

# **Distributed Systems**

(4th edition, version 01)

## **Chapter 05: Coordination**

# Physical clocks

## Problem

Sometimes we simply need the exact time, not just an ordering.

## Solution: Universal Coordinated Time (UTC)

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

## Note

UTC is **broadcast** through short-wave radio and satellite. Satellites can give an accuracy of about  $\pm 0.5$  ms.

# Clock synchronization

## Precision

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

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

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  $\alpha$ :

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

## Synchronization

- **Internal synchronization**: keep clocks **precise**
- **External synchronization**: keep clocks **accurate**

# Clock drift

## Clock specifications

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

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

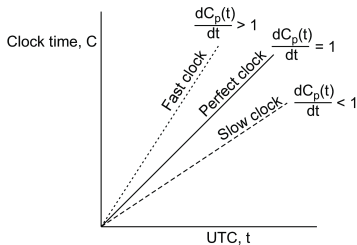
## Observation

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

$$C_p(t) = \frac{1}{F} \int_0^t F(t) dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

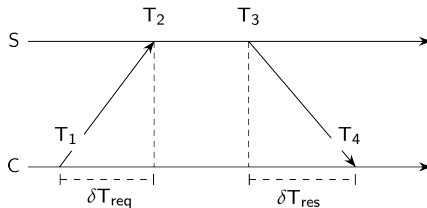
$$\Rightarrow \forall t: 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

## Fast, perfect, slow clocks



# Detecting and adjusting incorrect times

## Getting the current time from a timeserver



## Computing the relative offset $\theta$ and delay $\delta$

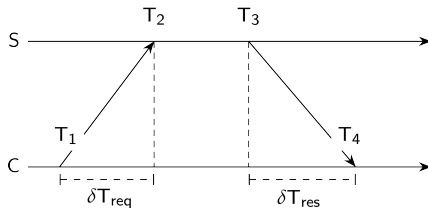
**Assumption:**  $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

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

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

# Detecting and adjusting incorrect times

## Getting the current time from a timeserver



## Computing the relative offset $\theta$ and delay $\delta$

**Assumption:**  $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

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

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

## Network Time Protocol

Collect  $(\theta, \delta)$  pairs. Choose  $\theta$  for which associated delay  $\delta$  was minimal.

# Reference broadcast synchronization

## Essence

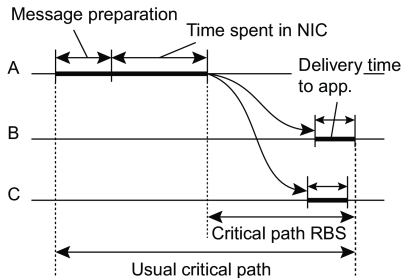
- A node broadcasts a reference message  $m \Rightarrow$  each receiving node  $p$  records the time  $T_{p,m}$  that it received  $m$ .
- **Note:**  $T_{p,m}$  is read from  $p$ 's local clock.

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

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

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

RBS minimizes critical path



# The Happened-before relationship

## Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the **order in which events occur**. Requires a notion of **ordering**.



# The Happened-before relationship

## Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the **order in which events occur**. Requires a notion of ordering.

## 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.

# Logical clocks

## Problem

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

# Logical clocks

## 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  $C(a) < C(b)$ .

# Logical clocks

## 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  $C(a) < C(b)$ .

## Problem

How to attach a timestamp to an event when there's no global clock  $\Rightarrow$  maintain a **consistent** set of logical clocks, one per process.

## Logical clocks: solution

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

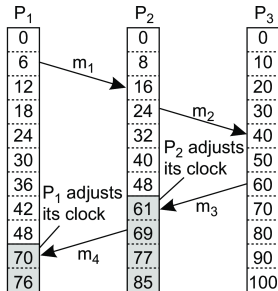
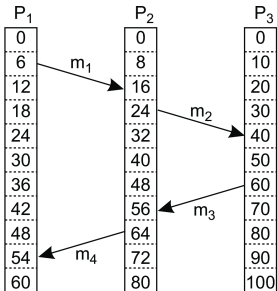
1. For each new event that takes place within  $P_i$ ,  $C_i$  is incremented by 1.
2. Each time a message  $m$  is sent by process  $P_i$ , the message receives a timestamp  $ts(m) = C_i$ .
3. Whenever a message  $m$  is received by a process  $P_j$ ,  $P_j$  adjusts its local counter  $C_j$  to  $\max\{C_j,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

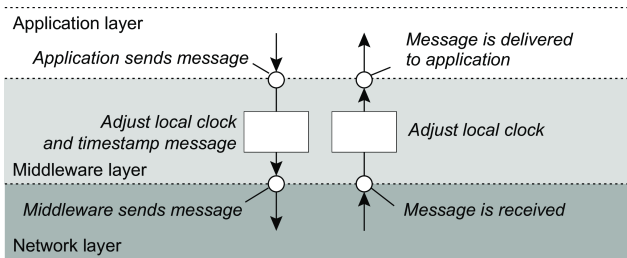
## Logical clocks: example

Consider three processes with **event counters** operating at different rates



# Logical clocks: where implemented

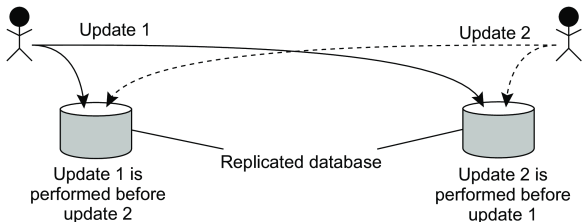
## Adjustments implemented in middleware



## Example: Totally ordered multicast

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

- $P_1$  adds \$100 to an account (initial value: \$1000)
- $P_2$  increments account by 1%
- There are two replicas



### Result

In absence of proper synchronization:

replica #1  $\leftarrow$  \$1111, while replica #2  $\leftarrow$  \$1110.



## 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  $queue_i$ .
- Any incoming message at  $P_j$  is queued in  $queue_j$ , **according to its timestamp**, and **acknowledged** to every other process.

## 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  $queue_i$ .
- 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.

