Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025

Section 5: Coordination

This content is based on the following public resources: https://www.distributed-systems.net/index.php/books/ds4/

Mutual exclusion

Process Critical Sections

Problem

Several processes in a distributed system want exclusive access to some resource - and we want to avoid concurrent access to this resource: The processes all have a region of code where the concurrency takes place. This is called the **Critical Section**

Critical Section

A **critical section** is a code segment within a process (or thread) that accesses or modifies shared resources (e.g., variables, data structures, files, database or devices) and must be executed atomically — as an indivisible unit—by at most one **one concurrent process or thread** at a time.

Mutual exclusion: Example

```
do {
    // Non-critical section (local work, safe for concurrency)

    // Entry section (acquire synchronization primitive)
    while (!can_enter_critical_section());  // Busy-wait or block

    // *** CRITICAL SECTION ***
    // Shared resource access/modification (e.g., balance += amount;)
    // Must be atomic w.r.t. other threads

    // Exit section (release synchronization primitive)
    signal_exit_critical_section();

    // Non-critical section
} while (true);
```

Mutual exclusion

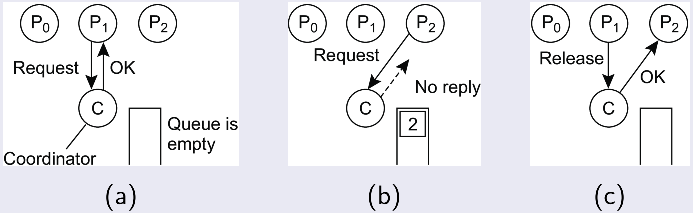
Basic solutions

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

Acknowledgments are not sent (the caller blocks)

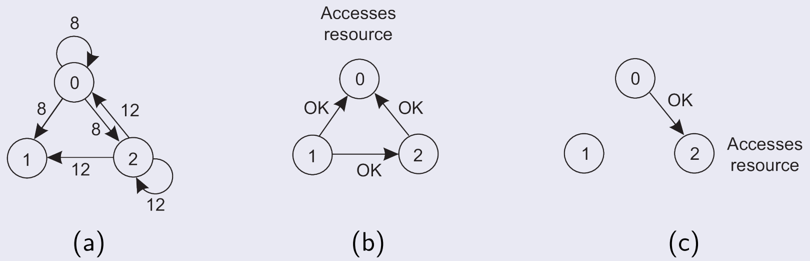
Return a response to a request only when:

- 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



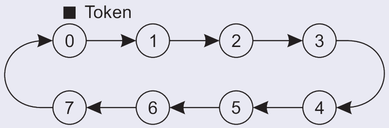
(a) Two processes want to access a shared resource at the same moment.
(b) P_0 has the lowest timestamp, so it wins.
(c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

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 Δt, while having a lifetime of T.
- The probability $\mathbb{P}[k]$ that k out of m coordinators reset during the same interval is

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

- for $m > N/2$, if 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

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	$1,\ldots,\infty$	$0,\ldots,N-1$
Decentralized	$2kN+(k-1)N/2+N, k=1,2,\ldots$	$2kN+(k-1)N/2$

Decentralized Case messages per entry/exit for $k \geq 1$ attempts:

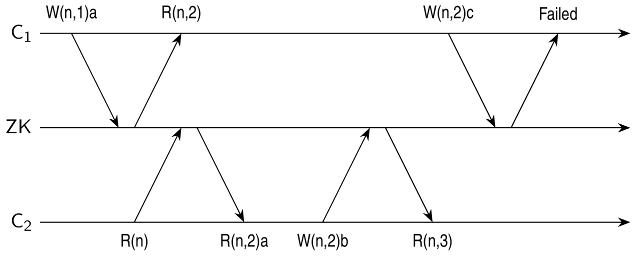
- send N messages to coordinators receive N responses.
- If it does not get a majority, release $N/2$ votes.
- If it did get enough votes, send N release messages later.
- A process may need to go through $k \geq 1$ attempts

Example: ZooKeeper

Basics (and keeping it simple)

- Centralized server setup
- Distributed Nodes
- 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 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

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 ⇒ 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

