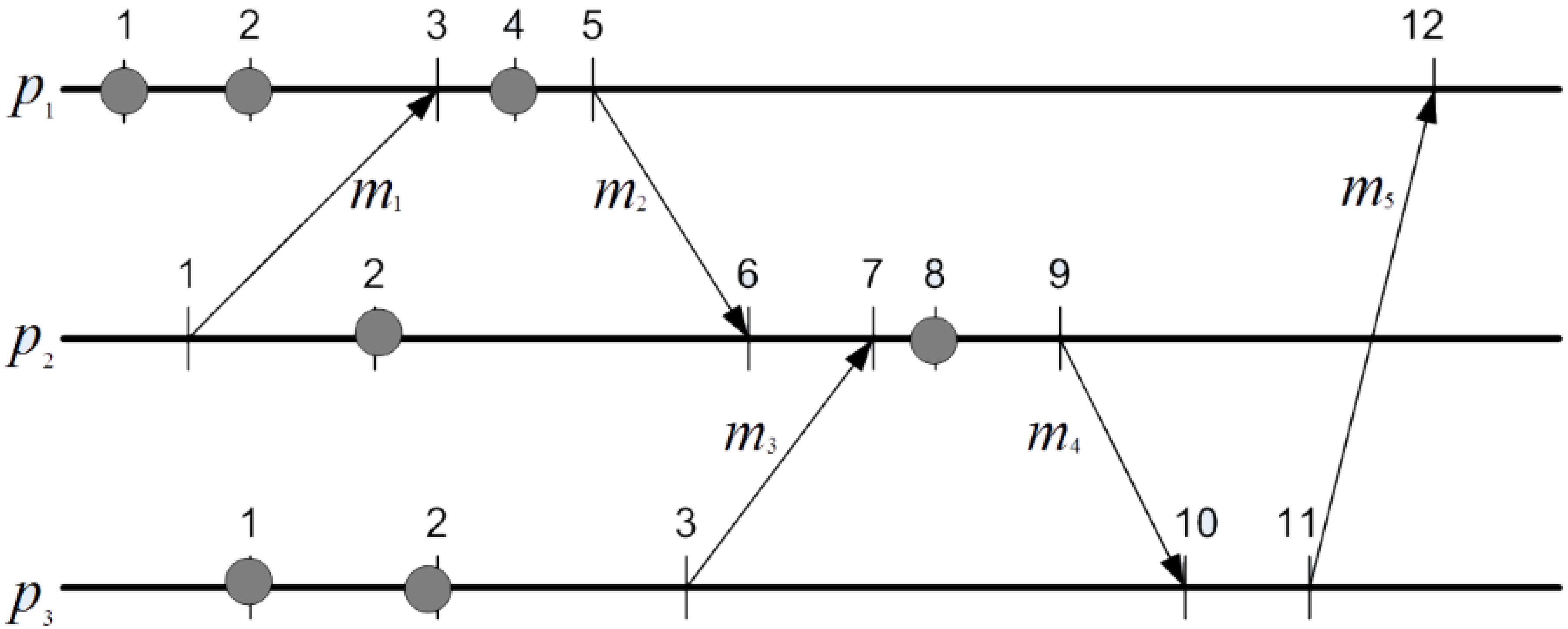
Lamport Timestamps

- The rules to update the logical clock:
  - LC(e) = LC+1 if e is a local event or send(m)
  - LC(e) = max (LC+1, TS(m)+1) if e is receive(m)