## 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  $queue_i$ .
- 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.

### Note

We are assuming that communication is **reliable** and **FIFO ordered**.

# Lamport's clocks for mutual exclusion

```
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):
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):
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):
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))
```

## Lamport's clocks for mutual exclusion

```
1  def receive(self):
2      msg = self.chan.recvFrom(self.otherProcs)[1]
3      self.clock = max(self.clock, msg[0])
4      self.clock = self.clock + 1
5      if msg[2] == ENTER:
6          self.queue.append(msg)
7          self.ackToEnter(msg[1])
8      elif msg[2] == ACK:
9          self.queue.append(msg)
10     elif msg[2] == RELEASE:
11         del(self.queue[0])
12     self.cleanupQ()
```

*# Pick up any message*  
*# Adjust clock value...*  
*# ...and increment*

*# Append an ENTER request*  
*# and unconditionally allow*

*# Append a received ACK*

*# Just remove first message*  
*# And sort and cleanup*

# Lamport's clocks for mutual exclusion

## 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

# Vector clocks

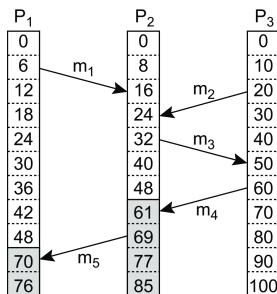
## Observation

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

## Concurrent message transmission using logical clocks

## Observation

Event  $a$ :  $m_1$  is received at  $T = 16$ ;  
Event  $b$ :  $m_2$  is sent at  $T = 20$ .

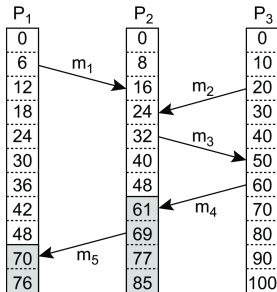


## Vector clocks

### Observation

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

### Concurrent message transmission using logical clocks



### Observation

Event  $a$ :  $m_1$  is received at  $T = 16$ ;  
 Event  $b$ :  $m_2$  is sent at  $T = 20$ .

### Note

We **cannot** conclude that  $a$  causally precedes  $b$ .



# Causal dependency

## 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  $b$ .

## Capturing potential causality

Solution: each  $P_i$  maintains a vector  $VC_i$

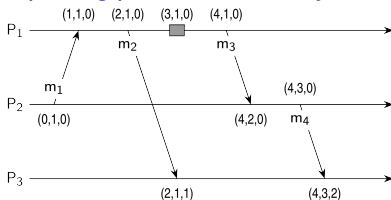
- $VC_i[i]$  is the local logical clock at process  $P_i$ .
- If  $VC_i[j] = k$  then  $P_i$  knows that  $k$  events have occurred at  $P_j$ .

### Maintaining vector clocks

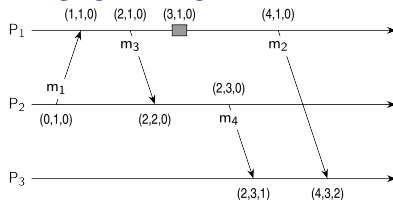
1. Before executing an event,  $P_i$  executes  $VC_i[i] \leftarrow VC_i[i] + 1$ .
2. When process  $P_i$  sends a message  $m$  to  $P_j$ , it sets  $m$ 's (vector) timestamp  $ts(m)$  equal to  $VC_i$  after having executed step 1.
3. Upon the receipt of a message  $m$ , process  $P_j$  sets  $VC_j[k]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

# Vector clocks: Example

## Capturing potential causality when exchanging messages



(a)



(b)

## 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

## Causally ordered multicasting

### Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

### 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  $VC_j[j]$ ).

# Causally ordered multicasting

## Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

## Adjustment

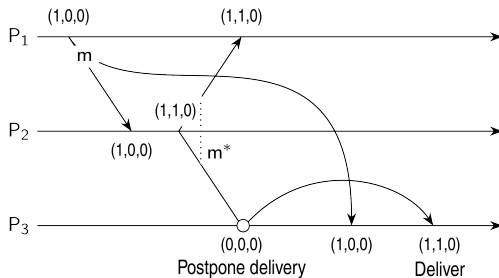
$P_i$  increments  $VC_i[i]$  only when sending a message, and  $P_j$  “adjusts”  $VC_j$  when receiving a message (i.e., effectively does not change  $VC_j[j]$ ).

$P_j$  postpones delivery of  $m$  until:

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

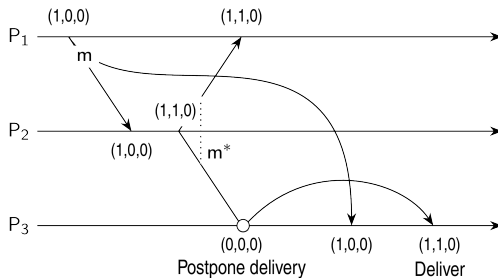
# Causally ordered multicasting

## Enforcing causal communication



# Causally ordered multicasting

## Enforcing causal communication



### Example

Take  $VC_3 = [0, 2, 2]$ ,  $ts(m) = [1, 3, 0]$  from  $P_1$ . What information does  $P_3$  have, and what will it do when receiving  $m$  (from  $P_1$ )?

# Mutual exclusion

## Problem

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

## Basic solutions

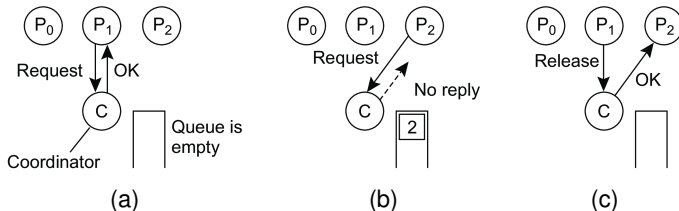
**Permission-based:** A process wanting to enter its critical region, or access a resource, needs permission from other processes.

**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

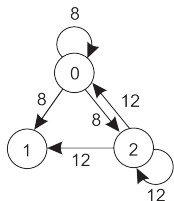
Return a response to a request only when:

- 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).