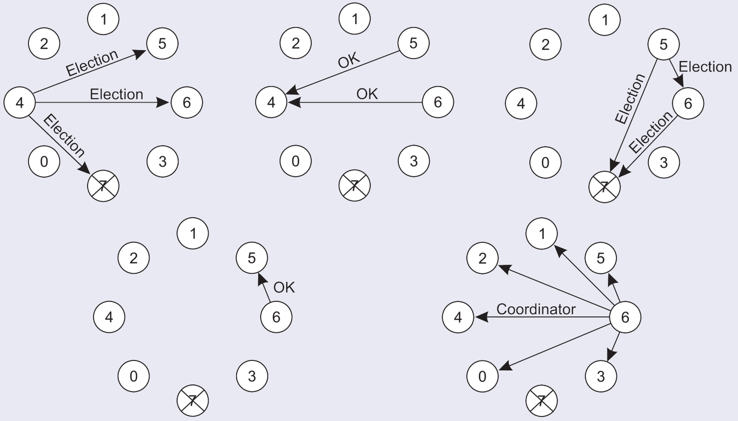
Principle

Consider N processes $\{P_0, \ldots, P_{N-1}\}$ 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: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
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

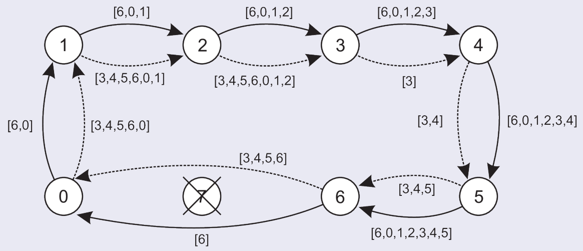
Principle

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.

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

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

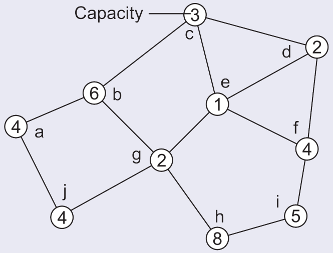
- Draw a random number $k \in \{1, \ldots, N\}$
- Look up the process P that owns the token with index k. P is the next leader.

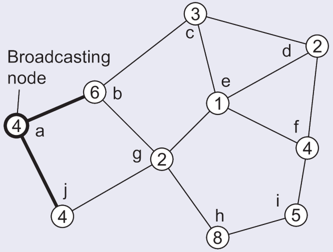
Observation

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

A solution for wireless networks

A sample network



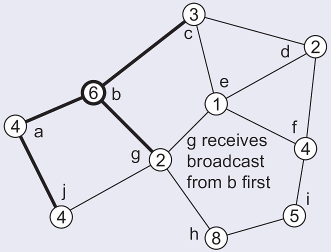


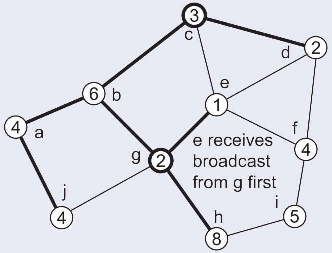
Essence

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

A solution for wireless networks

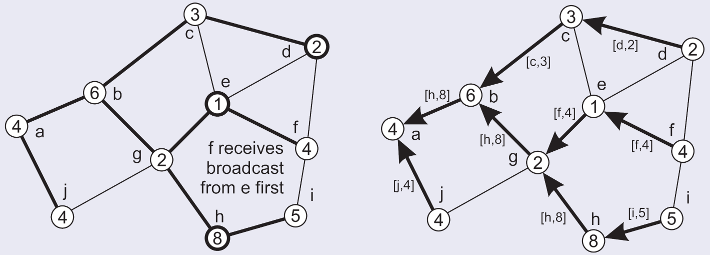
A sample network





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

Gossip-based coordination

- Nodes randomly select a few other nodes (peers) periodically and exchange state information (e.g., membership, failure detection, key-value updates, or metadata).
- Information spreads exponentially fast, similar to how gossip or a virus spreads in a population, achieving eventual consistency with very high probability in logarithmic rounds.
- Advantages: Extremely robust to network partitions, node failures, and churn; no single point of failure; low coordination overhead; scales to thousands or tens of thousands of nodes with minimal latency impact; self-healing membership and failure detection (e.g., SWIM, HyParView, or Lifeguard protocols).
- Trade-offs & Common Uses: Provides only probabilistic guarantees (not strict consistency); latency of propagation is O(log N); widely used for failure detection (accrual detectors),
- Lightweight broadcast instead of heavy consensus protocols like Paxos or Raft when strong consistency isnt required.

