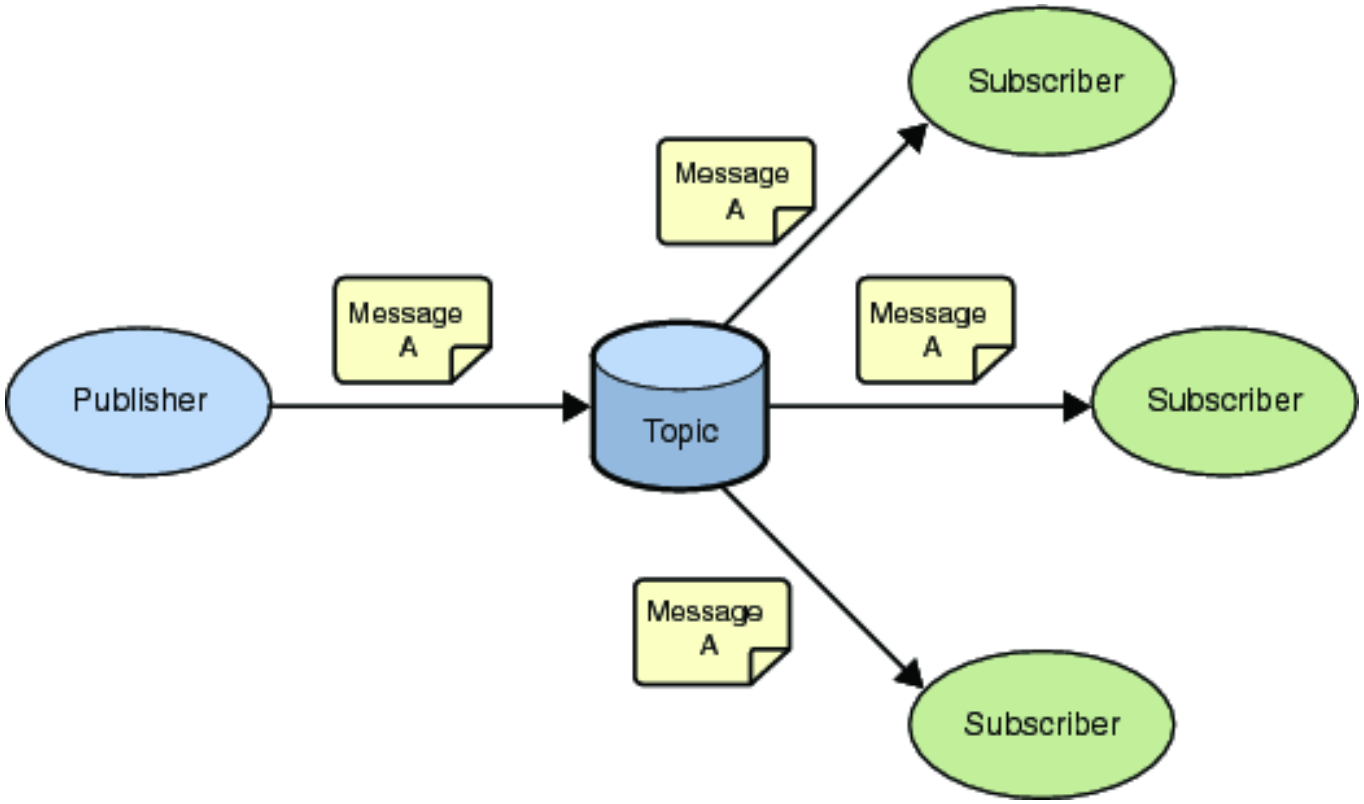
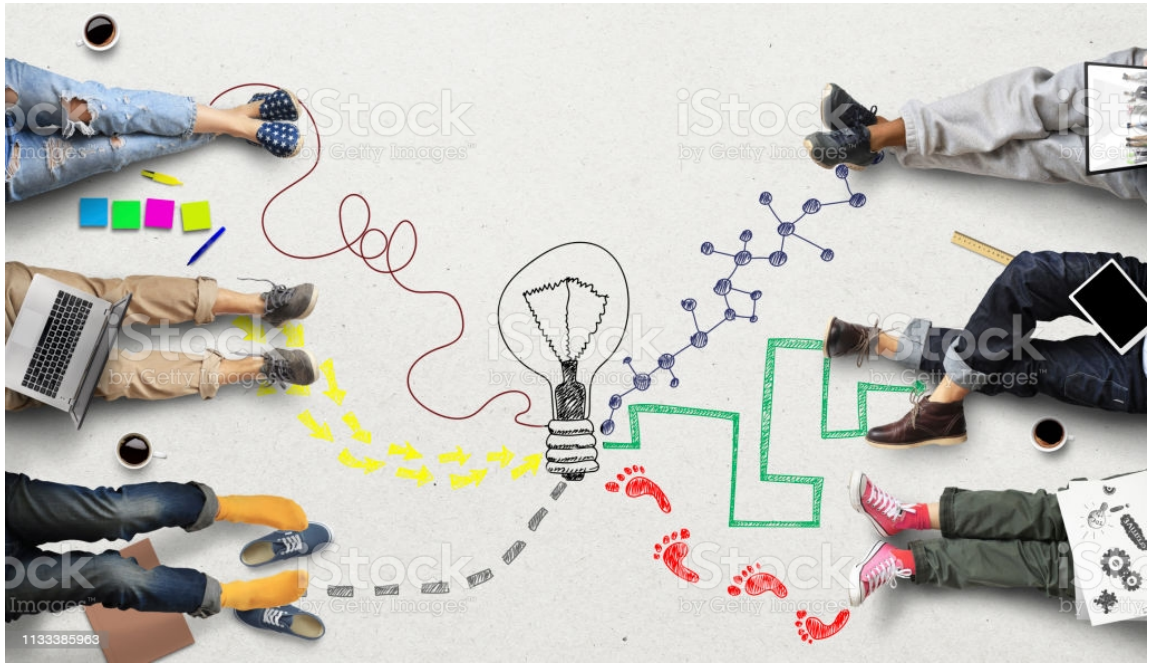
Publish/Subscribe Systems



Agenda

- Publish/Subscribe
  - Definition
  - Decoupling
  - Applications
- Matching
  - Topic-based Model
  - Content-based Model
- Routing
  - Rendezvous-based
  - Overlay-based routing