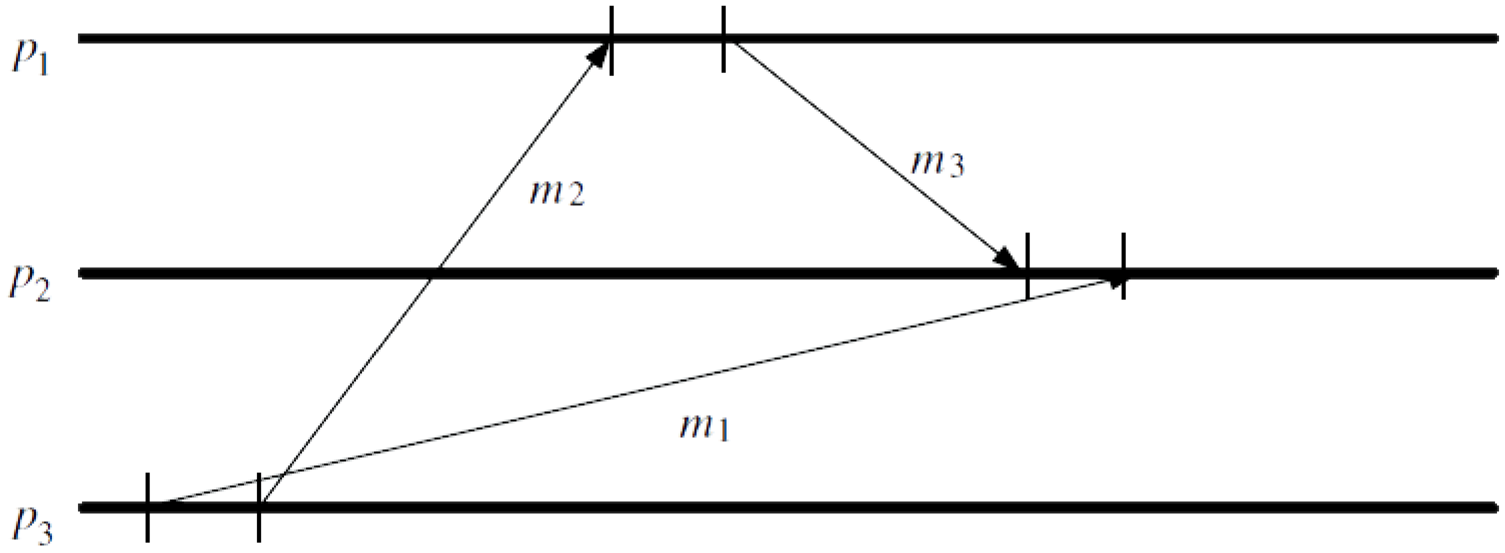
Lamport Timestamps



Lamport Timestamps



Lamport Timestamps



Clock Consistency Condition

- A Lamport clock does not allow a global ordering of all events.
- It lacks an important property, *gap detection*
  - Given two events e and e' and their logical clock values, $\mathrm{LC}(e)$ and $\mathrm{LC}(e')$, it is impossible to establish if an event e'' exists such that $\mathrm{LC}(e) < \mathrm{LC}(e'') < \mathrm{LC}(e')$.

Clock Consistency Condition

- A Lamport clock can create a *partial* causal ordering of events between processes.
  - Given a logical clock following these rules, the following relation is true:
    if one event comes before another, then that event's logical clock comes before the other's.
  - This relation only goes one way, and is called the <u>clock consistency condition</u>:
    if $a \rightarrow b$ then $\mathrm{LC}(a) < \mathrm{LC}(b)$

On the other way around, if $\mathrm{LC}(a) < \mathrm{LC}(b)$, then
(i) a must have happened-before b, or
(ii) it is impossible to tell which one is earlier than the other.

Clock Consistency Condition

- The **strong** clock consistency condition, which is two way, i.e.,
  if $a \rightarrow b$ then $\mathrm{LC}(a) < \mathrm{LC}(b)$ and vice versa,
  can be obtained by other techniques such
  as *vector clocks*.

Snapshot Algorithms

- Checkpoint-Restart Procedures
  - Many cloud computations run for extended periods of time on multiple servers.
  - Checkpoints are taken periodically in anticipation of the need to restart a process when one or more systems fail.
  - When a failure occurs, the computation is restarted from the last checkpoint rather than from the beginning.

Snapshot Algorithms

- Snapshots
  - Intuitively, the construction of the global state is equivalent to taking snapshots of individual processes and then combining these snapshots into a global view.
  - Yet, combining snapshots is straightforward if and only if all processes have access to a global clock and the snapshots are taken at the same time.

Chandy-Lamport Algorithm

- A protocol to construct consistent global states.
  - The global state of a distributed system consisting of several processes and communication channels is the *union* of the states of the individual processes and channels.
- It has three steps.

