

Introducción - Clase 1

Sistema Distribuido: Es una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente. Los procesos y recursos están suficientemente distribuidos en varias computadoras. Se busca sincronización, un mejor rendimiento, escalabilidad y consistencia. (Peer to peer, BlockChain, todos se conectan con todos)

Sistema Centralizado: Todos a 1 servidor general (Proyectos de Pingeso)

Sistema descentralizado: Es un sistema informático en red en el que los procesos y recursos están necesariamente distribuidos entre varias computadoras. (Varios servidores en distintas partes del mundo y los clientes se comunican dependiendo de donde estén) (Netflix)

Un sistema descentralizado se convierte en distribuidos cuando hay más de un link entre nodos

Vista integrativa: Se conectan computadores existentes entre sí.

Vista expansiva: Un sistema de computadores se expande con más computadores. Ayuda a la dependencia.

Se dice que las soluciones centralizadas no escalan, lo cual no es cierto. DNS está lógicamente centralizado, pero físicamente distribuido (Es un árbol prácticamente) Descentralizado en muchas organizaciones.

También se dice que las soluciones centralizadas tienen un único punto de falla, lo cual generalmente no es verdad dado que un único punto de falla no es algo malo, es más fácil de administrar y se puede hacer más robusto. (Con solución centralizada es más fácil arreglar bugs en comparación a distribuido)

El sistema distribuido es complejo y se debe hacer que los recursos sean fácilmente accesibles; debería ocultar el hecho de que los recursos estén distribuidos a través de una red; debería estar abierto (ofrece componentes que pueden ser usados por otros sistemas y suele estar constituido por componentes que vienen de otros lados); y debe ser escalable. Se quiere replicar información para que todos tengan la misma información y para reducir tiempos de acceso.

Ejemplos de compartición de recursos son Google Drive, Dropbox, OneDrive, multimedia streaming con peer-to-peer, servicios de correos y servicios de web hosting.

Transparencia es esconder que los procesos están físicamente distribuidos. Transparencia esconde lo que hay detrás. Un middleware es transparente, similar a Windows. Transparencia se maneja mediante un middleware. Existe transparencia de acceso, ubicación, reubicación, migración, replicación, concurrencia y fallas.

Cumplir todas las transparencias es casi imposible. Esconder las latencias no se puede hacer, por lo que esconder las fallas en la red o nodos es imposible. No se puede distinguir un computador lento a uno que está fallando y no se puede asegurar que un computador vaya a hacer una operación antes de que crashee. Además, la transparencia costa performance.

Exponer la distribución puede estar bien, como en servicios basados en ubicación (WhatsApp, Waze, Uber) y cuando facilita a un usuario entender que está pasando. Transparencia distribuida esta bien pero es difícil y nunca se puede lograr transparencia total.

Los sistemas distribuidos abiertos deben ser extensibles, portables, definir interfaces e interporar.

Dependencia: Un componente provee servicios a clientes. Un componente requiere de servicios desde otros computadores. (Un componente depende de otros componentes). Los componentes son procesos o canales. Para la dependencia se requiere mantenibilidad, seguridad, disponibilidad y confiabilidad.

Un sistema distribuido que no es seguro no puede ser dependiente. Se necesita confidencialidad e integridad. Autenticación, autorización y confianza.

En sistemas distribuidos se usan llave de seguridad. $K(\text{dato})$ es una llave de K para encriptar datos.

Symmetric cryptosystem

With encryption key $E_K(\text{data})$ and decryption key $D_K(\text{data})$:

if $\text{data} = D_K(E_K(\text{data}))$ then $D_K = E_K$. Note: encryption and decryption key are the same and should be kept **secret**.

[Las dos son iguales](#)

Asymmetric cryptosystem

Distinguish a **public key** $PK(\text{data})$ and a **private (secret) key** $SK(\text{data})$. [¶ 6 ¶](#)

- Encrypt message from Alice to Bob: $\text{data} = \underbrace{SK_{\text{bob}}}_{\text{Action by Bob}}(\overbrace{PK_{\text{bob}}(\text{data})}^{\text{Sent by Alice}})$
- Sign message for Bob by Alice:

$$[\text{data}, \underbrace{H(\text{data}) \stackrel{?}{=} PK_{\text{alice}}(SK_{\text{alice}}(\text{data}))}_{\text{Check by Bob}}] = [\text{data}, SK_{\text{alice}}(\text{data})]$$
Sent by Alice



Secure hashing

In practice, we use secure hash functions: $H(\text{data})$ returns a **fixed-length string**.

- Any change from data to data^* will lead to a completely different string $H(\text{data}^*)$.
- Given a hash value, it is computationally impossible to find a data with $h = H(\text{data})$

Practical digital signatures

Sign message for Bob by Alice:

$$[\text{data}, \underbrace{H(\text{data}) \stackrel{?}{=} PK_{\text{alice}}(\text{sgn})}_{\text{Check by Bob}}] = [\text{data}, H, \text{sgn} = SK_{\text{alice}}(H(\text{data}))]$$



Nunca se explica porque un sistema es escalable. Se puede escalar por numero de usuarios o procesos (tamaño), la máxima distancia entre nodos (geográfica) o número de dominios administrativos (administrativa). La mayoría de los sistemas buscan escalar en tamaño. La solución que dan es tener más soluciones operando en forma independiente y paralelo. El desafío sigue en como escalar de forma geográfica y administrativa.

Cuando se trata de escalar con soluciones centralizadas, se está limitado. Se puede cambiar con una solución descentralizada o distribuida.

En la escalabilidad geográfica no se puede ir simplemente de LAN a WAN. La latencia va a prohibir esto dado que se necesita síncrono. Los enlaces WAN no son confiables, con un cambio se va a tener un fallo. Hay una falta de múltiples puntos de comunicación, por lo que no se puede hacer una simple búsqueda con broadcast.

Para la escalabilidad administrativa, la esencia es que se tienen políticas respecto al uso y pago de administración y seguridad. Hay políticas que conflictúan entre si. Por ejemplo:

- Computational grids: share expensive resources between different domains.
↳ grillas
- Shared equipment: how to control, manage, and use a shared radio telescope constructed as large-scale shared sensor network?

Las redes peer-to-peer no tienen el problema de escalabilidad administrativa. Por ejemplo, BitTorrent, Skype, Spotify.

Para escalar, se puede hacer uso con la comunicación asíncrona. WhatsApp tiene comunicación asíncrona. Pero no todas las aplicaciones se ajustan a la comunicación asíncrona como Zoom, la cual debe ser síncrona.

Otras técnicas de escalar son mover los cálculos a un cliente y otro partitionar data y cálculos a través de múltiples máquinas. Otra forma es usar replicación y caching, hacer copias de la data disponible en distintas máquinas. Hacer replicación es fácil, excepto por las inconsistencias. También se necesita sincronización global y si se hace sincronización global, se necesitan soluciones a gran escala.