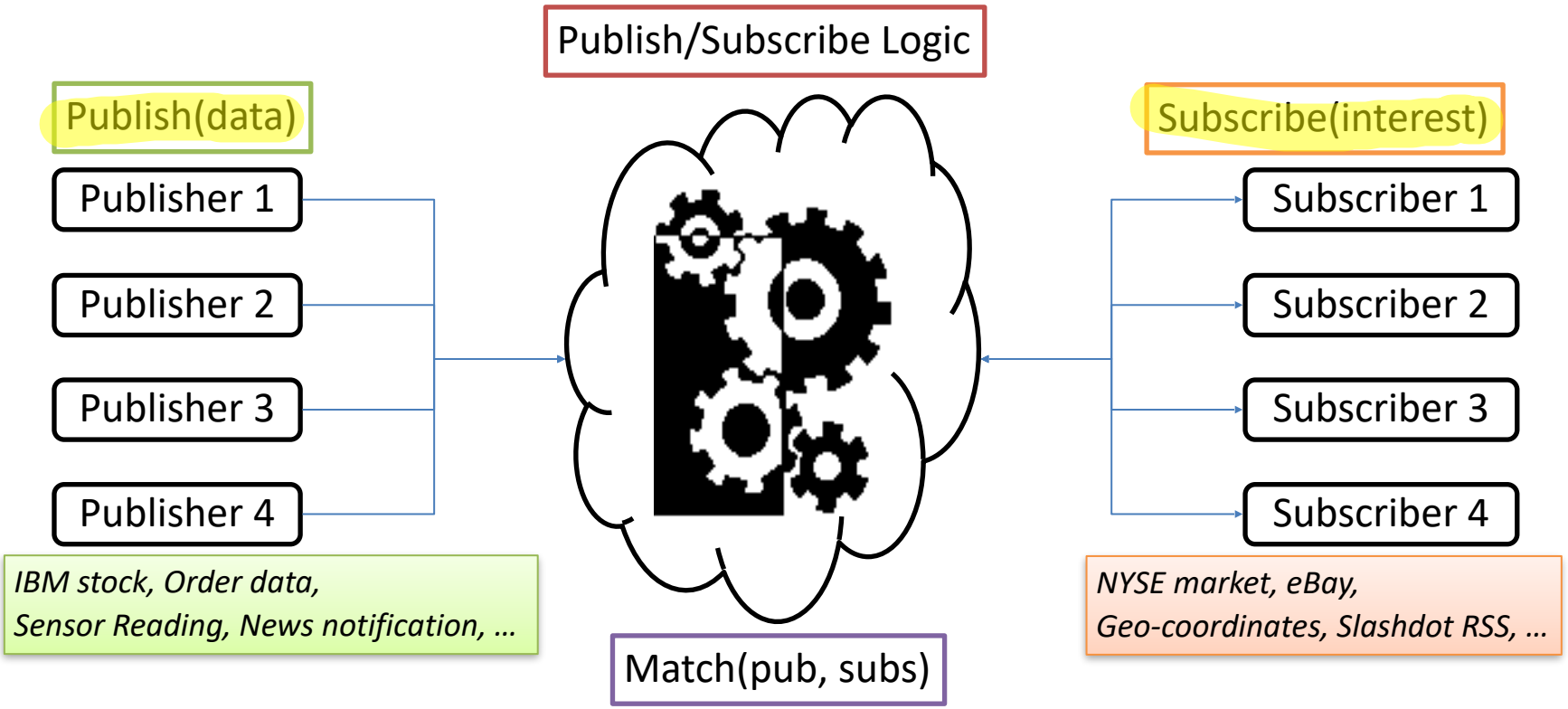


Pixabay.com

PUBLISH/SUBSCRIBE

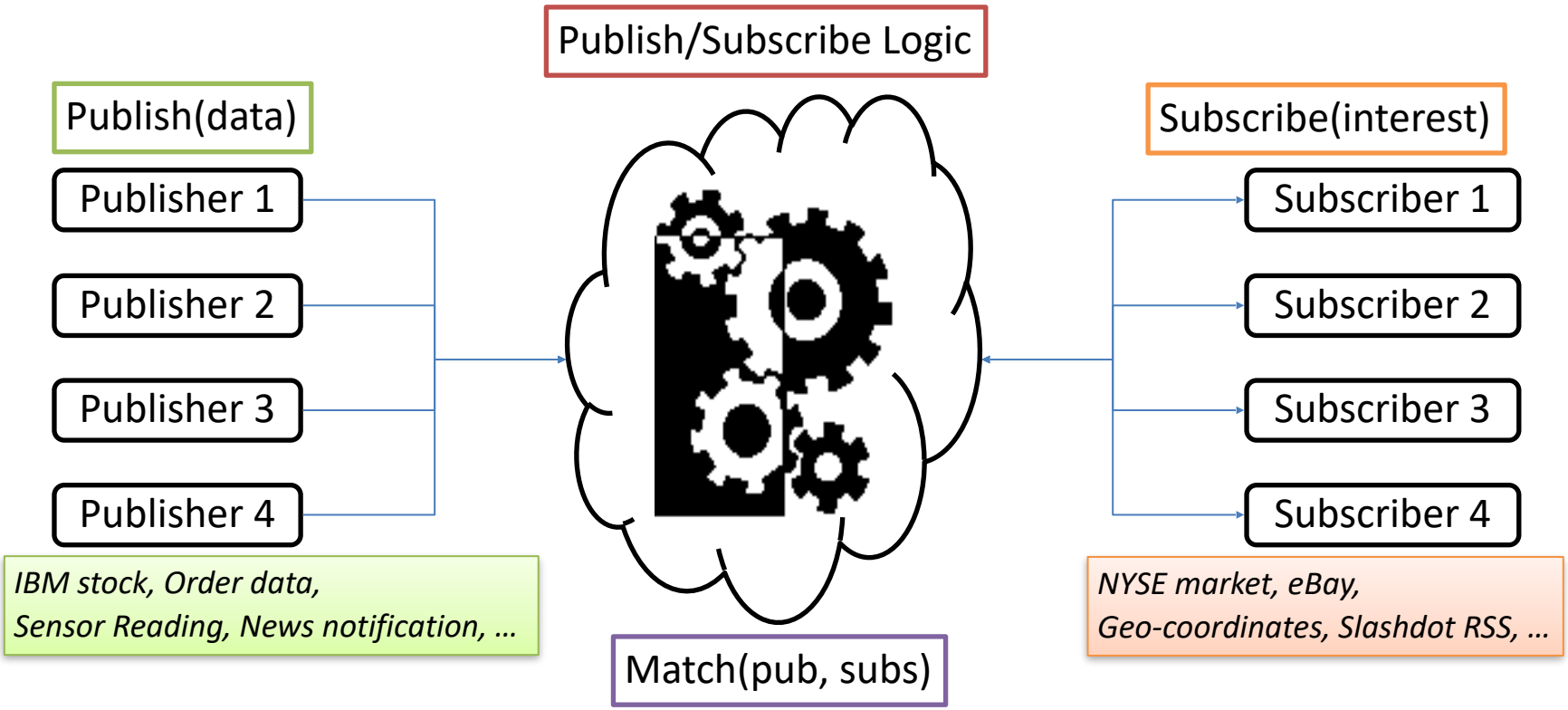
DEFINITION AND MODEL

Definitions



Publish/Subscribe is a **messaging pattern** which allows for **many-to-many** communication: Data sources (publishers) publish **publications**, data sinks (subscribers) subscribe via **subscriptions** and receive **notifications**, i.e., publications of interest (publications matching registered subscriptions.)

Definitions



Publish/Subscribe is a **messaging pattern** which allows for **many-to-many** communication: Data sources (publishers) publish **publications**, data sinks (subscribers) subscribe via **subscriptions** and receive **notifications**, i.e., publications of interest (publications matching registered subscriptions.)