Chandy-Lamport Algorithm

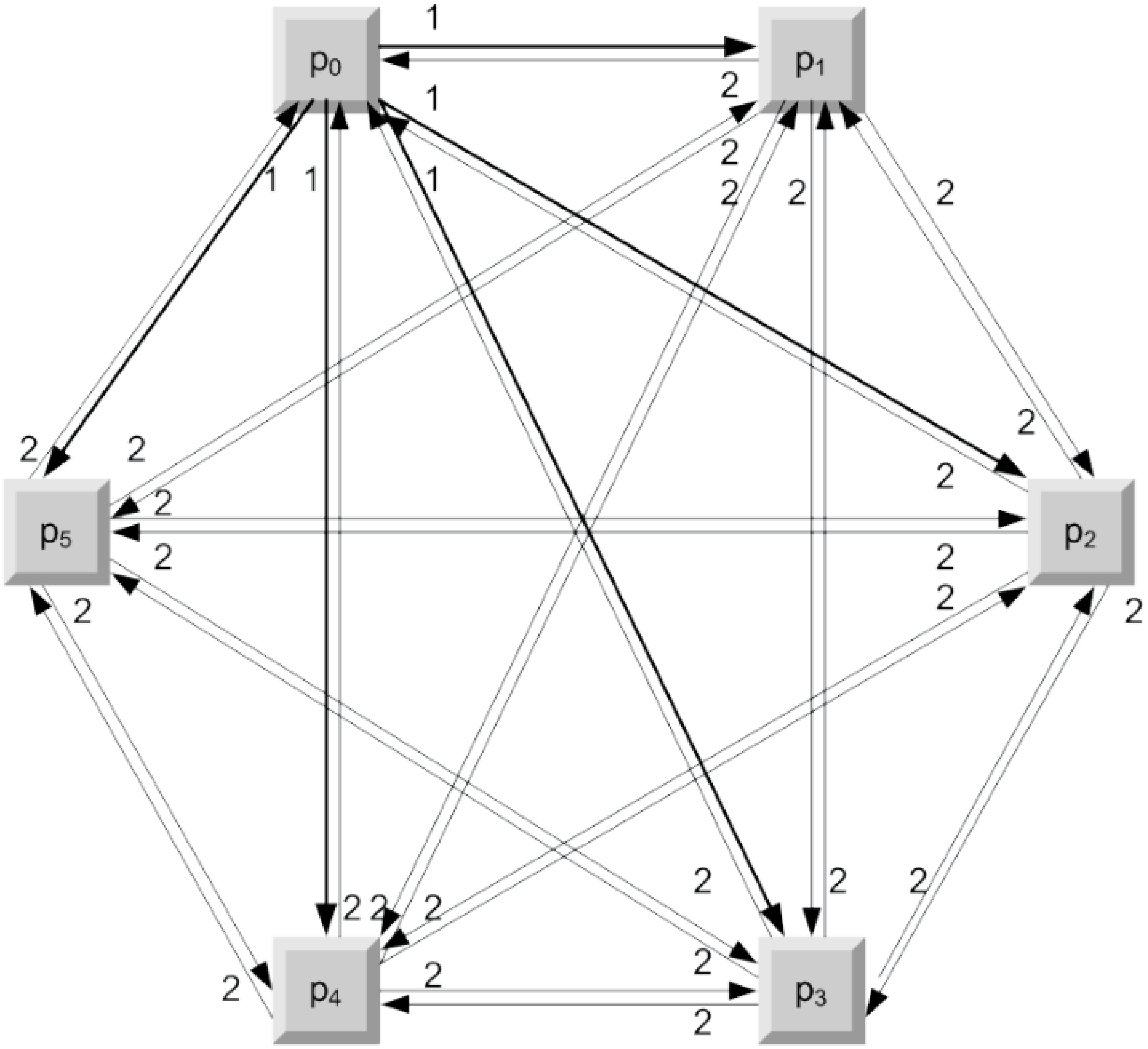
– (1) Process p_0 sends to itself a "*take snapshot*" message.

Chandy-Lamport Algorithm

– (2) Let p_{f} be the process from which p_{i} receives the *take snapsh*ot message for the first time. Upon receiving the message, the process p_{i} records its local state σ_{i} and relays the message "*take snapshot*" along all its outgoing channels without executing any events on behalf of its underlying computation;
  channel state $\xi_{\mathrm{f,i}}$ is set to empty and process p_{i} starts recording messages received over each of its incoming channels.

Chandy-Lamport Algorithm

– (3) Let p_s be the process from which p_i receives the "*take snapshot*" message after the first time. Process p_i stops recording messages along the incoming channel from p_s and declares channel state $\xi_{\mathrm{s,i}}$ between processes p_s and p_i as those messages that have been recorded.


p0
p1
p2
p3
p4
p5
1
1
1
1
1
2
2

Consensus Protocols

- The processes in a distributed system are often required to agree on some data value that is needed during computation.
  - For example,
    whether to commit a transaction to a database,
    agreeing on the identity of a leader.