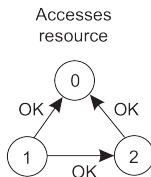
In all other cases, reply is **deferred**, implying some more local administration.

# Mutual exclusion: Ricart & Agrawala

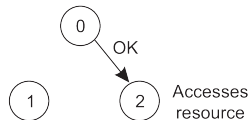
## Example with three processes



(a)



(b)



(c)

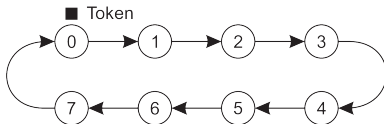
- (a) Two processes want to access a shared resource at the same moment.
- (b)  $P_0$  has the lowest timestamp, so it wins.
- (c) When process  $P_0$  is done, it sends an OK also, so  $P_2$  can now go ahead.

## 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



# Decentralized mutual exclusion

## 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.

# Decentralized mutual exclusion

## How robust is this system?

- Let  $p = \Delta t / T$  be the probability that a coordinator resets during a time interval  $\Delta t$ , while having a lifetime of  $T$ .

# Decentralized mutual exclusion

## How robust is this system?

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

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

# Decentralized mutual exclusion

## How robust is this system?

- Let  $p = \Delta t / T$  be the probability that a coordinator resets during a time interval  $\Delta t$ , 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,  $f \geq 2m - N$ .



# Decentralized mutual exclusion

## How robust is this system?

- Let  $p = \Delta t / T$  be the probability that a coordinator resets during a time interval  $\Delta t$ , 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  $\sum_{k=2m-N}^m \mathbb{P}[k]$ .

# Decentralized mutual exclusion

## Violation probabilities for various parameter values

<b>N</b>	<b>m</b>	<b>p</b>	<b>Violation</b>
8	5	3 sec/hour	$< 10^{-5}$
8	6	3 sec/hour	$< 10^{-11}$
16	9	3 sec/hour	$< 10^{-4}$
16	12	3 sec/hour	$< 10^{-21}$
32	17	3 sec/hour	$< 10^{-4}$
32	24	3 sec/hour	$< 10^{-43}$

<b>N</b>	<b>m</b>	<b>p</b>	<b>Violation</b>
8	5	30 sec/hour	$< 10^{-3}$
8	6	30 sec/hour	$< 10^{-7}$
16	9	30 sec/hour	$< 10^{-2}$
16	12	30 sec/hour	$< 10^{-13}$
32	17	30 sec/hour	$< 10^{-2}$
32	24	30 sec/hour	$< 10^{-27}$

# Decentralized mutual exclusion

## Violation probabilities for various parameter values

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-5}$
8	6	3 sec/hour	$< 10^{-11}$
16	9	3 sec/hour	$< 10^{-4}$
16	12	3 sec/hour	$< 10^{-21}$
32	17	3 sec/hour	$< 10^{-4}$
32	24	3 sec/hour	$< 10^{-43}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-3}$
8	6	30 sec/hour	$< 10^{-7}$
16	9	30 sec/hour	$< 10^{-2}$
16	12	30 sec/hour	$< 10^{-13}$
32	17	30 sec/hour	$< 10^{-2}$
32	24	30 sec/hour	$< 10^{-27}$

So....

What can we conclude?

## Mutual exclusion: comparison

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	$2kN + (k - 1)N/2 + N, k = 1, 2, \dots$	$2kN + (k - 1)N/2$

## Example: ZooKeeper

### Basics (and keeping it simple)

- Centralized server setup
- All client-server communication is **nonblocking**: a client immediately gets a response
- ZooKeeper maintains a **tree-based namespace**, akin to that of a filesystem
- Clients can **create**, **delete**, or **update** nodes, as well as **check existence**.

## ZooKeeper race condition

### Note

ZooKeeper allows a client to be **notified** when a node, or a branch in the tree, changes. This may easily lead to **race conditions**.

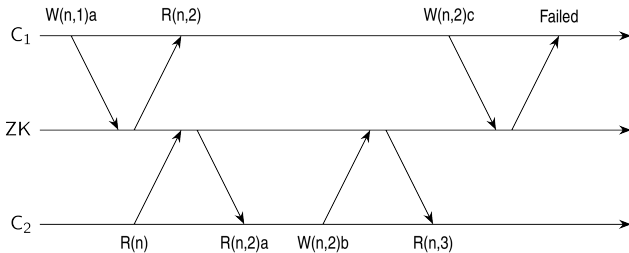
### Consider a simple locking mechanism

1. A client  $C_1$  creates a node `/lock`.
2. A client  $C_2$  wants to acquire the lock but is notified that the associated node already exists.
3. Before  $C_2$  subscribes to a notification,  $C_1$  releases the lock, i.e., deletes `/lock`.
4. Client  $C_2$  subscribes to changes to `/lock` and blocks locally.

### Solution

Use version numbers

## ZooKeeper versioning



### Notations

- $W(n,k)a$ : request to write  $a$  to node  $n$ , assuming current version is  $k$ .
- $R(n,k)$ : current version of node  $n$  is  $k$ .
- $R(n)$ : client wants to know the current value of node  $n$
- $R(n,k)a$ : value  $a$  from node  $n$  is returned with its current version  $k$ .

# ZooKeeper locking protocol

It is now very simple

1. **lock**: A client  $C_1$  creates a node `/lock`.
2. **lock**: A client  $C_2$  wants to acquire the lock but is notified that the associated node already exists  $\Rightarrow C_2$  subscribes to notification on changes of `/lock`.
3. **unlock**: Client  $C_1$  deletes node `/lock`  $\Rightarrow$  all subscribers to changes are notified.



# Election algorithms

## Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

## Note

In many systems, the coordinator is chosen manually (e.g., file servers). This leads to centralized solutions  $\Rightarrow$  single point of failure.

# Election algorithms

## Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process **dynamically**.

## Note

In many systems, the coordinator is chosen manually (e.g., file servers). This leads to centralized solutions  $\Rightarrow$  single point of failure.