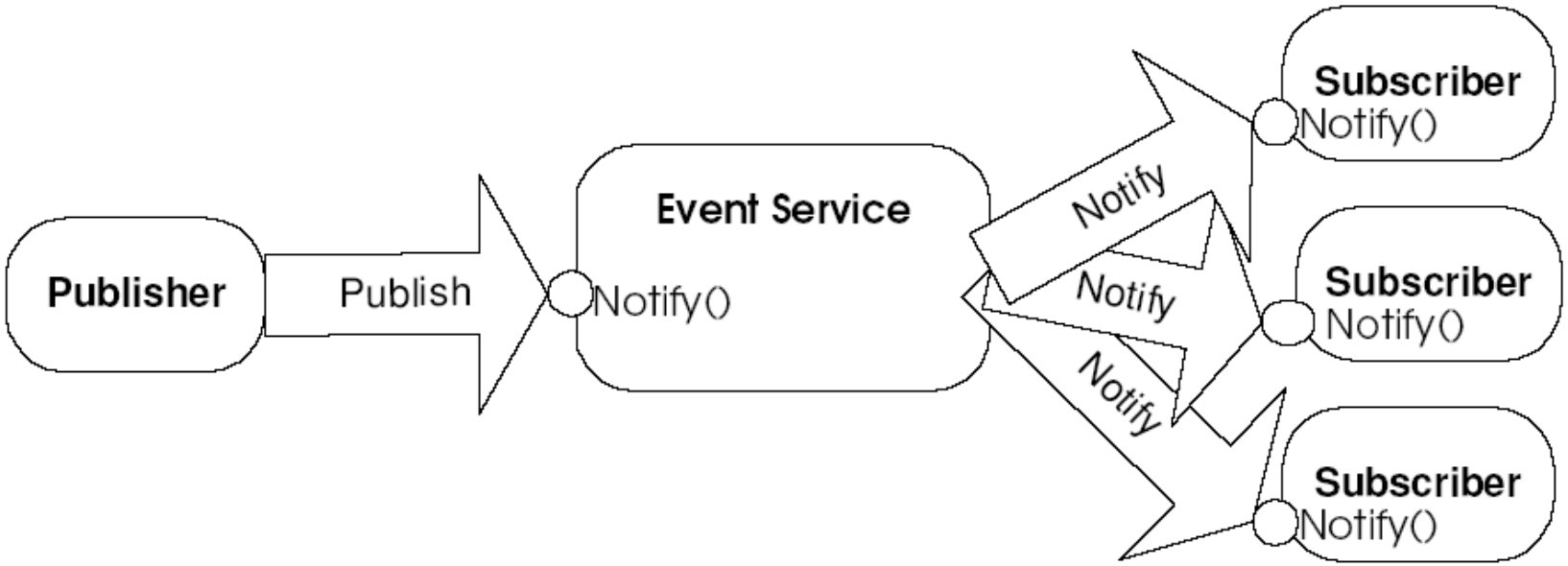
Pub/Sub API

- `Publish(data, metadata)` – publication metadata depends on the language (cf. Matching model)
- `Subscribe(interest)` – subscription interest is defined depending on the language (cf. Matching model)
- `Unsubscribe(interest)` – remove a subscription
- *`(Un-)Advertise(data type/domain)` – publisher advertises type of content it will publish*

Pub/Sub System Design Dimensions

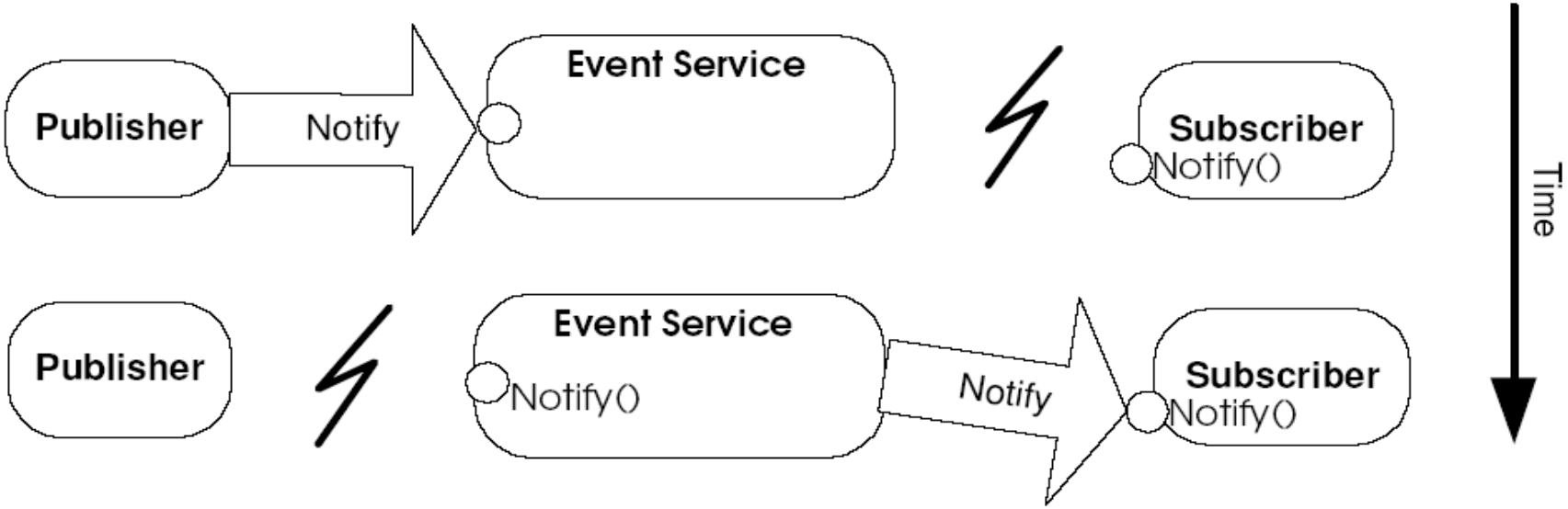
- Matching-based communication
  - Not host-to-host communication
  - Communication is based on the **matching model**
- Optimized for **scalability** and **performance**
  - Large number of publishers and subscribers
  - High rate of publications
  - Fast matching: usually **stateless** (does not consider sequences or composite patterns)
  - Simple matching: commonly limited to **topics**
- **Decoupling** properties (most popular):
  - Time
  - Space
  - Synchronization