Los clientes se conectan al middleware y este actúa como una sola aplicación para los clientes. Formas de comunicación:

- Request Procedure Call (RPC): Llamado de procedimiento local, empaquetado como un mensaje procesado y respondido mediante un mensaje, con el resultado siendo devuelto desde la llamada.
- Middleware orientado al mensaje (MOM): Los mensajes se envían a un contacto lógico donde se suscribe a ciertos mensajes

Hay distintos tipos de sistemas distribuidos, en base a sensores, redes de sensores, decodificadores de casa, telescopios, sensores de sismos, redes de teléfonos

Subtipos de sistemas penetrantes de sistemas distribuidos:

- Omnipresentes: Continuamente presente, interacción continua entre sistema y usuario
- Móvil: Énfasis en Móviles
- Sensor Énfasis en sensor y conciencia sobre el contexto

UBIQUITOUS SYSTEMS

↳ omnipresente

Core elements

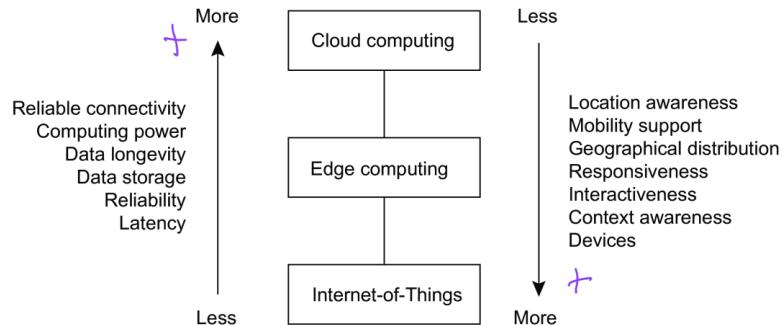
1. (Distribution) Devices are networked, distributed, and accessible transparently
2. (Interaction) Interaction between users and devices is highly unobtrusive
3. (Context awareness) The system is aware of a user's context to optimize interaction
4. (Autonomy) Devices operate autonomously without human intervention, and are thus highly self-managed
5. (Intelligence) The system as a whole can handle a wide range of dynamic actions and interactions

MOBILE COMPUTING

Distinctive features

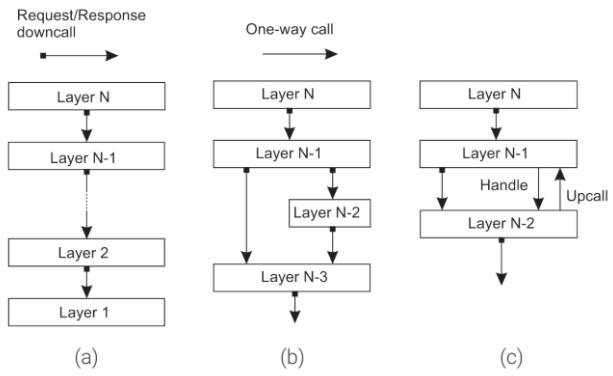
- A myriad of different mobile devices (smartphones, tablets, GPS devices, remote controls, active badges).
- Mobile implies that a device's location is expected to change over time ⇒ change of local services, reachability, etc. Keyword: **discovery**.
- Maintaining stable communication can introduce serious problems.
- For a long time, research has focused on directly sharing resources between mobile devices. It never became popular and is by now considered to be a fruitless path for research.

Edge computing: Dispositivo hace los cálculos en los bordes (En tesis las calculos se entregan en el auto)

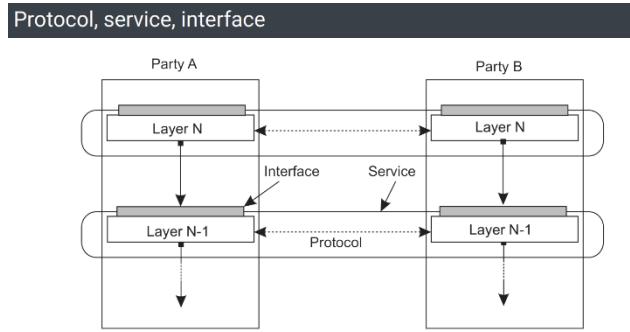


Arquitectura – Clase 2

Comunicación por capas:



Comunicación por protocolos:



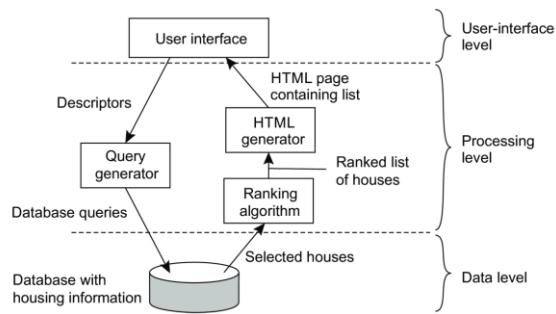
En sockets, primero hay que mirar el cliente. El cliente se conecta al servidor y se bloquea hasta que se acepta la conexión.

Capa de aplicación: Interfacea con los usuarios, es la que el usuario ve. (Front)

Capa de procesamiento: Todo lo relacionado con el procesamiento (Back)

Capa de datos: Contiene los datos que el cliente quiere manipular mediante la capa de aplicación (BD)

Example: a simple search engine



Una desventaja de esta arquitectura es la dependencia de ciertas capas.

En un estilo basado en objetos, los componentes son objetos. Se puede llamar un método de otro objeto y cada objeto tiene su estado, métodos e interfaz. Una de las características

de este estilo es la encapsulación, donde se ofrecen métodos sin revelar su implementación.

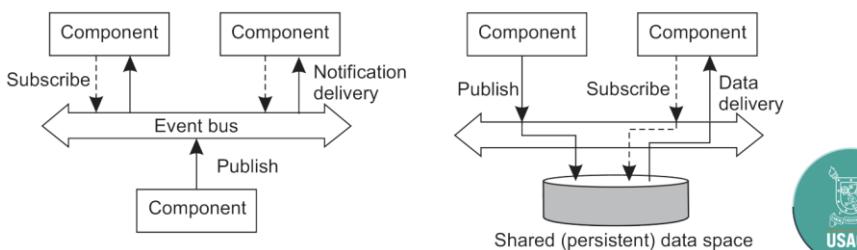
La arquitectura RESTful tiene las operaciones POST, GET, PUT, DELETE.

Temporal and referential coupling

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Mailbox: Alguien coloca un mensaje y nunca se pierde

Event-based and Shared data space



"Temporal coupling means that processes that are communicating will both have to be up and running"

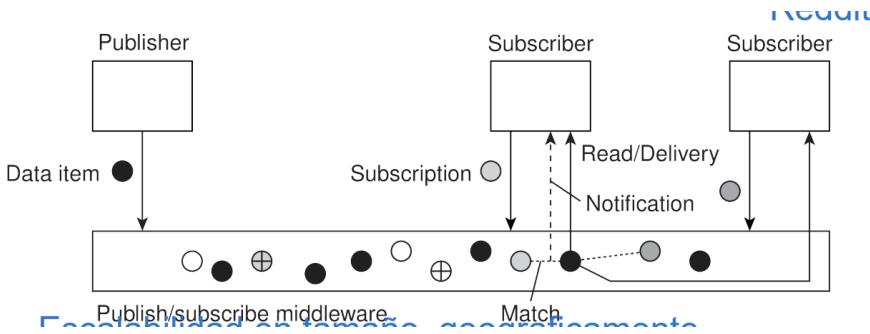
En el contexto de los sistemas distribuidos, la coordinación entre los distintos nodos o procesos es un aspecto crucial. El acoplamiento temporal y referencial son dos conceptos clave en esta coordinación.

Acoplamiento Temporal: Este se refiere a la necesidad de que el remitente y el receptor o receptores deben tener una línea temporal común para la comunicación, es decir, deben existir al mismo tiempo. En un sistema con acoplamiento temporal, un mensaje solo puede ser enviado si el receptor está disponible para recibirla en ese momento. Si el receptor no está disponible, el remitente tendrá que esperar o reintentar el envío más tarde. Esto puede ser una limitación en sistemas donde los nodos pueden tener tiempos de actividad variables o intermitentes. Un ejemplo de acoplamiento temporal sería una llamada telefónica en tiempo real, donde tanto el que llama como el que recibe deben estar disponibles al mismo tiempo para la comunicación 1.

Acoplamiento Referencial: Este tiene que ver con cómo un proceso se refiere o identifica a otro en la comunicación. En un sistema con acoplamiento referencial, un proceso debe conocer el identificador o nombre del otro proceso con el que se quiere comunicar. Esto puede ser una limitación en sistemas donde los procesos pueden cambiar con frecuencia o donde la identidad de los procesos no es fácilmente conocida o accesible. Un ejemplo de acoplamiento referencial sería un correo electrónico, donde necesitamos conocer la dirección de correo electrónico exacta del destinatario para enviar un mensaje 1, 2.