## Teasers

1. If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?
2. Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

## Basic assumptions

- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

# Election by bullying

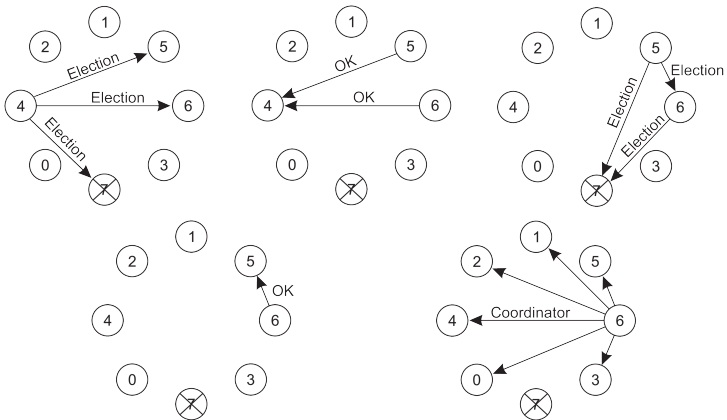
## Principle

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 by bullying

## The bully election algorithm



# Election in a ring

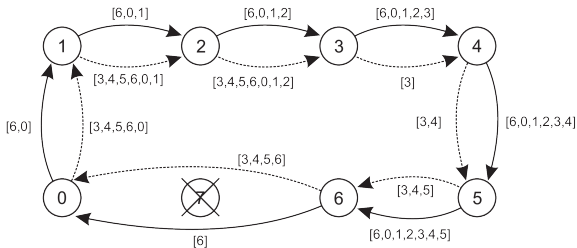
## Principle

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.

# Election in a ring

## Election algorithm using a ring



- The solid line shows the election messages initiated by  $P_6$
- The dashed one, the messages by  $P_3$

## Example: Leader election in ZooKeeper server group

### Basics

- Each server  $s$  in the server group has an identifier  $id(s)$
- Each server has a monotonically increasing counter  $tx(s)$  of the latest transaction it handled (i.e., series of operations on the namespace).
- When follower  $s$  suspects leader crashed, it broadcasts an *ELECTION* message, along with the pair  $(voteID, voteTX)$ . Initially,
  - $voteID \leftarrow id(s)$
  - $voteTX \leftarrow tx(s)$
- Each server  $s$  maintains two variables:
  - $leader(s)$ : records the server that  $s$  believes may be final leader. Initially,  $leader(s) \leftarrow id(s)$ .
  - $lastTX(s)$ : what  $s$  knows to be the most recent transaction. Initially,  $lastTX(s) \leftarrow tx(s)$ .



## Example: Leader election in ZooKeeper server group

When  $s^*$  receives  $(voteID, voteTX)$

- If  $lastTX(s^*) < voteTX$ , then  $s^*$  just received more up-to-date information on the most recent transaction, and sets
  - $leader(s^*) \leftarrow voteID$
  - $lastTX(s^*) \leftarrow voteTX$
- If  $lastTX(s^*) = voteTX$  and  $leader(s^*) < voteID$ , then  $s^*$  knows as much about the most recent transaction as what it was just sent, but its perspective on which server will be the next leader needs to be updated:
  - $leader(s^*) \leftarrow voteID$

### Note

When  $s^*$  believes it should be the leader, it broadcasts  $\langle id(s^*), tx(s^*) \rangle$ .

Essentially, we're bullying.

# Example: Leader election in Raft

## Basics

- We have a (relatively small) group of servers
- A server is in one of three states: *follower*, *candidate*, or *leader*
- The protocol works in *terms*, starting with term 0
- Each server starts in the *follower* state.
- A leader is to regularly broadcast messages (perhaps just a simple *heartbeat*)

## Example: Leader election in Raft

### Selecting a new leader

When follower  $s^*$  hasn't received anything from the alleged leader  $s$  for some time,  $s^*$  broadcasts that it volunteers to be the next leader, increasing the term by 1.  $s^*$  enters the **candidate** state. Then:

- If leader  $s$  receives the message, it responds by acknowledging that it is still the leader.  $s^*$  returns to the **follower** state.
- If another follower  $s^{**}$  gets the election message from  $s^*$ , and it is the first election message during the current term,  $s^{**}$  votes for  $s^*$ . Otherwise, it simply ignores the election message from  $s^*$ . When  $s^*$  has collected a majority of votes, a new term starts with a new leader.

## Example: Leader election in Raft

### Selecting a new leader

When follower  $s^*$  hasn't received anything from the alleged leader  $s$  for some time,  $s^*$  broadcasts that it volunteers to be the next leader, increasing the term by 1.  $s^*$  enters the **candidate** state. Then:

- If leader  $s$  receives the message, it responds by acknowledging that it is still the leader.  $s^*$  returns to the **follower** state.
- If another follower  $s^{**}$  gets the election message from  $s^*$ , and it is the first election message during the current term,  $s^{**}$  votes for  $s^*$ . Otherwise, it simply ignores the election message from  $s^*$ . When  $s^*$  has collected a majority of votes, a new term starts with a new leader.

### Observation

By slightly differing the timeout values per follower for deciding when to start an election, we can avoid concurrent elections, and the election will rapidly converge.

# 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

# Elections by proof of work

## 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  $h$

# Elections by proof of work

## 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:  $p = (1/2)^K$ .

# Elections by proof of work

## 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  $\tilde{h}$
- With  $K = 64$ , it takes about 100 years on a laptop to find  $\tilde{h}$



# Elections by proof of stake

## Basics

We assume a blockchain system in which  $N$  **secure tokens** are used:

- Each token has a unique **owner**
- Each token has a uniquely associated **index**  $1 \leq k \leq N$
- A token cannot be modified or copied without this going unnoticed

## Principle

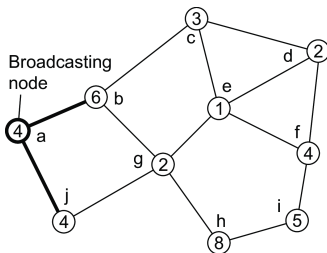
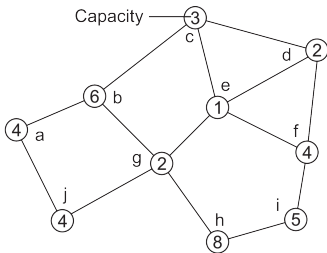
- Draw a random number  $k \in \{1, \dots, N\}$
- Look up the process  $P$  that owns the token with index  $k$ .  $P$  is the next leader.