Space Decoupling



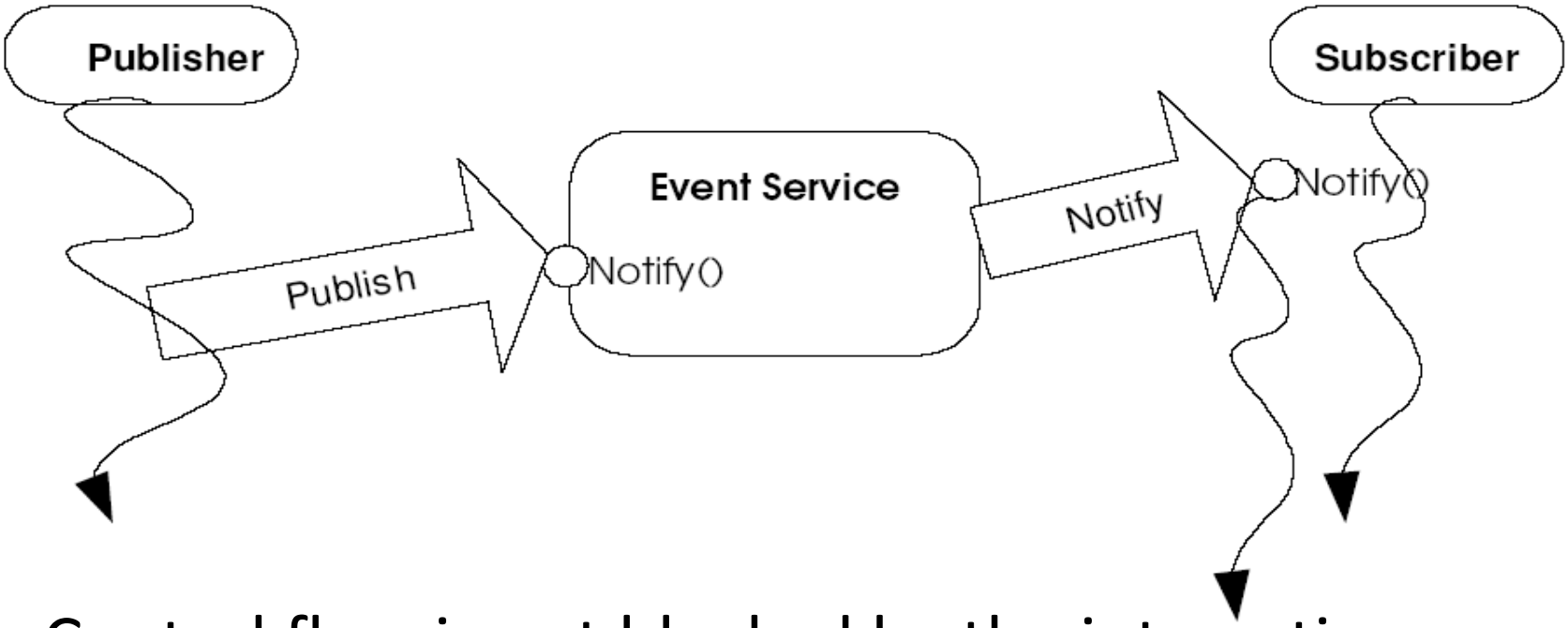
- No need for clients (publishers & subscribers) to hold references to or even know about each other
- Clients may be physically distributed

Time Decoupling



- No need for clients to be available at the same time

Synchronization Decoupling



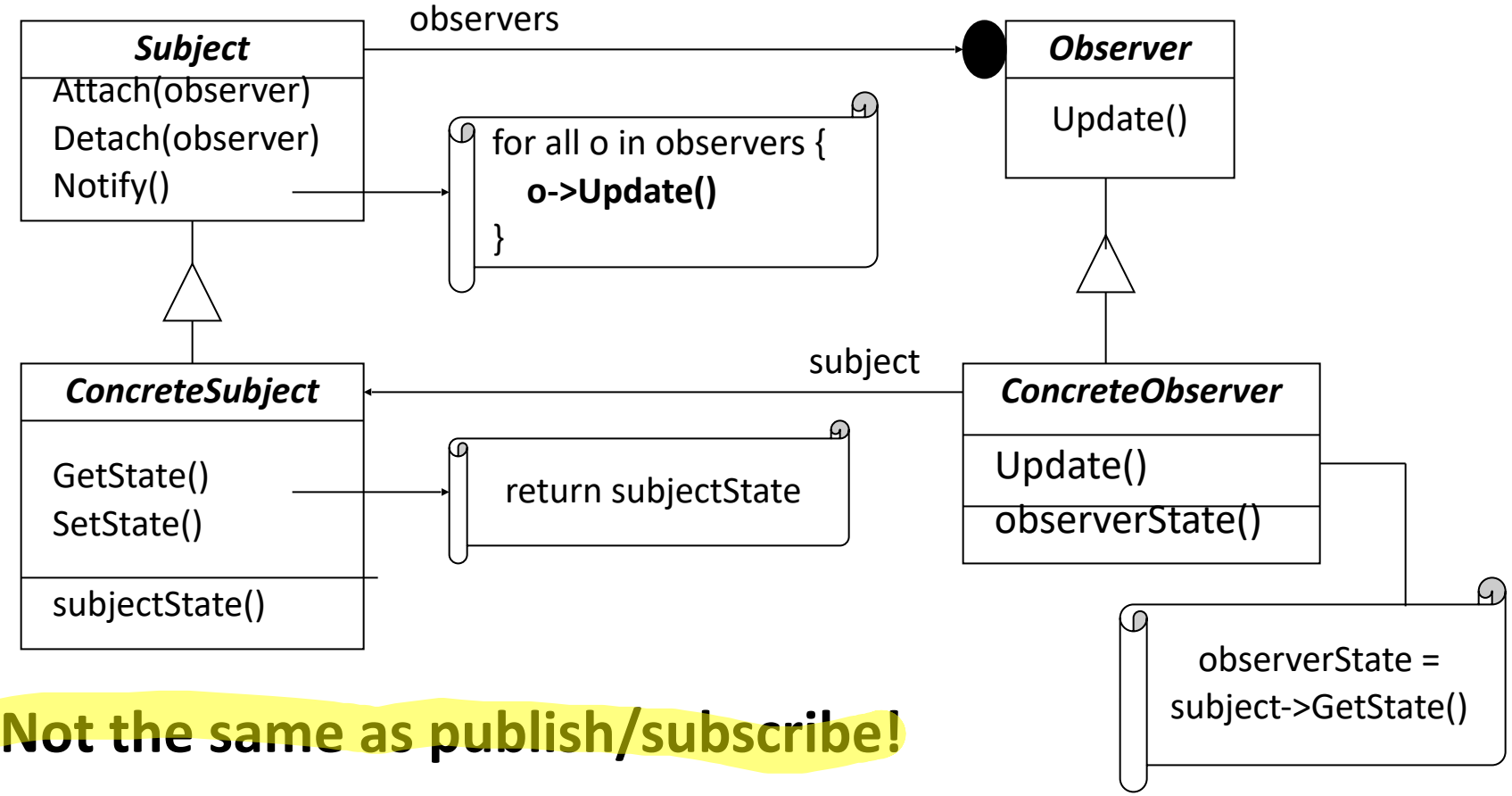
- Control flow is not blocked by the interaction

Why Decouple?