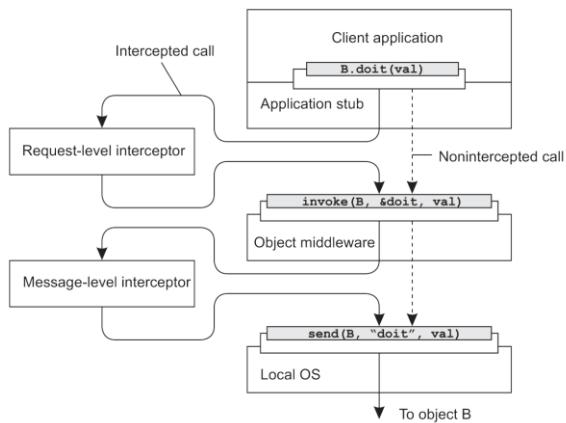
En muchos sistemas distribuidos, se busca minimizar tanto el acoplamiento temporal como el referencial para aumentar la flexibilidad y robustez del sistema. Por ejemplo, en un sistema de colas de mensajes, los mensajes pueden ser enviados sin que el receptor esté disponible (desacoplamiento temporal) y sin que el remitente tenga que conocer la identidad exacta del receptor (desacoplamiento referencial).

Publish and Subscribe: Alguien publica y se suscribe. Basado en tópicos, donde el atributo debe ser igual al valor y en contenidos, donde el atributo debe pertenecer a un rango de valores. (Ej: Blogs, # en Twitter, # en Youtube, TikTok, Reddit)

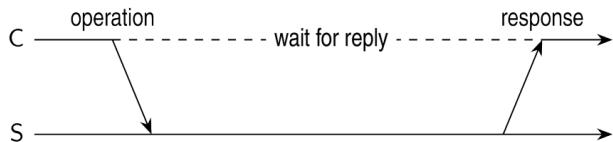


Si hay un dato que hace match con la subscripción, se le envia una notificación. La otra forma es que le llegue una notificación y además se le envie el dato.

Middleware llama funciones que esta en otras maquinas:



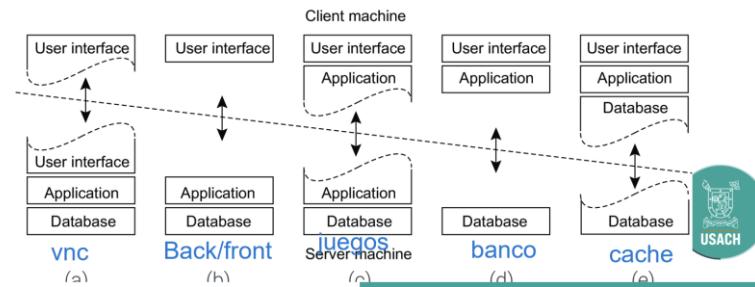
En arquitectura centralizadas, cliente y servidor pueden estar en distintas maquinas. Los servidores ofrecen servicios y los clientes usan estos servicios. Se sigue un modelo requerimiento / respuestas



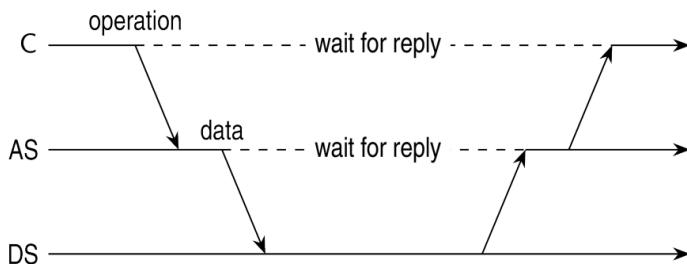
Some traditional organizations

- Single-tiered: dumb terminal/mainframe configuration
- Two-tiered: client/single server configuration
- Three-tiered: each layer on separate machine

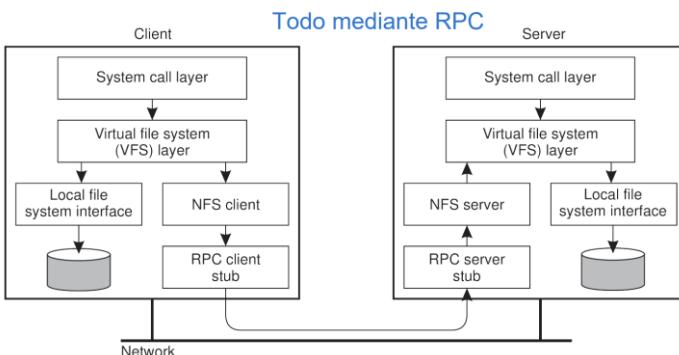
Traditional two-tiered configurations



A veces un servidor tiene que actuar como cliente



Arquitectura NFS:



NFS, o Sistema de Archivos de Red (Network File System), es un protocolo que permite a diferentes computadoras en la misma red acceder y compartir archivos. Este sistema permite que un ordenador, por ejemplo, utilice la información almacenada en otro ordenador como si fuera un disco duro local 2, 3.

En la arquitectura NFS, hay un servidor que tiene los archivos almacenados y permite el acceso de otros dispositivos, que serían los clientes. Desde el lado del cliente, este solicitará acceso al servidor, generalmente a través de comandos. Esto permitirá al cliente interactuar con el servidor, visualizar el contenido, descargar archivos, etc. 2

Vertical distribution

Comes from dividing distributed applications into three logical layers, and running the components from each layer on a different server (machine).

Horizontal distribution

A client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set.

Peer-to-peer architectures

Processes are all equal: the functions that need to be carried out are represented by every process ⇒ each process will act as a client and a server at the same time (i.e., acting as a **servant**).

Arquitectura Vertical: En este tipo de arquitectura, los diferentes componentes o niveles de la aplicación se distribuyen en diferentes sistemas o máquinas. Por ejemplo, podrías tener una aplicación de tres niveles donde la interfaz de usuario se ejecuta en el dispositivo del cliente, la lógica de negocio se ejecuta en un servidor y la base de datos se ejecuta en otro servidor. Esta es una forma de distribución vertical. La ventaja de la arquitectura vertical es que permite separar y aislar diferentes partes de la aplicación para mejorar la eficiencia y la seguridad 4.

Arquitectura Horizontal: En esta arquitectura, los mismos componentes de la aplicación se distribuyen en diferentes sistemas o máquinas. Por ejemplo, podrías tener varias instancias de un servidor web ejecutándose en diferentes máquinas y distribuir las solicitudes de los clientes entre ellas. Esta es una forma de distribución horizontal. La ventaja de la arquitectura horizontal es que permite manejar una mayor carga de trabajo al distribuir las solicitudes entre múltiples instancias de la misma aplicación 4.

Arquitectura Peer-to-Peer (P2P): En una arquitectura P2P, todos los nodos son iguales y pueden actuar tanto como clientes como servidores. Por ejemplo, en una red de compartición de archivos P2P, cada nodo puede proporcionar archivos a otros nodos y descargar archivos de ellos. La ventaja de la arquitectura P2P es que no requiere un servidor central y puede ser muy robusta y escalable. Sin embargo, también puede presentar desafíos en términos de seguridad y control 3, 5.

15.06 ✓

En peer-to-peer cada maquina actúa como cliente y servidor, mientras que en centralizada siempre se tiene una maquina cliente y otro servidor.

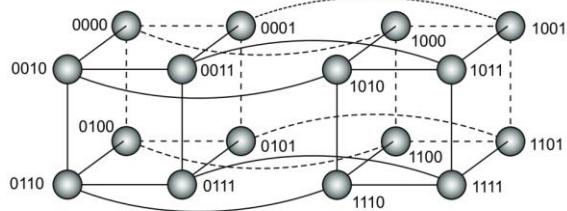
Peer-to-peer estructurada tiene una forma, una topología. Se usa un índice libre semántico. Cada item está asociada con una llave y es usado como índice. Se usa una función Hash

Make use of a **semantic-free index**: each data item is uniquely associated with a key, in turn used as an index. Common practice: use a **hash function**

$$\text{key}(\text{data item}) = \text{hash}(\text{data item's value}).$$

P2P system now responsible for storing (key,value) pairs.