## Observation

The more tokens a process owns, the higher the probability it will be selected as leader.

# A solution for wireless networks

## A sample network

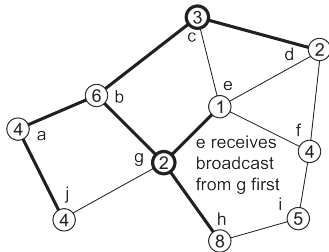
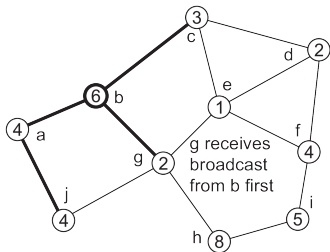


## Essence

Find the node with the highest capacity to select as the next leader.

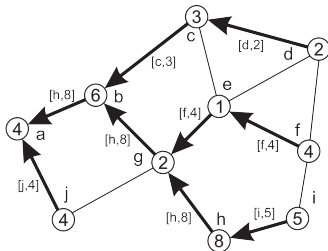
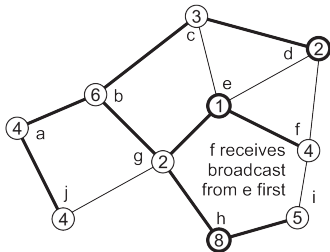
# A solution for wireless networks

## A sample network



# A solution for wireless networks

## A sample network



## Essence

A node reports back only the node that it found to have the highest capacity.

# Gossip-based coordination: aggregation

## Typical apps

- **Data dissemination**: Perhaps the most important one. Note that there are many variants of dissemination.
- **Aggregation**: Let every node  $P_i$  maintain a variable  $v_i$ . When two nodes gossip, they each reset their variable to

$$v_i, v_j \leftarrow (v_i + v_j)/2$$

Result: in the end each node will have computed the average  $\bar{v} = \sum_i v_i / N$ .

# Gossip-based coordination: aggregation

## Typical apps

- **Data dissemination**: Perhaps the most important one. Note that there are many variants of dissemination.
- **Aggregation**: Let every node  $P_i$  maintain a variable  $v_i$ . When two nodes gossip, they each reset their variable to

$$v_i, v_j \leftarrow (v_i + v_j)/2$$

Result: in the end each node will have computed the average  $\bar{v} = \sum_i v_i / N$ .

- What happens in the case that initially  $v_i = 1$  and  $v_j = 0, j \neq i$ ?

# Gossip-based coordination: peer sampling

## Problem

For many gossip-based applications, you need to **select a peer uniformly at random** from the entire network. In principle, this means you need to know all other peers. **Impossible?**

## Basics

- Each node maintains a list of  $c$  references to other nodes
- **Regularly**, pick another node at random (from the list), and **exchange** roughly  $c/2$  references
- When the **application** needs to select a node at random, it also picks a random one from its local list.

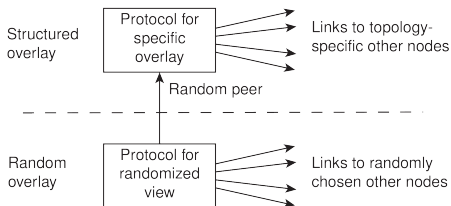
## Observation

Statistically, it turns out that the selection of a peer from the local list is indistinguishable from selecting uniformly at random peer from the entire network

# Gossip-based overlay construction

## Essence

Maintain two local lists of neighbors. The lowest is used for providing a **peer-sampling service**; the highest list is used to carefully select **application-dependent neighbors**.

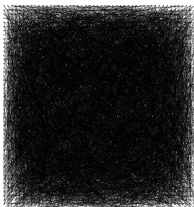




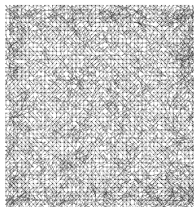
## Gossip-based overlay construction: a 2D torus

Consider a logical  $N \times N$  grid, with a node on each point of the grid.

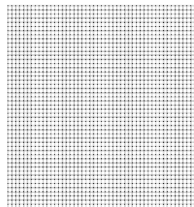
- Every node must maintain a list of  $c$  nearest neighbors
- Distance between node at  $(a_1, a_2)$  and  $(b_1, b_2)$  is  $d_1 + d_2$ , with  $d_i = \min(N - |a_i - b_i|, |a_i - b_i|)$
- Every node picks a random other node from its lowest-level list, and keeps only the closest one in its top-level list.
- Once every node has picked and selected a random node, we move to the next round



start ( $N = 50$ )



after 5 rounds



after 20 rounds

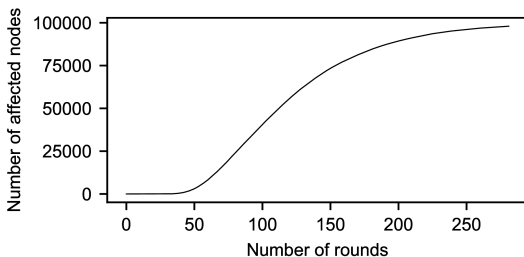
## A gossip-based 2D torus in Python (outline)

```
1 def maintainViews():
2     for viewType in [viewOverlay, viewPSS]: # For each view, do the same
3         peer[viewType] = None
4         if time to maintain viewType: # This viewType needs to be updated
5             peer[viewType] = selectPeer(viewType) # Select a peer
6             links = selectLinks(viewType, peer[viewType]) # Select links
7             sendTo(peer[viewType], Request[viewType], links) # Send links asynchronously
8
9     while True:
10        block = (peer[viewOverlay] != None) or (peer[viewPSS] != None)
11        sender, msgType, msgData = rcvFromAny(block) # Block if expecting something
12
13        if msg == None: # All work has been done, simply return from the call
14            return
15
16        for viewType in [viewOverlay, viewPSS]: # For each view, do the same
17            if msgType == Response[viewType]: # Response to previously sent links
18                updateOwnView(viewType, msgData) # Just update the own view
19
20            elif msgType == Request[viewType]: # Request for exchanging links
21                if peer[viewType] == None: # No outstanding exchange request
22                    links = selectLinks(viewType, sender) # Select links
23                    sendTo(sender, Response[viewType], links) # Send them asynchronously
24                    updateOwnView(viewType, msgData) # Update own view
25                else: # This node already has a pending exchange request, ignore this one
26                    sendTo(sender, IgnoreRequest[viewType])
27
28            elif msgType == IgnoreRequest[viewType]: # Request has been denied, give up
29                peer[viewType] = None
```