Consensus Protocols

- Protocols that solve consensus problems are designed to deal with limited numbers of *faulty* processes.
  - Clients send requests, propose a value and wait for a response;
  - The goal is to get the set of processes to reach consensus on a single proposed value.

Process Failure Types

- (1) Crash failure: the process abruptly stops and does not resume.

Process Failure Types

- (2) Byzantine failure: the process fail in *arbitrary* ways (due to malfunctioning hardware, buggy software, malicious attacks from hackers, etc.)
  - Omission failures, e.g., failing to receive a request or send a response.
  - Commission failures, e.g., corrupting the local state, sending an incorrect/inconsistent response to the request.

Of the two types of failures,
Byzantine failures are far more disruptive.

Paxos

- A family of protocols to reach consensus based on a finite state machine approach.
  - It was first published in 1989 and named after a fictional legislative consensus system used on the Paxos island in Greece.

Paxos

- The basic Paxos protocol assumes
  - Processes run on processors and communicate through a network;
    processors and network may experience failures, but ***not*** Byzantine failures.

Paxos

- The Processors
  - (i) operate at arbitrary speeds;
  - (ii) have stable storage and may rejoin the protocol after a failure;
  - (iii) send messages to one another.
- The network:
  - (i) may lose, reorder, or duplicate messages;
  - (ii) messages are sent asynchronously; it may take arbitrary long time to reach the destination.

Paxos

- The basic Paxos protocol has *two phases*.
  - Phase I
    - Proposal preparation
    - Proposal promise
  - Phase II
    - Accept request
    - Accepted

Paxos

- The benefit of the Paxos protocols is the guarantee of *three safety properties*
  - Non-triviality: Only proposed values can be learned.
  - Consistency: at most one value can be learned.
  - Liveness: if a value *v* has been proposed, eventually every learner will learn some value, provided that sufficient processors remain non-faulty.

Paxos

- Applications
  - Apache *ZooKeeper*
  - Google *Chubby*

RESTful APIs

- **REST** = “**RE**presentational **S**tate **T**ransfer”
  - A term coined by Roy Fielding (one of the principal authors of HTTP 1.0 and 1.1) in his Ph.D dissertation to describe an *architecture style* of networked systems
  - The Web is an example of a REST system
  - REST is not a standard, but it does prescribe the use of standards like URI, HTTP, etc.

REST

- “Representation State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (**represent**ing the next **state** of the application) being **transfer**red to the user and rendered for their use.” --- Roy Fielding

Resource

- Things vs Actions (Nouns vs Verbs)
  - In contrast to SOAP
- A resource can be essentially any coherent & meaningful concept that may be addressed
- Identified by URIs
  - Multiple URIs may refer to the same resource

Representation

- A document that captures the current or intended state of a resource
  - Example
    - Resource: person (Dr Zhang)
    - Service: contact information (GET)
    - Representation: name, address, phone-number, etc.
- Typically in JSON or XML format

HTTP Methods

- Resource: Collection URI, e.g., http://example.com/resources/
  - **GET**: List the URIs and perhaps other details of the collection's members
  - **PUT**: Replace the entire collection with another collection
  - **POST**: Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation
  - **DELETE**: Delete the entire collection

HTTP Methods

- Resource: Element URI, e.g., http://example.com/resources/item17
  - **GET**: Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type
  - **PUT**: Replace the addressed member of the collection, or if it doesn't exist, create it
  - **POST**: Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.
  - **DELETE**: Delete the addressed member of the collection.

HTTP Methods

- The GET method should be *safe* (or *nullipotent*)
  - calling it produces no side-effects.
- The GET, PUT and DELETE methods should be *idempotent*
  - multiple identical requests should have the same effect as a single request.

HTTP Methods

- Example: the Web service of an online bookshop
  - List all books
    - GET http://www.bookshop.com/books
  - Show the details of a book
    - GET http://www.bookshop.com/book/isbn12345
  - Place orders
    - POST http://www.bookshop.com/orders

```
<purchase-order>
  <item> ... </item>
</purchase-order>
```

HTTP Status Codes

- 200: OK
- 201: Created
- 400: Bad Request
- 401: Unauthorized
- 404: Not Found
- 405: Method Not Allowed
- 409: Conflict
- 500: Internal Server Error

Service	Description
List Hosted Services	Lists the hosted services available under the current subscription.
`GET https://management.core.windows.net/<subscription-id>/services/hostedservices`	
Get Hosted Service Properties	Retrieves system properties for the specified hosted service. These properties include the service name and service type; the name of the affinity group to which the service belongs, or its location if it isn't part of an affinity group; and, optionally, information about the service's deployments.
`GET https://management.core.windows.net/<subscription-id>/services/hostedservices/<service-name>`	

Create Deployment	Uploads a new service package and creates a new deployment on staging or production.
	