- Decouple publishing and subscribing end-points
  - Removes explicit dependencies
  - Reduces coordination
  - Reduces synchronization
- Creates highly dynamic systems
  - Data sources and sinks come and go
  - Supports continuous operation
- Enables event-based systems, increases scalability of system
- Makes debugging system more difficult

Pub/Sub vs. Observer Design Pattern

a.k.a. Listener

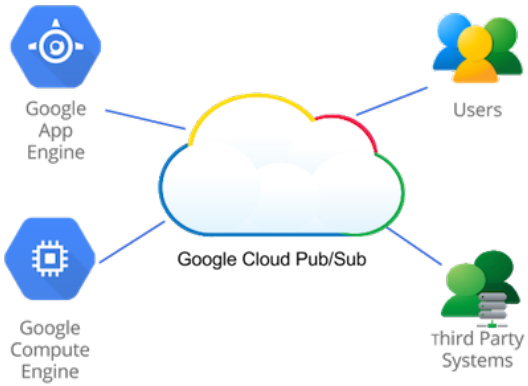


- **Not the same as publish/subscribe!**
- Publisher (subject) knows subscribers (observer) and vice versa (tightly coupled)

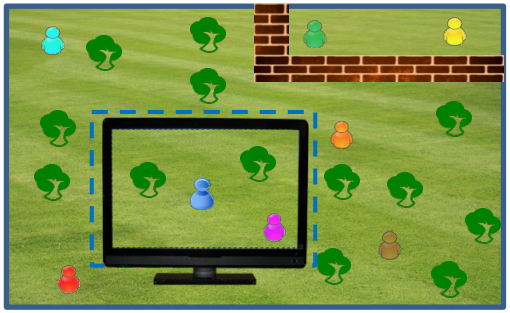
Applications

News Syndication

Stock Tracker



App Notifications

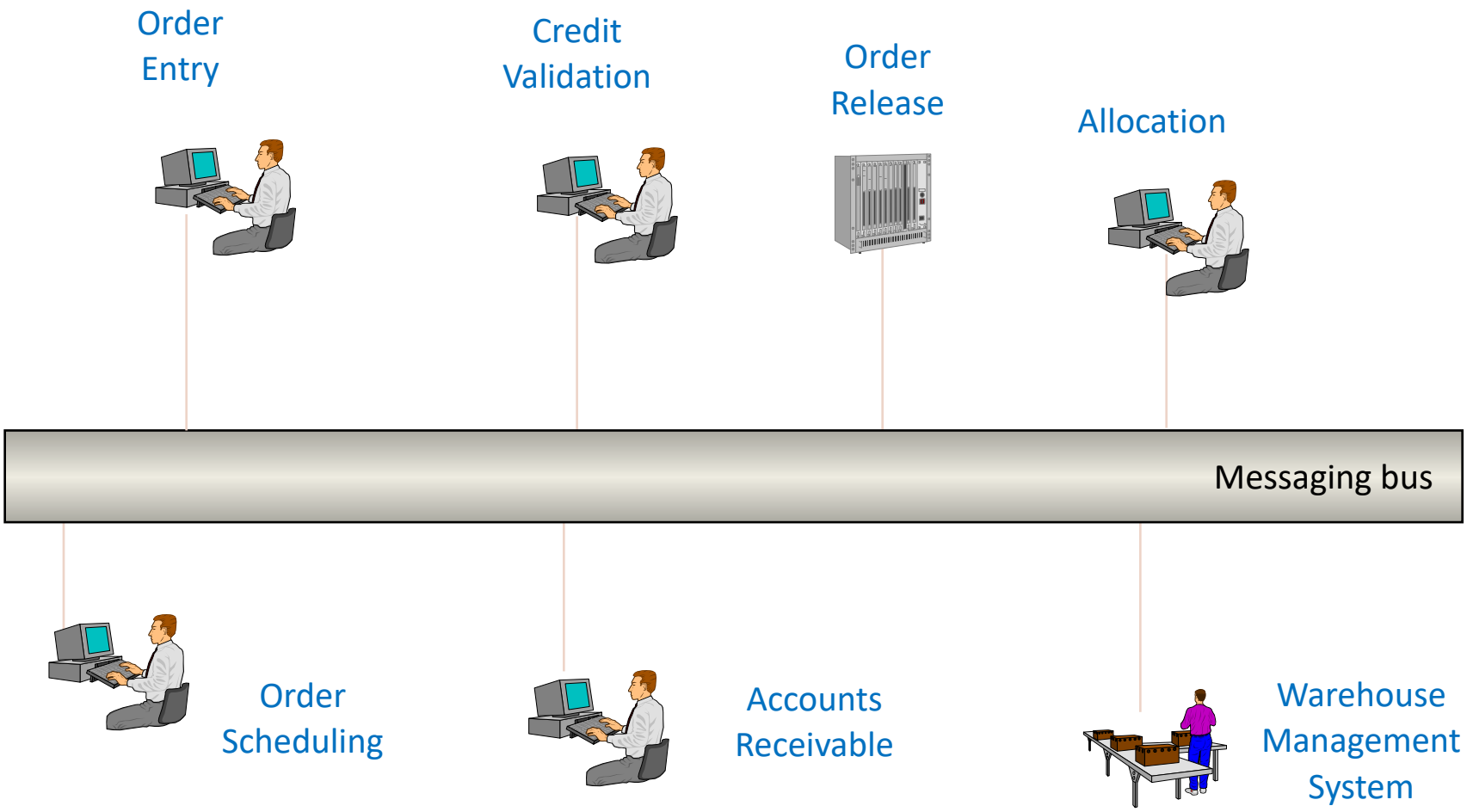


Network Engine for Games

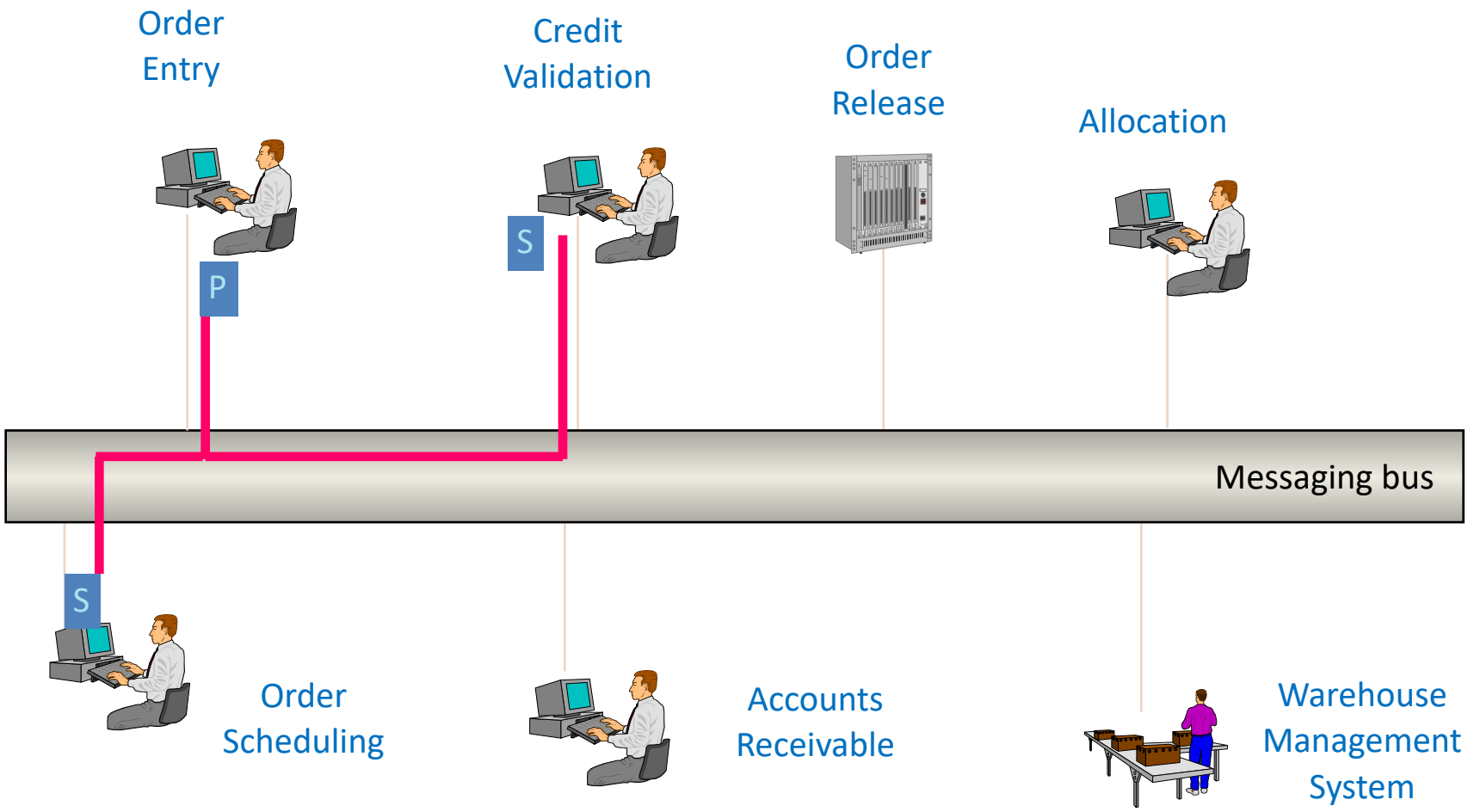
Social Networks

Traffic Monitoring

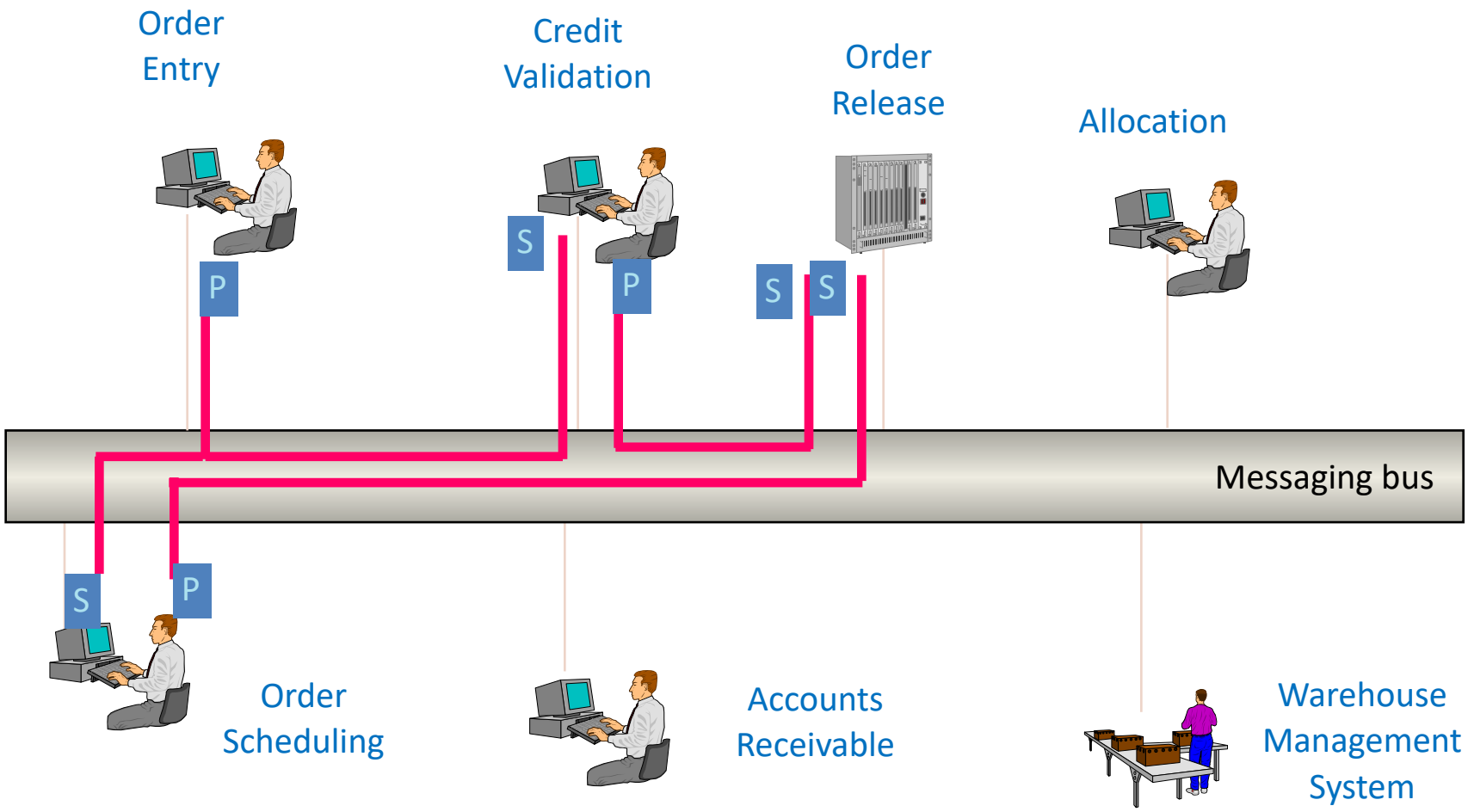
Enterprise Application Integration



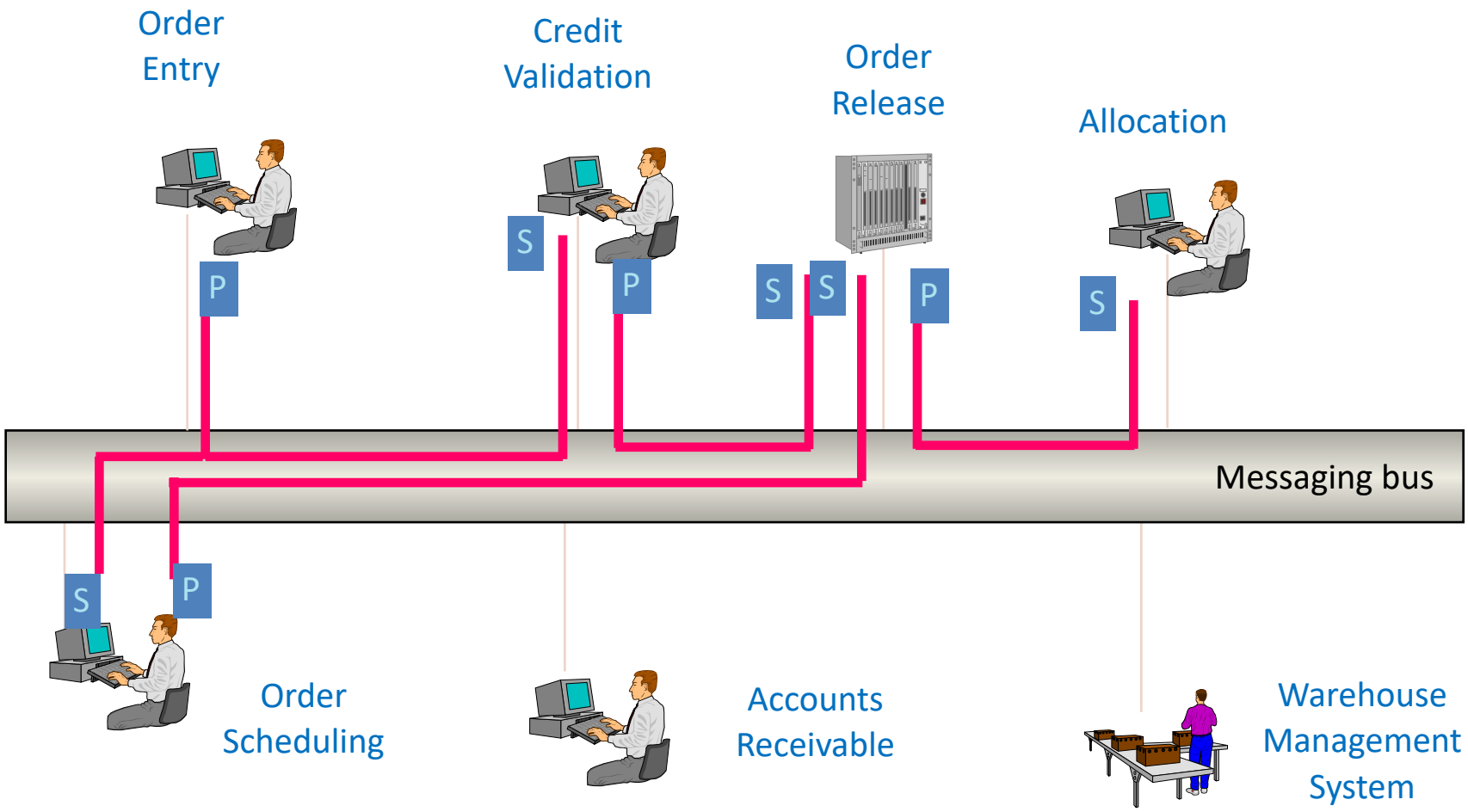