# Secure gossiping

## Dramatic attack

Consider when exchanging references, a set of **colluding nodes** systematically returns links only to each other  $\Rightarrow$  we are dealing with **hub attack**.

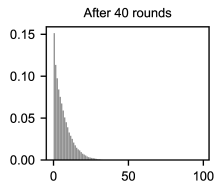
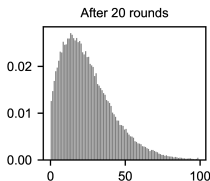
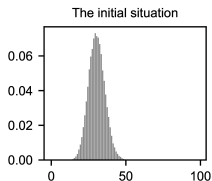


## Situation

A network with 100,000 nodes, a local list size  $c = 30$ , and only 30 attackers. The y-axis shows the number of nodes with links **only** to the attackers. After less than 300 rounds, the attackers have full control.

## A solution: gathering statistics

This is what measuring indegree distributions tells us: which fraction of nodes (y-axis) have how many other nodes pointing to them (x-axis)?

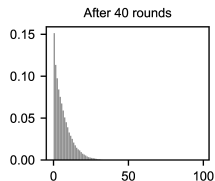
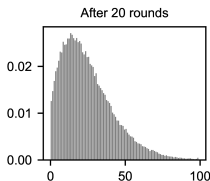
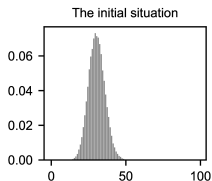


### Basic approach

When a benign node initiates an exchange, it may either use the result for gathering statistics, or for updating its local list. An attacker is in limbo: will its response be used for statistical purposes or for functional purposes?

## A solution: gathering statistics

This is what measuring indegree distributions tells us: which fraction of nodes (y-axis) have how many other nodes pointing to them (x-axis)?



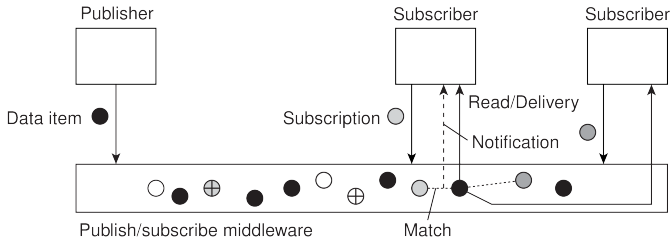
### Basic approach

When a benign node initiates an exchange, it may either use the result for gathering statistics, or for updating its local list. An attacker is in limbo: will its response be used for statistical purposes or for functional purposes?

### Observation

When gathering statistics may reveal colluders, a colluding node will be **forced** to behave according to the protocol.

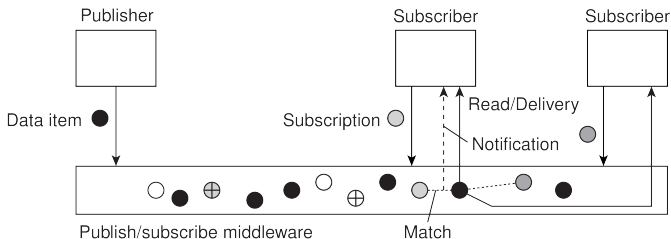
# Distributed event matching



## Principle

- A process specifies in which events it is interested (**subscription**  $S$ )
- When a process **publishes a notification**  $N$  we need to see whether  $S$  **matches**  $N$ .

# Distributed event matching



## Principle

- A process specifies in which events it is interested (**subscription**  $S$ )
- When a process **publishes a notification**  $N$  we need to see whether  $S$  **matches**  $N$ .

## Hard part

Implementing the **match** function in a scalable manner.

# General approach

## What is needed

- $sub2node(S)$ : map a subscription  $S$  to a nonempty subset  $\mathbf{S}$  of servers
- $not2node(N)$ : map a notification  $N$  to a nonempty subset  $\mathbf{N}$  of servers

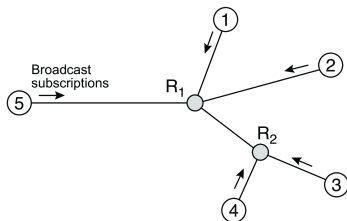
Make sure that  $\mathbf{S} \cap \mathbf{N} \neq \emptyset$ .