Simple example: hypercube

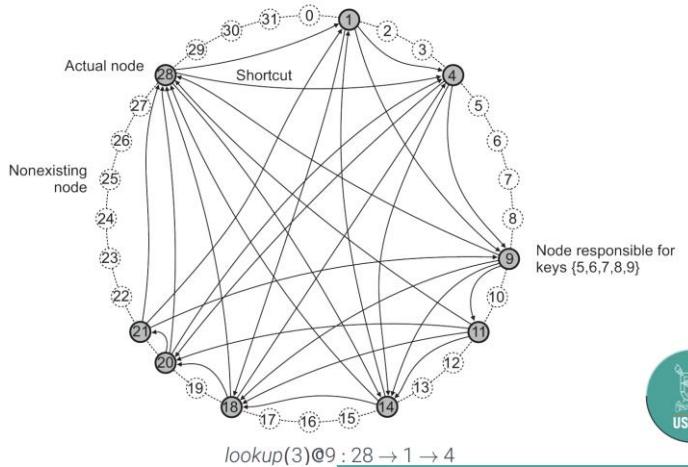


Looking up d with key $k \in \{0, 1, 2, \dots, 2^4 - 1\}$ means routing request to node with identifier k .

Otro ejemplo de p2p estructurado es CHORD

Principle

- Nodes are logically organized in a ring. Each node has an m -bit identifier.
- Each data item is hashed to an m -bit key.
- Data item with key k is stored at node with smallest identifier $id \geq k$, called the successor of key k .
- The ring is extended with various shortcut links to other nodes.



Peer-to-peer no estructurado es con nodos como un grafo aleatorio. Existe cierta probabilidad de ir a un nodo y hay dos formas de buscar:

- Flooding (inundamiento): Nodo u pasa la request del dato d a todos sus vecinos. La request es ignorada si es que el nodo que la recibió ya la ha visto antes, si no, el nodo v (al que le llega la request) busca localmente por d de forma recursiva (Uno manda a todos y todos buscan por todos, con tiempo limitado y costoso, con time to live).
- Random Walk: Nodo u pasa la request por d a un vecino aleatorio v . Si v no tiene d , entonces reenvía la request a otro vecino aleatorio. No tiene un time to leave y después de un tiempo se encuentra el dato, por lo que es mejor.

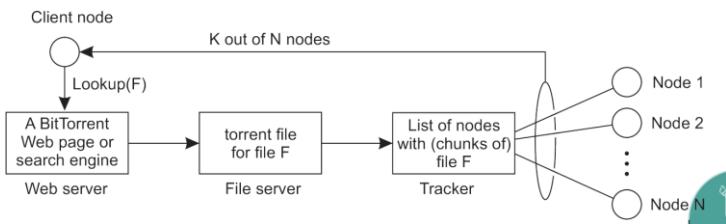
Super peer networks se llaman arquitecturas peer to peer hibridas. No hay simetría, donde hay servidores de indexación para mejorar la performance. Hay super peers y peers débiles. Los peers débiles se conectan a los super peers, pero el problema es elegir quien es el super peer.

Siempre pueden haber nodos maliciosos en nodos peer to peer, los cuales suben datos que no tienen que ver con la red peer to peer

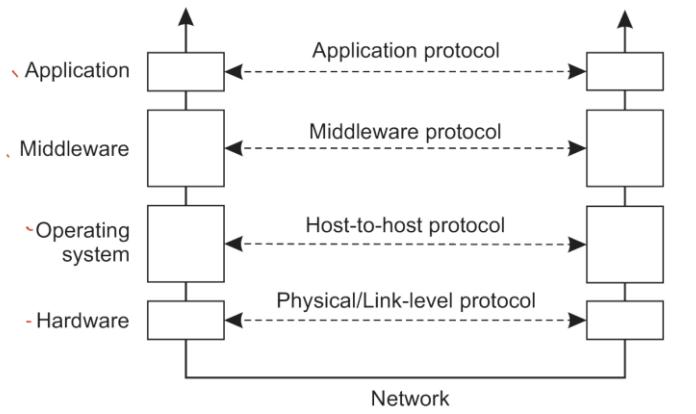
En el caso de BitTorrent:

Principle: search for a file F

- Lookup file at a global directory \Rightarrow returns a **torrent file**
- Torrent file contains reference to **tracker**: a server keeping an accurate account of **active** nodes that have (chunks of) F .
- P can join **swarm**, get a chunk for free, and then trade a copy of that chunk for another one with a peer Q also in the swarm.

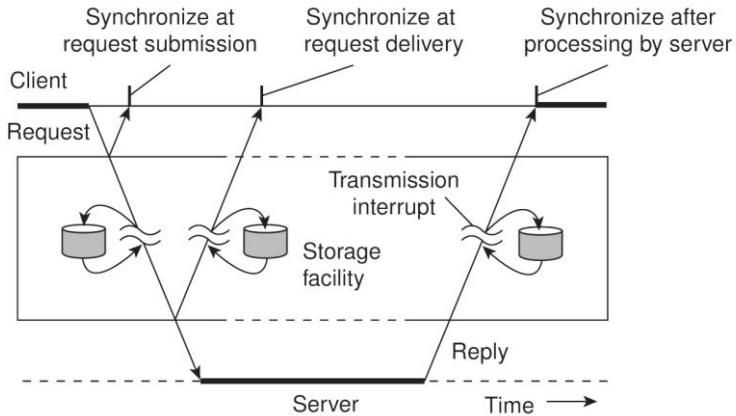


Comunicación – Clase 4 y 6



Comunicación persistente: Hay un storage entre medio. Quiere decir que no importa si es que se envio un mensaje y luego se desconecta, el que lo recibe igual lo va a recibir dado que queda guardado en un storage. Un ejemplo es el correo y WhatsApp.

Transiente es lo contrario. Un ejemplo es una request HTTP / HTTPS



- **Transient communication:** Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
- **Persistent communication:** A message is stored at a communication server as long as it takes to deliver it.

Sincrono: Los dos deben estar conectados para ver el mensaje (EJ: Zoom)

Asincrono: Los dos no deben estar conectados al mismo tiempo (WhatsApp)

Tres lugares de sincronización: Cuando se envía la request, cuando llega la request y cuando se procesa la request

Cliente y servidor esta basado en un modelo transiente síncrono de comunicación. Cliente y servidor deben estar activos. El cliente manda una request y se bloquea hasta que recibe una respuesta y el servidor debe esperar a que llegue una request.

Problemas de la comunicación sincrónica es que el cliente no puede hacer otra cosa mientras espera por una respuesta, las fallas deben ser manejadas de inmediato dado que el cliente espera y para algunos casos, el modelo no es adecuado (EJ: Correo, noticias)

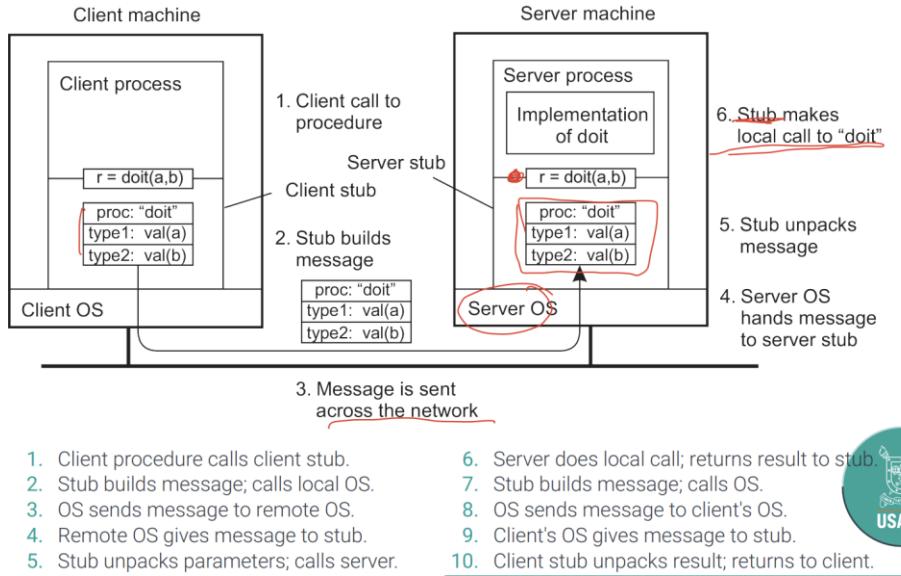
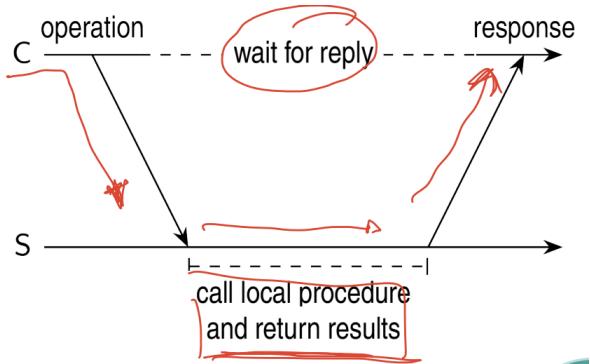
Message-oriented-middleware (mom): Busca una comunicación persistente asincrónica de alto nivel. Procesa y envia mensajes que son encolados y el que envia no tiene que estar esperando inmediatamente por una respuesta y el middleware asegura tolerancia de fallos.