Gossip-based coordination: aggregation

Typical apps

- Data dissemination: Perhaps the most important one. Note that there are many variants of dissemination.
- Aggregation Function (Average): 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 from its local list.

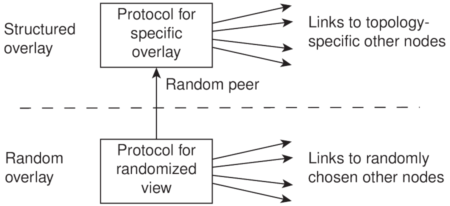
Observation

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

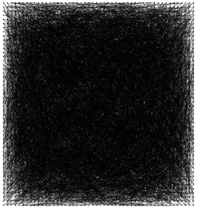
Maintain two local lists of neighbors. The lowest is used for providing a peer-sampling service; the highest list is used to carefully select application-dependent neighbors.

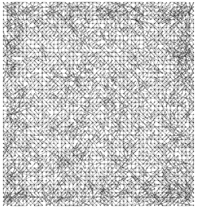

Structured overlay
Protocol for specific overlay
Links to topology-specific other nodes
Random peer
Random overlay
Protocol for randomized view
Links to randomly chosen other nodes

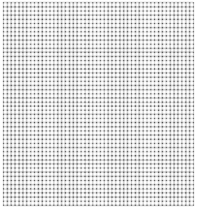
Gossip-based overlay construction: a 2D torus

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 $(N = 50)$

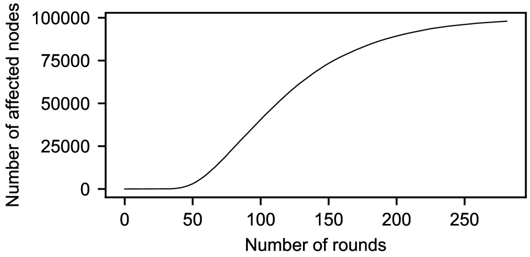

after 5 rounds


after 20 rounds

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.

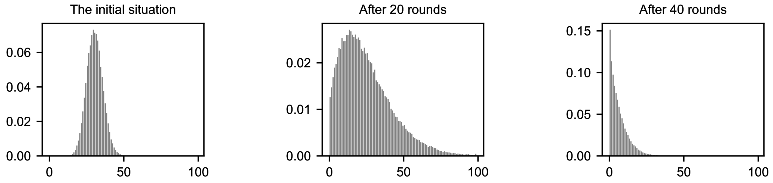


Situation

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

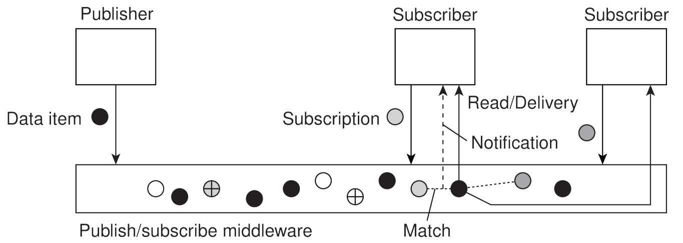
When a benign node initiates an exchange, it may either use the result for gathering statistics, or for updating its local list. An attacker is in limbo: will its response be used for statistical purposes or for functional purposes?

Observation

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

Distributed event matching

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

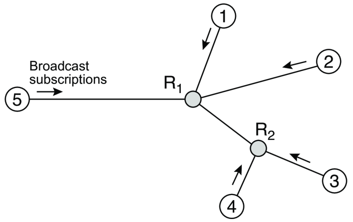
- *sub2node*(S): map a subscription S to a nonempty subset $\mathbf{S}$ of servers
- *not2node*(N): map a notification N to a nonempty subset $\mathbf{N}$ of servers

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

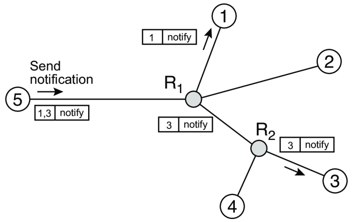
Observations

- 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 rendezevous 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