## Observations

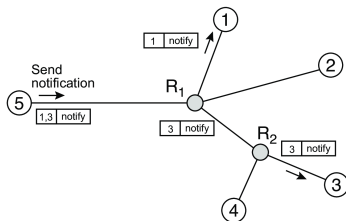
- Centralized solution is simple:  $\mathbf{S} = \mathbf{N} = \{s\}$ , i.e. a single server.
- Topic-based publish-subscribe is also simple: each  $S$  and  $N$  is tagged with a single topic; each topic is handled by a single server (a rendezvous node). Several topics may be handled by same server).
- Content-based publish-subscribe is tough: a subscription takes the form (*attribute*, *value*) pair, with example values:
  - range: " $1 \leq x < 10$ "
  - containment: " $x \in \{red, blue\}$ "
  - prefix and suffix expressions: "`url.startsWith("https")`"



# Selective routing



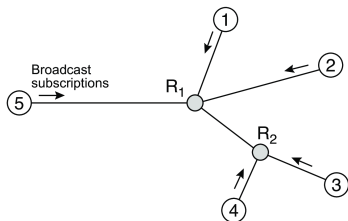
(a)



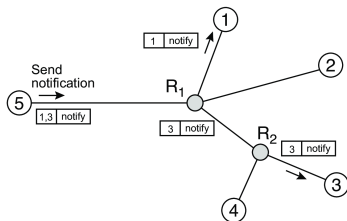
(b)

- (a) first broadcast subscriptions
- (b) forward notifications only to relevant rendezvous nodes

## Selective routing



(a)



(b)

- (a) first broadcast subscriptions
- (b) forward notifications only to relevant rendezvous nodes

### Example of a (partially filled) routing table

Interface	Filter
To node 3	$a \in [0, 3]$
To node 4	$a \in [2, 5]$
Toward router $R_1$	(unspecified)

# Gossiping: Sub-2-Sub

## Basics

- **Goal:** To realize scalability, make sure that subscribers with the same interests form just a single group
- **Model:** There are  $N$  attributes  $a_1, \dots, a_N$ . An attribute value is always (mappable to) a floating-point number.
- **Subscription:** Takes forms such as  $S = \langle a_1 \rightarrow 3.0, a_4 \rightarrow [0.0, 0.5] \rangle$ :  $a_1$  should be 3.0;  $a_4$  should lie between 0.0 and 0.5; other attribute values don't matter.

## Observations

- A subscription  $S_i$  specifies a subset  $\mathbf{S}_i$  in a  $N$ -dimensional space.
- We are interested only in notifications that fall into  $\bar{\mathbf{S}} = \cup \mathbf{S}_i$ .

# Gossiping: Sub-2-Sub

## Basics

- **Goal:** To realize scalability, make sure that subscribers with the same interests form just a single group
- **Model:** There are  $N$  attributes  $a_1, \dots, a_N$ . An attribute value is always (mappable to) a floating-point number.
- **Subscription:** Takes forms such as  $S = \langle a_1 \rightarrow 3.0, a_4 \rightarrow [0.0, 0.5] \rangle$ :  $a_1$  should be 3.0;  $a_4$  should lie between 0.0 and 0.5; other attribute values don't matter.

## Observations

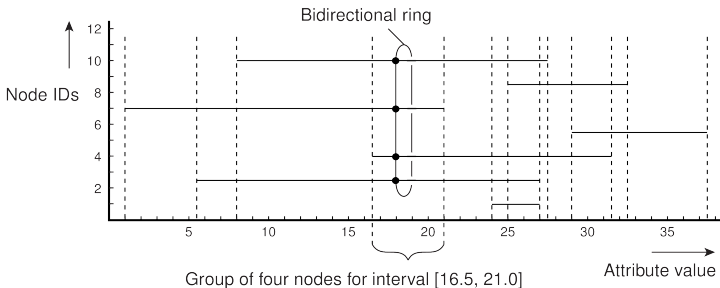
- A subscription  $S_i$  specifies a subset  $\mathbf{S}_i$  in a  $N$ -dimensional space.
- We are interested only in notifications that fall into  $\bar{\mathbf{S}} = \cup \mathbf{S}_i$ .

## Goal

Partition  $\bar{\mathbf{S}}$  into  $M$  disjoint subspaces  $\bar{\mathbf{S}}_1, \dots, \bar{\mathbf{S}}_M$  such that

- **Partitioning:**  $\forall k \neq m : \bar{\mathbf{S}}_k \cap \bar{\mathbf{S}}_m = \emptyset$  and  $\cup_m \bar{\mathbf{S}}_m = \bar{\mathbf{S}}$
- **Subscription coverage:**  $(\bar{\mathbf{S}}_m \cap \mathbf{S}_i \neq \emptyset) \Rightarrow (\bar{\mathbf{S}}_m \subseteq \mathbf{S}_i)$

## Gossiping: Sub-2-Sub



### Consider a single attribute

- Nodes regularly exchange their subscriptions through gossiping
- An intersection between two nodes leads to a mutual reference
- If  $S_{ijk} = S_i \cap S_j \cap S_k \neq \emptyset$  and  $S_{ij} - S_{ijk} \neq \emptyset$ , then:
  - nodes  $i, j, k$  are grouped into a **single overlay network** (for  $S_{ijk}$ )
  - nodes  $i, j$  are grouped into a **single overlay network** (for  $S_{ij} - S_{ijk}$ )

# Secure publish-subscribe

## We are facing nasty dilemma's

- **Referential decoupling**: messages should be able to flow from a publisher to subscribers while guaranteeing mutual anonymity  $\Rightarrow$  we cannot set up a secure channel.
- Not knowing where messages come from imposes **integrity problems**.
- Assuming a **trusted broker** may easily be practically impossible, certainly when dealing with sensitive information  $\Rightarrow$  we now have a **routing problem**.

# Secure publish-subscribe

## We are facing nasty dilemma's

- **Referential decoupling**: messages should be able to flow from a publisher to subscribers while guaranteeing mutual anonymity  $\Rightarrow$  we cannot set up a secure channel.
- Not knowing where messages come from imposes **integrity problems**.
- Assuming a **trusted broker** may easily be practically impossible, certainly when dealing with sensitive information  $\Rightarrow$  we now have a **routing problem**.

## Solution

- Allow for searching (and matching) on encrypted data, without the need for decryption.
- **PEKS**: accompany encrypted messages with a collection of (again encrypted) keywords and search for matches on keywords.