Message-oriented middleware

Aims at high-level persistent asynchronous communication:

- Processes send each other messages, which are queued
- Sender need not wait for immediate reply, but can do other things
- Middleware often ensures fault tolerance

Remote Procedure Call: Es llamar una función y esa función se traduce en lo que el SO de la otra maquina necesita. Opera por el principio de la caja negra y no hay una razón fundamental para no ejecutar un procedimiento en una maquina separada.



El servidor y cliente puede tener distintas representaciones de datos. Wrapping significa transformar un valor a una secuencia de bits por lo que el cliente y servidor deben de estar de acuerdo con la transformación de datos. Cliente y servidor deben de estar de acuerdo en el paso e interpretación de mensajes, teniendo una representación de acuerdo a las maquinas.

Some assumptions

- Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
- All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

Conclusion

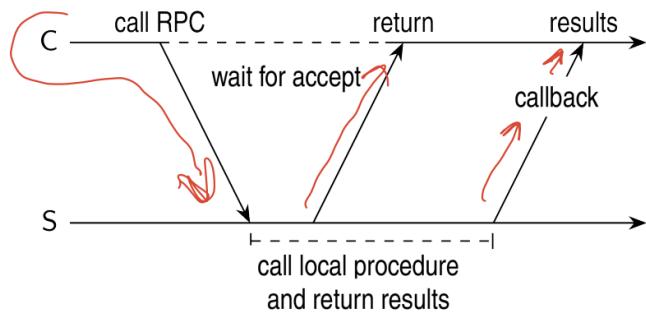
Full access transparency cannot be realized.

A remote reference mechanism enhances access transparency

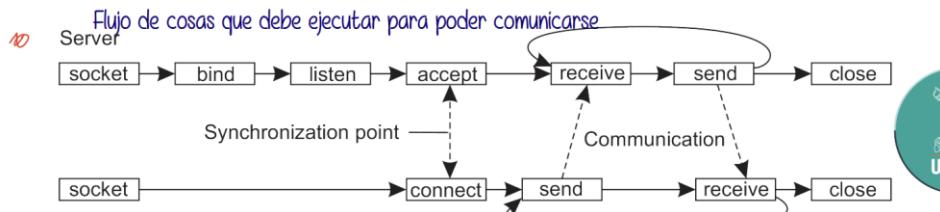
- Remote reference offers unified access to remote data ↗
- Remote references can be passed as parameter in RPCs ↘
- Note: stubs can sometimes be used as such references ↘



RPC Asincrono es distinto, trata de deshacerse del comportamiento request – reply, permitiendo que el cliente continue sin tener respuesta



Sockets - Mensajería transiente: No se guarda la información enviada en un storage.

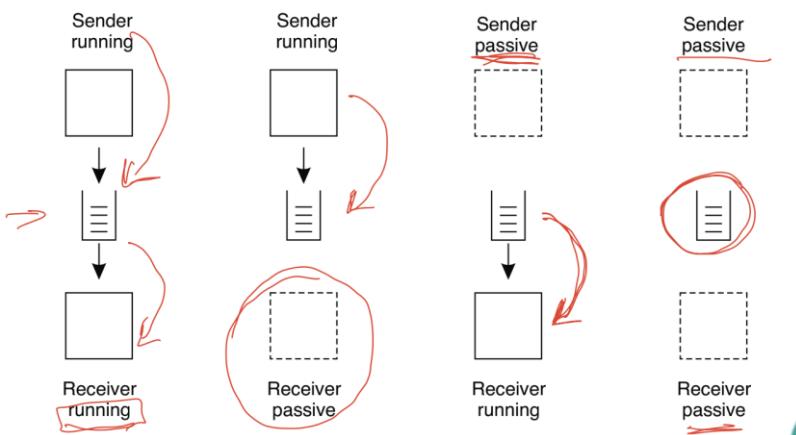


Sockets son de bajo nivel, por lo que pueden haber errores en la programación. La mayoría de las veces se usan en cliente – servidor y se usan de la misma forma. Una alternativa es ZeroMQ, donde se sube el nivel de la comunicación y parea sockets y toda la comunicación es asincrónica.

QUEUE-BASED MESSAGING



Four possible combinations



Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

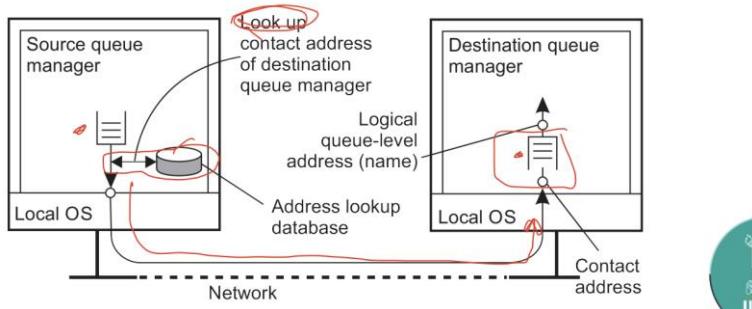
Operations

Operation	Description
PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

Queue managers

- Queues are managed by **queue managers**. An application can put messages only into a **local** queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to **route** messages.

Routing

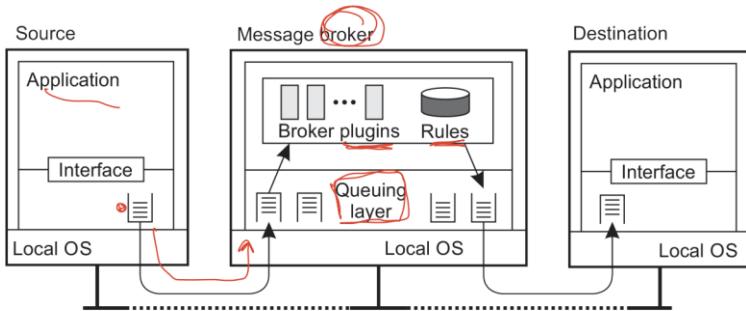


El problema es que si los mensajes están en distinto formato, debe de haber una forma de lidiar con los distintos formatos. Ahí aparece el message bróker.

Observation

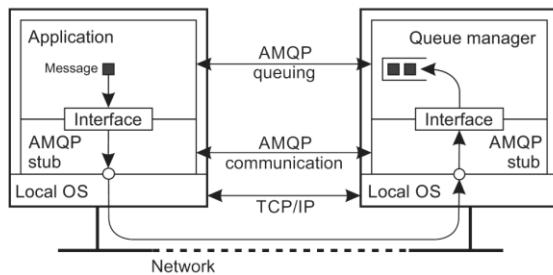
- Message queuing systems assume a **common messaging protocol**: all applications agree on message format (i.e., structure and data representation)
- Broker handles application heterogeneity in an MQ system**
 - Transforms incoming messages to target format
 - Very often acts as an **application gateway**
 - May provide **subject-based** routing capabilities (i.e., **publish-subscribe** capabilities)

De esta cola recibo datos y van a estas colas por lo que debo transformar de esto a esto otro. Una vez transformados envía al servidor de destino



④ Lack of standardization

Advanced Message-Queuing Protocol was intended to play the same role as, for example, TCP in networks: a protocol for high-level messaging with different implementations.