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

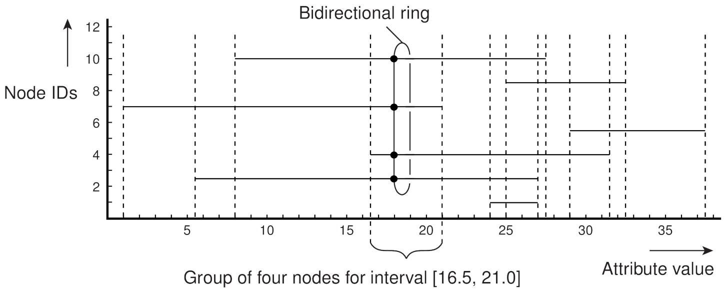
- A subscription S_i specifies a subset $\mathbf{S_i}$ in a N-dimensional space.
- We are interested only in notifications that fall into $\overline{\mathbf{S}} = \cup \mathbf{S_i}$.

Goal

Partition $\overline{\mathbf{S}}$ into M disjoint subspaces $\overline{\mathbf{S}}_\mathbf{1}, \dots, \overline{\mathbf{S}}_\mathbf{M}$ such that

- Partitioning: $\forall k \neq m : \overline{\mathbf{S}}_\mathbf{k} \cap \overline{\mathbf{S}}_\mathbf{m} = \emptyset$ and $\cup_m \overline{\mathbf{S}}_\mathbf{m} = \overline{\mathbf{S}}$
- Subscription coverage: $(\overline{\mathbf{S}}_\mathbf{m} \cap \mathbf{S_i} \neq \emptyset) \Rightarrow (\overline{\mathbf{S}}_\mathbf{m} \subseteq \mathbf{S_i})$

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 $\mathbf{S_{ijk}} = \mathbf{S_i} \cap \mathbf{S_j} \cap \mathbf{S_k} \neq \emptyset$ and $\mathbf{S_{ij}} - \mathbf{S_{ijk}} \neq \emptyset$, then:
  - nodes *i*, *j*, *k* are grouped into a single overlay network (for $\mathbf{S_{ijk}}$)
  - nodes *i*, *j* are grouped into a single overlay network (for $\mathbf{S_{ij}} - \mathbf{S_{ijk}}$)

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.

This is a real concern when the broker / message bus is untrusted or semi-trusted (e.g., cloud provider, third-party MQTT broker, Kafka cluster you don't fully control)

Secure publish-subscribe

Some real world examples:

- Financial market data feeds: Bloomberg, Refinitiv, or cloud feeds cannot let the provider see which banks are interested in “Argentina default"
- Privacy-preserving: ride-sharing / location pub-subDrivers publish encrypted location blocks; riders only want their 1 km grid square
- Military / classified pub-sub: Topics themselves are classified (**DRONE-STRIKE-X17**)

We must use secure pub/sub whenever:

- The broker is not fully trusted with message confidentiality, and
- Subscriptions are fine-grained (keywords, topics, attributes), and
- Publishers and subscribers are not in the same closed trust domain (open or Internet-facing system).

Encrypted Everything?

Why "just encrypt everything" doesn't work If you encrypt messages (using asymmetric key encryption):

- Broker can't read them
- But the broker also can't route them, because:
- It doesn't know which subscription matches which message
- So the broker either:
- Sees everything (bad for security), or
- Sees nothing (bad for routing)

This is the core dilemma.

Use Encrypted Search

Solution

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

```
Subscriber:
  1. has private key sk
  2. decides he wants all documents containing "confidential"
  3. computes T = Trapdoor(sk, "confidential")
  4. sends T (in the clear, over TLS) to the server/cloud
```

PEKS

Broker (Router/Server) tasks:

```
for each stored ciphertext C_i:
  if Test(public_key, C_i, trapdoor_bytes) == True:
      return this document to the Subscriber
```

- No encryption of the trapdoor is required,
- No shared secret between the Subscriber and the server is needed beyond normal transport-layer security (TLS).
- In this way, **PEKS** trapdoors are passed in all real systems.

Summary

Summary

The topics discussed in the *coordination* section of the lecture notes include

1. Mutual Exclusion
2. Election Algorithms
3. Gossip-based Coordination
4. Distributed Event Matching
5. Gossip-based Pub-Sub
6. Secure Gossip-based Pub-Sub