	`POST`
	`https://management.core.windows.net/<subscription-id>/services/`
	`hostedservices/`
	`<service-name>/deploymentslots/<deployment-slot-name>`

Get Deployment	May be specified as follows. Note that you can delete a deployment either by specifying the deployment slot (staging or production) or by specifying the deployment's unique name.
	
	`GET`
	`https://management.core.windows.net/<subscription-id>/services/`
	`hostedservices/`
	`<service-name>/deploymentslots/<deployment-slot>/`
	`GET`
	`https://management.core.windows.net/<subscription-id>/services/`
	`hostedservices/`
	`<service-name>/deployments/<deployment-name>/`

Swap Deployment	Initiates a virtual IP swap between the staging and production deployment slots for a service. If the service is currently running in the staging environment, it's swapped to the production environment. If it's running in the production environment, it's swapped to staging. This is an asynchronous operation whose status must be checked using Get Operation Status.
`POST` `https://management.core.windows.net/<subscription-id>/hostedservices/<service-name>`	
Delete Deployment	Deletes the specified deployment. This is an asynchronous operation.
`DELETE` `https://management.core.windows.net/<subscription-id>/services/hostedservices/<service-name>/deploymentslots/<deployment-slot>` `DELETE` `https://management.core.windows.net/<subscription-id>/services/hostedservices/<service-name>/deployments/<deployment-name>`	

Six Constraints

- (1) Client-Server
- (2) Uniform interface
- (3) Stateless
- (4) Cacheable
- (5) Layered system
- (6) Code on demand

(1) Client-Server

- Assume a distributed architecture that consists of clients and servers
  - Clients initiate requests to servers
  - Servers process requests and return appropriate responses
- Separation of concerns
  - Benefits?

(2) Uniform Interface

- All resources are accessed with a generic interface between client and server
  - URIs (resource names)
  - HTTP verbs (GET, PUT, POST, DELETE)
  - HTTP response (status, body)
- It simplifies and decouples the architecture, which enables each part to evolve independently
- Fundamental to RESTful design

(3) Stateless

- Server contains no client state
- Each request from any client to server must contain enough context (i.e., all the information necessary) to process it
  - All messages are self-descriptive
- Any session state is held on the client

(4) Cacheable

- To improve network efficiency, clients should be able to cache responses (representations)
- Therefore responses must label themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data
  - Implicitly
  - Explicitly
  - Negotiated

(5) Layered System

- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
  - proxies, gateways, and firewalls etc.
- Intermediary servers may improve system scalability (e.g., by enabling load-balancing and providing shared caches) and they may also enforce security policies

(6) Code on Demand

- Servers are able to temporarily extend or customize the functionality of a client by the transfer of executable code
  - compiled components, e.g., Java applets
  - client-side scripts, e.g., JavaScript
- This is the only optional constraint

Being RESTful

- Complying with those constraints, and thus conforming to the REST architectural style, is generally referred to as being “RESTful”
  - Violating any constraint other than (6) means the service is not strictly RESTful
    - e.g., three-legged OAUTH2
  - Being RESTful enables *scalability*, *simplicity*, *modifiability*, *visibility*, *portability*, and *reliability*

REST vs SOAP

- REST reuses the vocabulary of the well-known, well-defined HTTP protocol
- SOAP encourages each application designer to define new, application specific methods that supplant HTTP methods
  - It disregards many of HTTP's existing capabilities (such as authentication, caching, and content-type negotiation)
  - It can work equally well over raw TCP, named pipes, message queues, etc.

Flask RESTful

web development,
one drop at a time

https://flask-restful.readthedocs.io/en/latest/quickstart.html

https://www.kdnuggets.com/2019/01/build-api-machine-learning-model-using-flask.html

Take Home Messages

- Parallel Computing
  - speed-up, parallelism
  - locking
- Distributed Systems
  - CAP theorem
  - logical clock
  - snapshot algorithm, consensus protocol
  - RESTful APIs