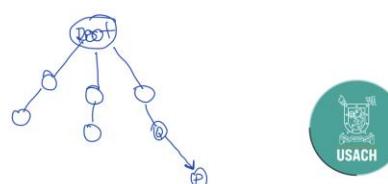
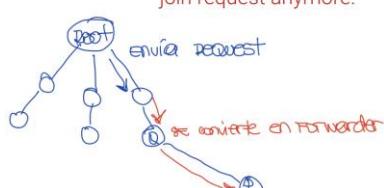
⑤ Basic model

Client sets up a (stable) **connection**, which is a container for several (possibly ephemeral) **one-way channels**. Two one-way channels can form a **session**. A **link** is akin to a socket, and maintains state about message transfers.

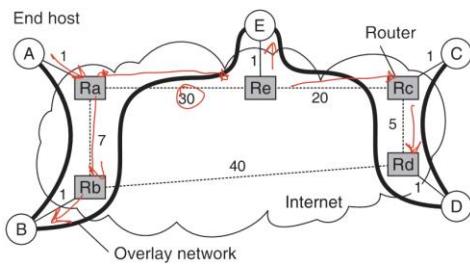
Multicasting: Envío de mensajes a ciertos nodos. La esencia del multicasting es organizar los nodos en un sistema distribuido y usar una red overlay (como se distribuyen de forma lógica y en la red). Una forma es hacer un árbol, lo que implica caminos únicos a los nodos y mallas en red, donde se requiere una forma de rutear los mensajes al destinatario. Para el caso de chord:

Basic approach

- ① Initiator generates a multicast identifier *mid*.
- ② Lookup *succ(mid)*, the node responsible for *mid*.
- ③ Request is routed to *succ(mid)*, which will become the root.
- ④ If *P* wants to join, it sends a *join* request to the root.
- ⑤ When request arrives at *Q*:
 - *Q* has not seen a join request before \Rightarrow it becomes **forwarder**; *P* becomes child of *Q*. *Join request continues to be forwarded*.
 - *Q* knows about tree \Rightarrow *P* becomes child of *Q*. No need to forward join request anymore.



Different metrics



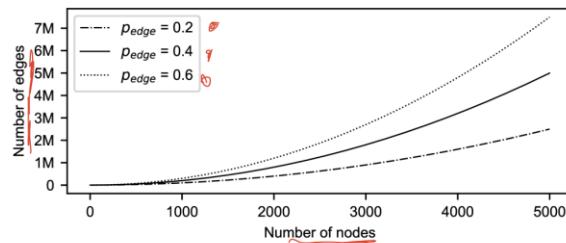
- **Link stress:** How often does an ALM message cross the same physical link? Example: message from A to D needs to cross $\langle Ra, Rb \rangle$ twice.
- **Stretch:** Ratio in delay between ALM-level path and network-level path. Example: messages B to C follow path of length 73 at ALM, but 47 at network level \Rightarrow stretch = 73/47.



Flooding (Peer-to-peer no estructurado):

Essence

P simply sends a message m to each of its neighbors. Each neighbor will forward that message, except to P , and only if it had not seen m before.



Probabilidad de que Dos vértices/nodos de un grafo estan unidos por una linea

Protocolos epidémicos: Una epidemia es local. Básicamente un nodo recibe un mensaje y le manda los nodos a los vecinos. Por ejemplo, en una red peer-to-peer se actualiza un archivo y se quiere actualizar el archivo en todos los otros nodos de la red.

Epidémico

Assume there are no write-write conflicts

- Update operations are performed at a single server
- A replica passes updated state to only a few neighbors
- Update propagation is lazy, i.e., not immediate
- Eventually, each update should reach every replica

Los ocupo cuando ...

Two forms of epidemics

- **Anti-entropy:** Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
- **Rumor spreading:** A replica which has just been updated (i.e., has been contaminated), tells several other replicas about its update (contaminating them as well).



En el exparcimiento del rumor, bob le dice a alice que tiene un rumor, alice se lo dice a chuk y chuk dice que ya sabe el chisme.

ANTI-ENTROPY

Principle operations

- A node P selects another node Q from the system at random.
- **Pull:** P only pulls in new updates from Q
- **Push:** P only pushes its own updates to Q
- **Push-pull:** P and Q send updates to each other

Observation

For push-pull it takes $\mathcal{O}(\log(N))$ rounds to disseminate updates to all N nodes
(**round** = when every node has taken the initiative to start an exchange).

El peor es el pull



Let s denote fraction of nodes that have not yet been updated (i.e., **susceptible**);
 i the fraction of updated (**infected**) and active nodes; and r the fraction of updated nodes that gave up (**removed**).

Cuando se quieren eliminar datos hay un problema: No se puede remover un valor antiguo y esperar que ese borrado se propague, dado que si se usa un algoritmo epidemico el borrado del archivo va a volver a estar. Para ello se utiliza un certificado de muerte



When to remove a death certificate (it is not allowed to stay for ever)

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
- Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)

Note

It is necessary that a removal actually reaches all servers.

Coordinación

Hay procesos se ejecutan antes que otros. La solución es usar UTC que se mide con reloj atómico, este mide los segundos en base a las trancisiones hechas en el átomo de cesio 133.

El UTC se hace entonces tomando el promedio de 50 relojes atómicos alrededor del mundo. Leap second es porque nuestro calendario funciona en base a lo que se llama un día solar, entonces, a medida que va pasando el tiempo, los días se van haciendo más y más corto.

Con esta hora se hace un broadcast a través de satélites de alta frecuencia (baja longitud de onda) y los satélites pueden dar una precisión de +- 0.5 milisegundos.

Si quiero sincronizar dos computadores (o relojes) que están en dos máquinas distintas entonces puedo usar la precisión π , que es el tiempo del computador p – menos el tiempo del computador q, en absoluto, lo cual debe dar menor o igual a π .

Precision

The goal is to keep the deviation between two clocks on any two machines within a specified bound, known as the precision π :

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \underline{\pi}$$

with $C_p(t)$ the computed clock time of machine p at UTC time t .

Accuracy

In the case of accuracy, we aim to keep the clock bound to a value α :

$$\forall t, \forall p : |C_p(t) - t| \leq \underline{\alpha}$$

Synchronization

- ⌚ Internal synchronization: keep clocks precise ~~relatives~~
- ⌚ External synchronization: keep clocks accurate ~~varios relojes~~



Se tiene lo que se llama Clock Drift, este Drift depende de la frecuencia del hardware.

⌚ Clock specifications

- A clock comes specified with its maximum clock drift rate ρ .
- ⌚ $F(t)$ denotes oscillator frequency of the hardware clock at time t
- ⌚ F is the clock's ideal (constant) frequency \Rightarrow living up to specifications:

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

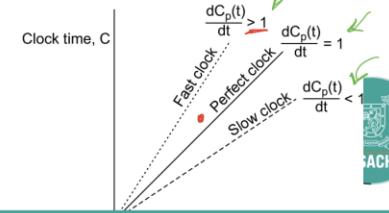
*Si lo
dejino*

Observation

By using hardware interrupts we couple a software clock to the hardware clock, and thus also its clock drift rate:

$$\begin{aligned} \⌚ C_p(t) &= \frac{1}{F} \int_0^t F(t) dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F} \\ \Rightarrow \forall t : 1 - \rho &\leq \frac{dC_p(t)}{dt} \leq 1 + \rho \end{aligned}$$

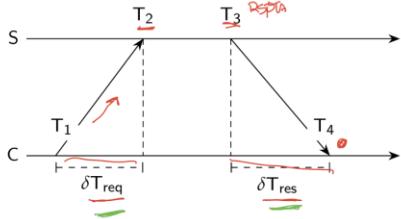
Fast, perfect, slow clocks



Reloj del hardware: Ticks del SO.

Reloj del software: Depende del software.

Getting the current time from a timeserver



Computing the relative offset θ and delay δ

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$ O si fuese perfecto.