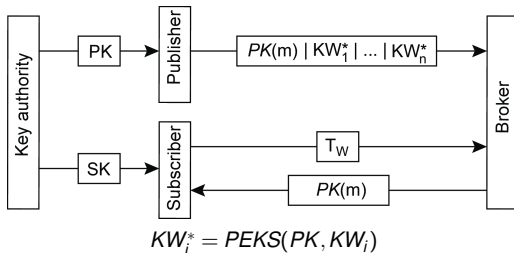
# Public-Key Encryption with Keyword Search (PEKS)

## Basics

- Use a public key  $PK$ , message  $m$  and its  $n$  keywords  $KW_1, \dots, KW_n$  are stored at a server as the message  $m^*$ :

$$m^* = [PK(m) | PEKS(PK, KW_1) | PEKS(PK, KW_2) | \dots | PEKS(PK, KW_n)]$$

- A subscriber gets the accompanying secret key.
- For each keyword  $KW_i$ , a **trapdoor**  $T_{KW_i}$  is generated:  $T_W(m^*)$  will return *true* iff  $W \in \{KW_1, \dots, KW_n\}$ .





# Positioning nodes

## Issue

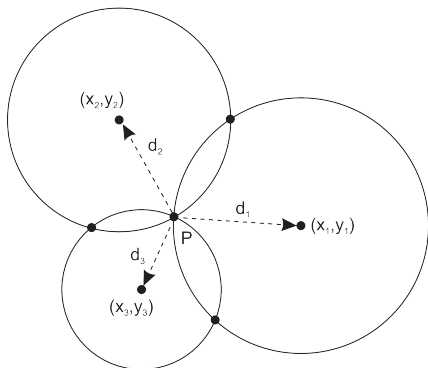
In large-scale distributed systems in which nodes are dispersed across a wide-area network, we often need to take some notion of **proximity** or **distance** into account  $\Rightarrow$  it starts with determining a (relative) **location** of a node.

# Computing position

## Observation

A node  $P$  needs  $d + 1$  **landmarks** to compute its own position in a  $d$ -dimensional space. Consider two-dimensional case.

## Computing a position in 2D



## Solution

$P$  needs to solve three equations in two unknowns  $(x_P, y_P)$ :

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

# Global Positioning System

Assuming that the clocks of the satellites are accurate and synchronized

- It takes a while before a signal reaches the receiver
- The receiver's clock is definitely out of sync with the satellite

## Basics

- $\Delta_r$ : unknown deviation of the receiver's clock.
- $x_r, y_r, z_r$ : unknown coordinates of the receiver.
- $T_i$ : timestamp on a message from satellite  $i$
- $\Delta_i = (T_{now} - T_i) + \Delta_r$ : measured delay of the message sent by satellite  $i$ .
- Measured distance to satellite  $i$ :  $c \times \Delta_i$  ( $c$  is speed of light)
- Real distance:  $d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$

## Observation

4 satellites  $\Rightarrow$  4 equations in 4 unknowns (with  $\Delta_r$  as one of them)

# WiFi-based location services

## Basic idea

- Assume we have a database of known access points (APs) with coordinates
- Assume we can estimate distance to an AP
- Then: with 3 detected access points, we can compute a position.

## War driving: locating access points

- Use a WiFi-enabled device along with a GPS receiver, and move through an area while recording observed access points.
- Compute the centroid: assume an access point  $AP$  has been detected at  $N$  different locations  $\{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$ , with known GPS location.
- Compute location of  $AP$  as  $\vec{x}_{AP} = \frac{\sum_{i=1}^N \vec{x}_i}{N}$ .

## Problems

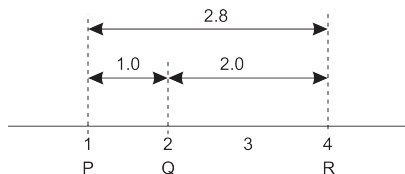
- Limited accuracy of each GPS detection point  $\vec{x}_i$
- An access point has a nonuniform transmission range
- Number of sampled detection points  $N$  may be too low.

# Computing position

## Problems

- Measured latencies to landmarks fluctuate
- Computed distances will not even be consistent

## Inconsistent distances in 1D space



## Solution: minimize errors

- Use  $N$  special **landmark nodes**  $L_1, \dots, L_N$ .
- Landmarks measure their pairwise latencies  $\tilde{d}(L_i, L_j)$
- A central node computes the coordinates for each landmark, minimizing:

$$\sum_{i=1}^N \sum_{j=i+1}^N \left( \frac{\tilde{d}(L_i, L_j) - \hat{d}(L_i, L_j)}{\tilde{d}(L_i, L_j)} \right)^2$$

where  $\hat{d}(L_i, L_j)$  is distance after nodes  $L_i$  and  $L_j$  have been positioned.

# Computing position

## Choosing the dimension $m$

The hidden parameter is the dimension  $m$  with  $N > m$ . A node  $P$  measures its distance to each of the  $N$  landmarks and computes its coordinates by minimizing

$$\sum_{i=1}^N \left( \frac{\tilde{d}(L_i, P) - \hat{d}(L_i, P)}{\tilde{d}(L_i, P)} \right)^2$$

## Observation

Practice shows that  $m$  can be as small as 6 or 7 to achieve latency estimations within a factor 2 of the actual value.

# Vivaldi

## Principle: network of springs exerting forces

Consider a collection of  $N$  nodes  $P_1, \dots, P_N$ , each  $P_i$  having coordinates  $\vec{x}_i$ .  
Two nodes exert a **mutual force**:

$$\vec{F}_{ij} = (\tilde{d}(P_i, P_j) - \hat{d}(P_i, P_j)) \times u(\vec{x}_i - \vec{x}_j)$$

with  $u(\vec{x}_i - \vec{x}_j)$  is the unit vector in the direction of  $\vec{x}_i - \vec{x}_j$

## Node $P_i$ repeatedly executes steps

1. Measure the latency  $\tilde{d}_{ij}$  to node  $P_j$ , and also receive  $P_j$ 's coordinates  $\vec{x}_j$ .
2. Compute the error  $e = \tilde{d}(P_i, P_j) - \hat{d}(P_i, P_j)$
3. Compute the direction  $\vec{u} = u(\vec{x}_i - \vec{x}_j)$ .
4. Compute the force vector  $\vec{F}_{ij} = e \cdot \vec{u}$
5. Adjust own position by moving along the force vector:  $\vec{x}_i \leftarrow \vec{x}_i + \delta \cdot \vec{u}$ .