$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3)) / 2 - T_4 = ((T_2 - T_1) + (T_3 - T_4)) / 2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2)) / 2$$

proceso refinado de tiempo



REFERENCE BROADCAST SYNCHRONIZATION



Essence

- A node broadcasts a reference message $m \Rightarrow$ each receiving node p records the time $T_{p,m}$ that it received m .
- Note: $T_{p,m}$ is read from p 's local clock.

PROCESADOR CAUTIVO \rightarrow + CLOCK DRIFT.

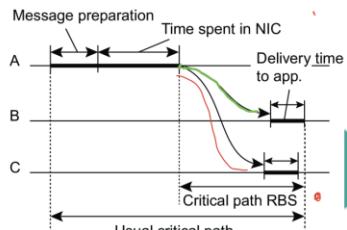
Problem: averaging will not capture drift \Rightarrow use linear regression

PROMEDIO DE t^* , SI HAY CLOCK DRIFT
HABRÍA DE X Y NO HACERLO AL CONTRARIO

NO: $Offset[p,q](t) = \frac{\sum_{k=1}^M (T_{p,k} - T_{q,k})}{M}$

YES: $Offset[p,q](t) = \alpha t + \beta$
REGRESIÓN LINEAL

RBS minimizes critical path



Que pasa si es que hay más que un cliente y servidor. Lo que se hace es sincronización por broadcast por referencia. Un nodo va a hacer un broadcast que referencia un mensaje m , cada nodo p va a grabar el tiempo $T_{p,m}$ que recibió m . Básicamente, alguien manda un mensaje y grabo el tiempo $T_{p,m}$. Note que el $T_{p,m}$ es leído desde el reloj local de p .

Se propone sacar un offset sacando el promedio, donde van a llegar k mensajes. Se restan los tiempos de cada mensaje, con M mensajes. Sin embargo, esto es una mala forma de hacerlo dado que a medida que tengo más mensajes los números que vienen no aportan a la hora de sacar decimales (problemas con parte numérica del promedio). Es por eso que se ofrece hacer una regresión lineal, donde alfa y beta depende de los tiempos.

Lanport menciona que en realidad no importa si es que los procesos estaban de acuerdo en la hora, lo que realmente importa es el orden en el que ocurren los eventos.

The happened-before relation

- If a and b are two events in the same process, and a comes before b , then $a \rightarrow b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Note

- This introduces a partial ordering of events in a system with concurrently operating processes.

sirve para ordenar eventos parciales en un sist. con ...



Problem

How do we maintain a global view of the system's behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event e , satisfying the following properties:

- P1** If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- P2** If a corresponds to sending a message m , and b to the receipt of that message, then also $C(a) < C(b)$.



Each process P_i maintains a local counter C_i and adjusts this counter

- For each new event that takes place within P_i , C_i is incremented by 1.
- Each time a message m is sent by process P_i , the message receives a timestamp $ts(m) = C_i$.
- Whenever a message m is received by a process P_j , P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

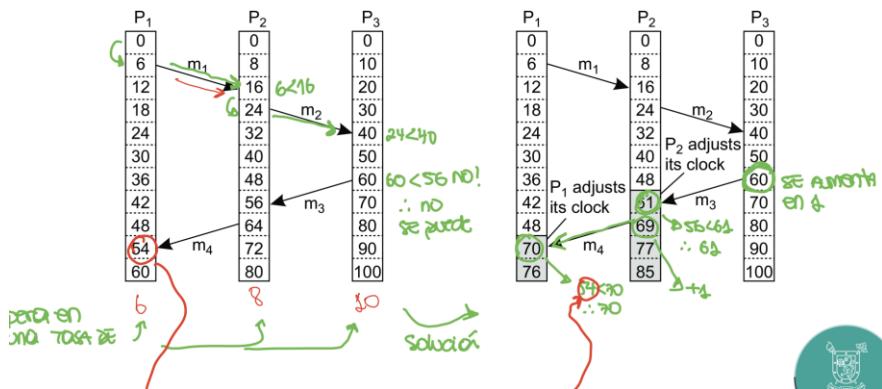
Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

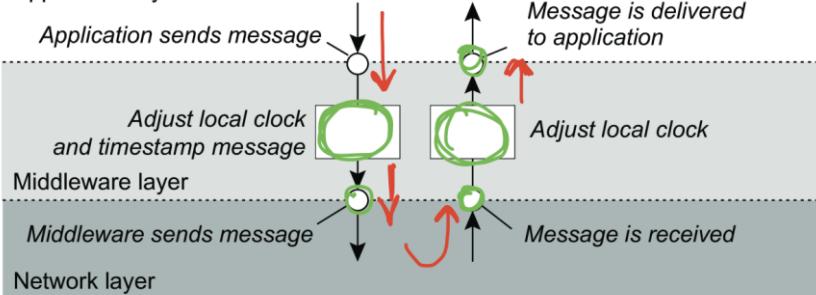




Consider three processes with event counters operating at different rates



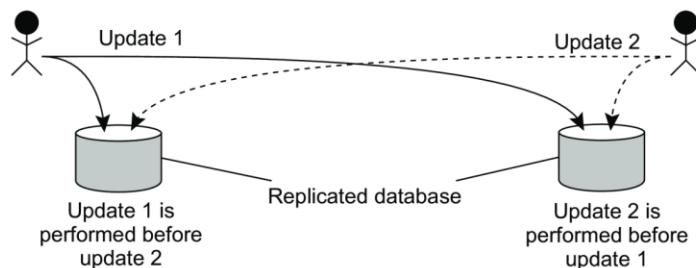
Application layer



Pueden haber problemas cuando se hacen transacciones de forma no ordenada.

Concurrent updates on a replicated database are seen in the same order everywhere

- P₁ adds \$100 to an account (initial value: \$1000)
- P₂ increments account by 1%
- There are two replicas



El proceso pi va a enviar la hora a todos los otros mensajes. El mensaje es colocado en una cola local Cola_i. Cualquier mensaje que llegue a un proceso p_j es encolada con una cola_j de acuerdo a su timestamp y debe ser reconocido por todos los otros procesos.

EXAMPLE: TOTALLY ORDERED MULTICAST

Solution

- Process P_i sends timestamped message m_i to all others. The message itself is put in a local queue $queue_j$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

P_j passes a message m_i to its application if:

- (1) m_i is at the head of $queue_j$
- (2) for each process P_k , there is a message m_k in $queue_j$ with a larger timestamp.

```

1  class Process:
2      def __init__(self, chanID, procID, procIDSet):
3          self.chan.join(procID)
4          self.procID = int(procID)
5          self.otherProcs.remove(self.procID)
6          self.queue = []                      # The request queue
7          self.clock = 0                        # The current logical clock
8
9  ● def requestToEnter(self): PERMIT SI PUEDE ENTRAR A SC.
10     self.clock = self.clock + 1           # Increment clock value
11     self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
12     self.cleanupQ()                     # Sort the queue
13     self.chan.sendTo(self.otherProcs, (self.clock, self.procID, ENTER)) # Send request
14
15     def ackToEnter(self, requester):
16         self.clock = self.clock + 1           # Increment clock value
17         self.chan.sendTo(requester, (self.clock, self.procID, ACK)) # Permit other
18
19     def release(self): PERMITIR SALIR DEL CRITICO
20         tmp = [r for r in self.queue[1:] if r[2] == ENTER] # Remove all ACKs
21         self.queue = tmp                                # and copy to new queue
22         self.clock = self.clock + 1           # Increment clock value
23         self.chan.sendTo(self.otherProcs, (self.clock, self.procID, RELEASE)) # Release
24
25     def allowedToEnter(self): PERMITIR SI PUEDE INGRESAR O NO
26         commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
27         return (self.queue[0][1] == self.procID and len(self.otherProcs) == len(commProcs))
    
```



```

1  def receive(self):
2      msg = self.chan.recvFrom(self.otherProcs)[1]          # Pick up any message
3      self.clock = max(self.clock, msg[0])                 # Adjust clock value...
4      self.clock = self.clock + 1                          # ...and increment
5      if msg[2] == ENTER:
6          self.queue.append(msg)                         # Append an ENTER request
7          self.ackToEnter(msg[1])                       # and unconditionally allow
8      elif msg[2] == ACK:
9          self.queue.append(msg)                         # Append a received ACK
10     elif msg[2] == RELEASE:
11         del(self.queue[0])                           # Just remove first message
12         self.cleanupQ()                            # And sort and cleanup
    
```

Analogy with totally ordered multicast

- With totally ordered multicast, all processes build identical queues, delivering messages in the same order
- Mutual exclusion is about agreeing in which order processes are allowed to enter a critical region

OBSERVATION

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b .

Por lo tanto no se puede concluir que a precede a b .

Definition

We say that b may causally depend on a if $ts(a) < ts(b)$, with:

- for all k , $ts(a)[k] \leq ts(b)[k]$ and
- there exists at least one index k' for which $ts(a)[k'] < ts(b)[k']$

Precedence vs. dependency

- We say that a causally precedes b .
- b **may** causally depend on a , as there may be information from a that is propagated into b .

Solution: each P_i maintains a vector VC_i

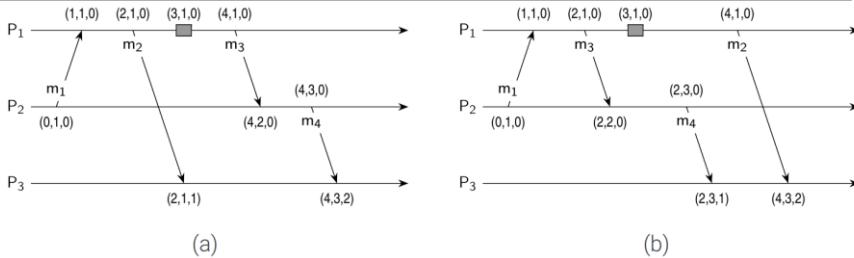
- $VC_i[i]$ is the local logical clock at process P_i .
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j .

Maintaining vector clocks

1. Before executing an event, P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j , it sets m 's (vector) timestamp $ts(m)$ equal to VC_i after having executed step 1.
3. Upon the receipt of a message m , process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k , after which it executes step 1 and then delivers the message to the application.



Capturing potential causality when exchanging messages



Analysis

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusion
(a)	(2,1,0)	(4,3,0)	Yes	No	m_2 may causally precede m_4
(b)	(4,1,0)	(2,3,0)	No	No	m_2 and m_4 may conflict

Cuando hay conflictos es el middleware el que tiene que ver como soluciona ese problema. Podemos asegurar que un mensaje sea enviado si todos los mensajes precedentes han sido enviados.

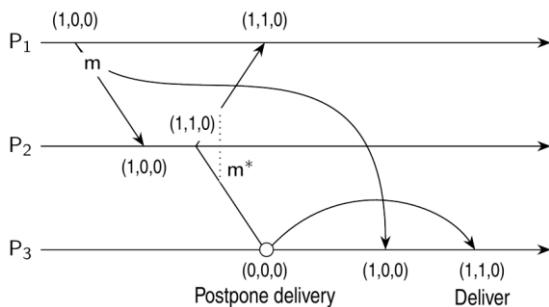
Adjustment

P_i increments $VC_i[i]$ only when sending a message, and P_j "adjusts" VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

P_j postpones delivery of m until:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

Enforcing causal communication



Problem

Several processes in a distributed system want exclusive access to some resource.

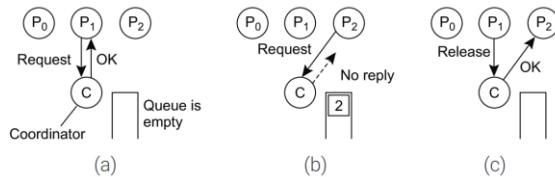
Basic solutions

Permission-based: A process wanting to enter its critical region, or access a resource, needs permission from other processes.

Token-based: A token is passed between processes. The one who has the token may proceed in its critical region, or pass it on when not interested.

PERMISSION-BASED, CENTRALIZED

Simply use a coordinator



- (a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.
- (b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.
- (c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2 .



MUTUAL EXCLUSION: RICART & AGRAWALA

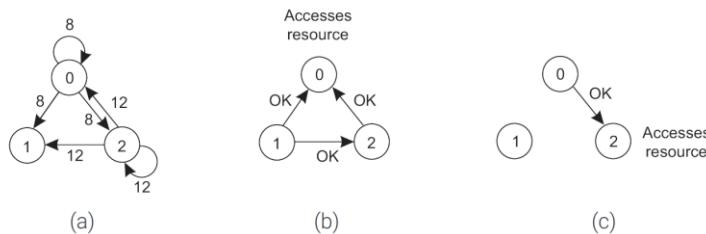
The same as Lamport except that acknowledgments are not sent

Return a response to a request only when:

- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is **deferred**, implying some more local administration.

Example with three processes



- (a) Two processes want to access a shared resource at the same moment.
- (b) P_0 has the lowest timestamp, so it wins.
- (c) When process P_0 is done, it sends an OK also, so P_2 can now go ahead



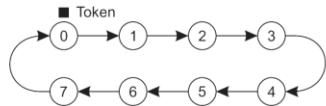
Tiene N puntos de fallo, dado que si es que algún proceso crashea, no podrá responder request.

MUTUAL EXCLUSION: TOKEN RING ALGORITHM

Essence

Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).

An overlay network constructed as a logical ring with a circulating token



Exclusion mutua descentralizada:

Principle

Assume every resource is replicated N times, with each replica having its own coordinator \Rightarrow access requires a majority vote from $m > N/2$ coordinators. A coordinator always responds immediately to a request.

Assumption

When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

How robust is this system?

- Let $p = \Delta t/T$ be the probability that a coordinator resets during a time interval Δt , while having a lifetime of T .
- The probability $\mathbb{P}[k]$ that k out of m coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- f coordinators reset \Rightarrow correctness is violated when there is only a minority of nonfaulty coordinators: when $N - (m - f) \geq m$, or, $f \geq 2m - N$.
- The probability of a violation is $\sum_{k=2m-N}^m \mathbb{P}[k]$.

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2(N - 1)$	$2(N - 1)$
Token ring	$1, \dots, \infty$	$0, \dots, N - 1$
Decentralized	$2kN + (k - 1)N/2 + N, k = 1, 2, \dots$	$2kN + (k - 1)N/2$

ELECTION BY BULLYING

Principle

Consider N processes $\{P_0, \dots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \dots, P_{N-1}$.
2. If no one responds, P_k wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and P_k 's job is done.

ELECTION IN A RING

Principle

Process priority is obtained by organizing processes into a (logical) ring. The process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

ELECTIONS BY PROOF OF WORK

Basics

- Consider a potentially large group of processes
- Each process is required to solve a computational puzzle
- When a process solves the puzzle, it broadcasts its victory to the group
- We assume there is a conflict resolution procedure when more than one process claims victory

Solving a computational puzzle

- Make use of a **secure hashing function** $H(m)$:
 - m is some data; $H(m)$ returns a **fixed-length bit string**
 - computing $h = H(m)$ is computationally efficient
 - finding a function H^{-1} such that $m = H^{-1}(H(m))$ is computationally extremely difficult
- Practice: finding H^{-1} boils down to an extensive trial-and-error procedure



Controlled race

- Assume a globally known secure hash function H^* . Let H_i be the hash function used by process P_i .
- Task: given a bit string $h = H_i(m)$, find a bit string \tilde{h} such that $h^* = H^*(H_i(\tilde{h} \odot h))$ where:
 - h^* is a bit string with K leading zeroes
 - $\tilde{h} \odot h$ denotes some predetermined bitwise operation on \tilde{h} and h

Observation

By controlling K , we control the difficulty of finding \tilde{h} . If p is the probability that a random guess for \tilde{h} will suffice: $p = (1/2)^K$.

Current practice

In many PoW-based blockchain systems, $K = 64$

- With $K = 64$, it takes about 10 minutes on a supercomputer to find \tilde{h}