Enterprise Application Integration



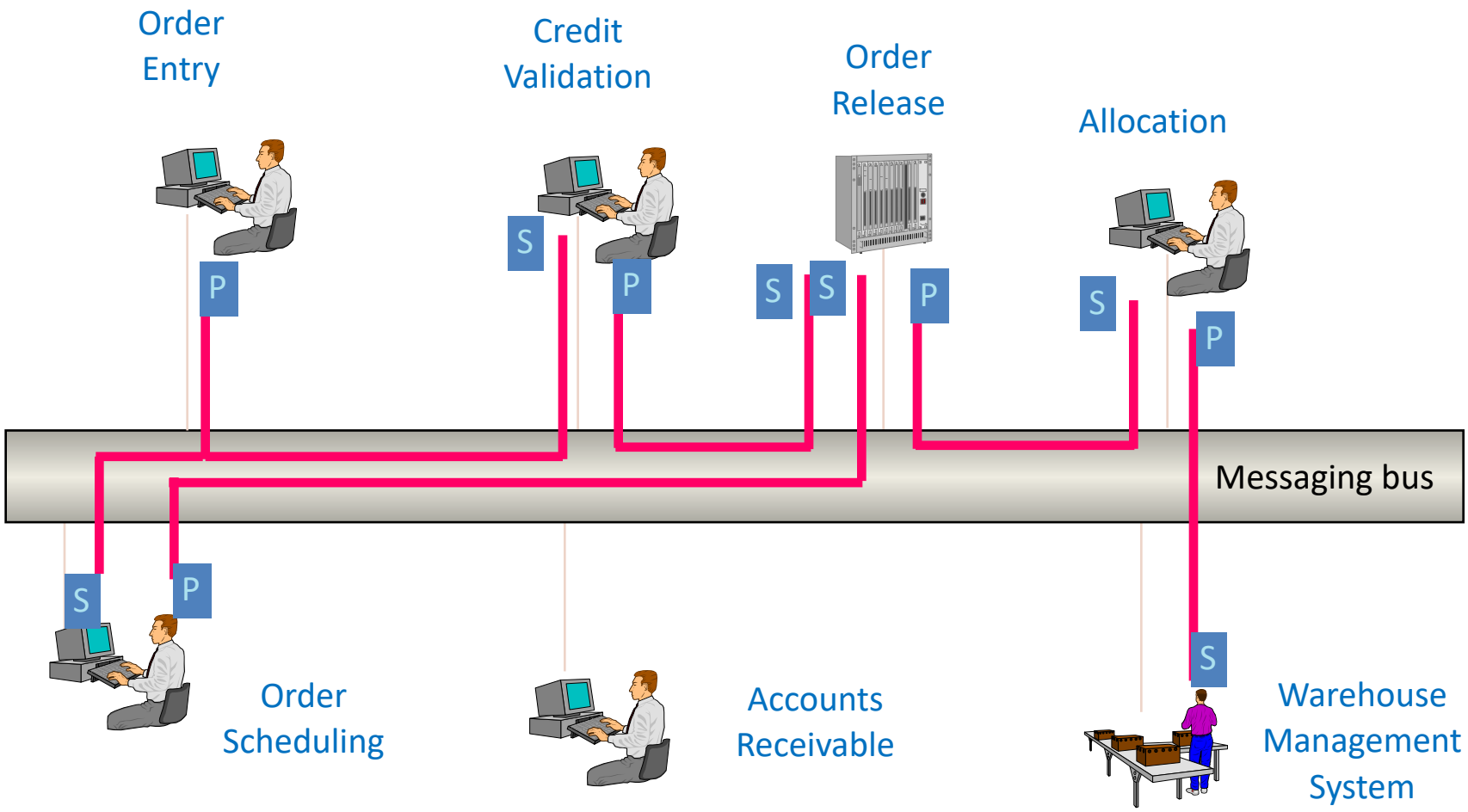
Enterprise Application Integration



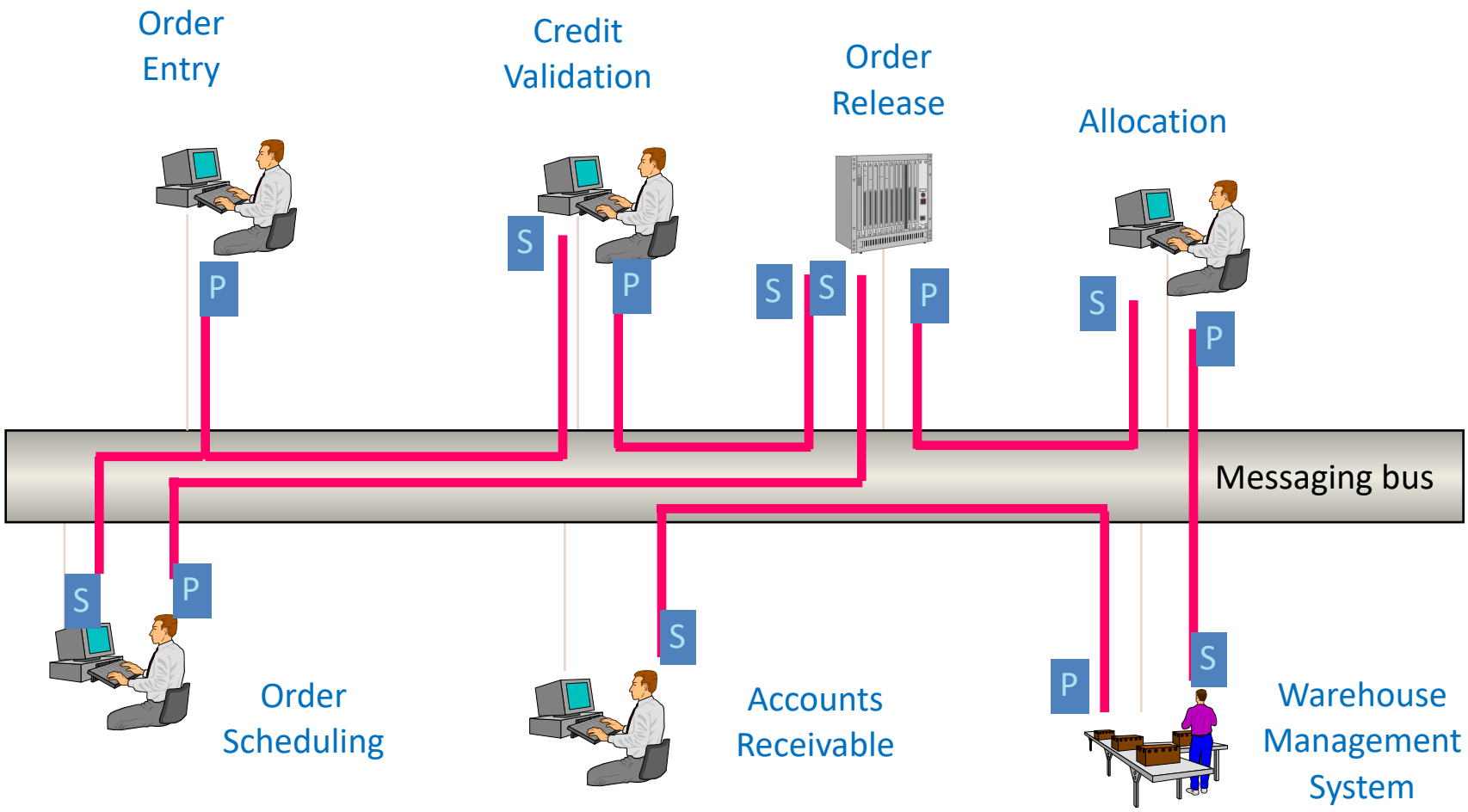
Enterprise Application Integration



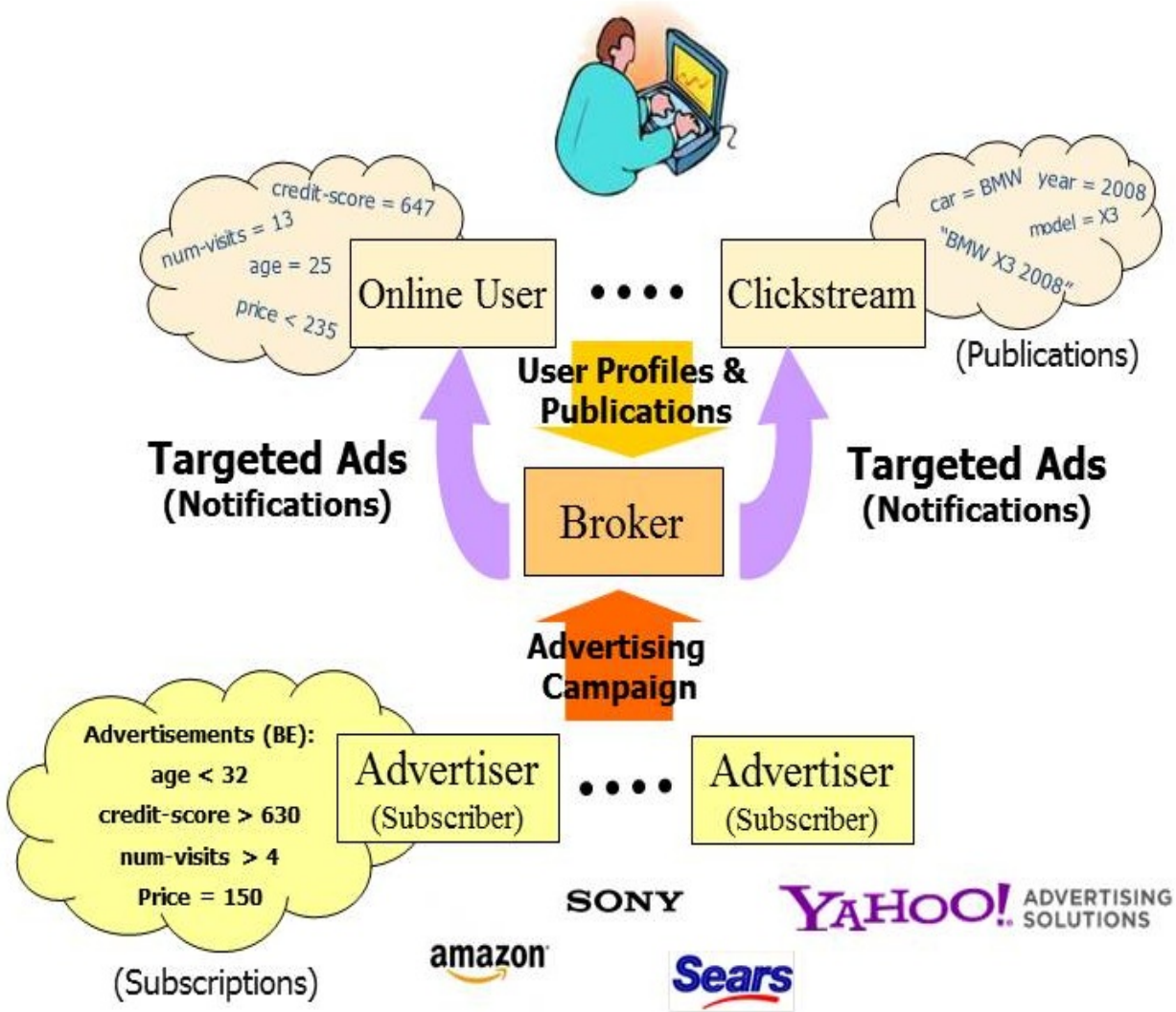
Enterprise Application Integration



Enterprise Application Integration



Computational Advertising: Filtering & Matching



Pub/Sub Standards

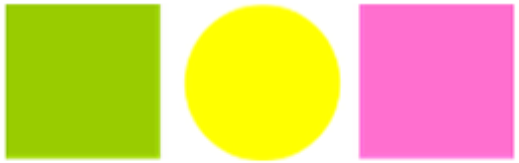
- DCE Event Management Service (1990s)
- OMG Event Channel (1995)
- OMG Notification Service (1997)
- OMG Data Dissemination Service
- MQ Telemetry Transport (1999)
- Java Messaging Service (JMS) (1998, 2002)
- WS Eventing (2004)
- WS Notifications (2004)
- Active Message Queueing Protocol (AMQP)
- OGSI-Notification (6/2003) (Open Grid Services Architecture)

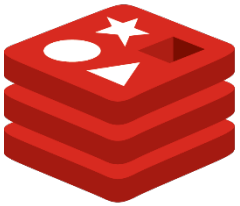
Popular Systems in Industry

PULSAR

kafka

IBM WebSphere.





redis

PubSubHubbub

Facebook Wormhole

SNS

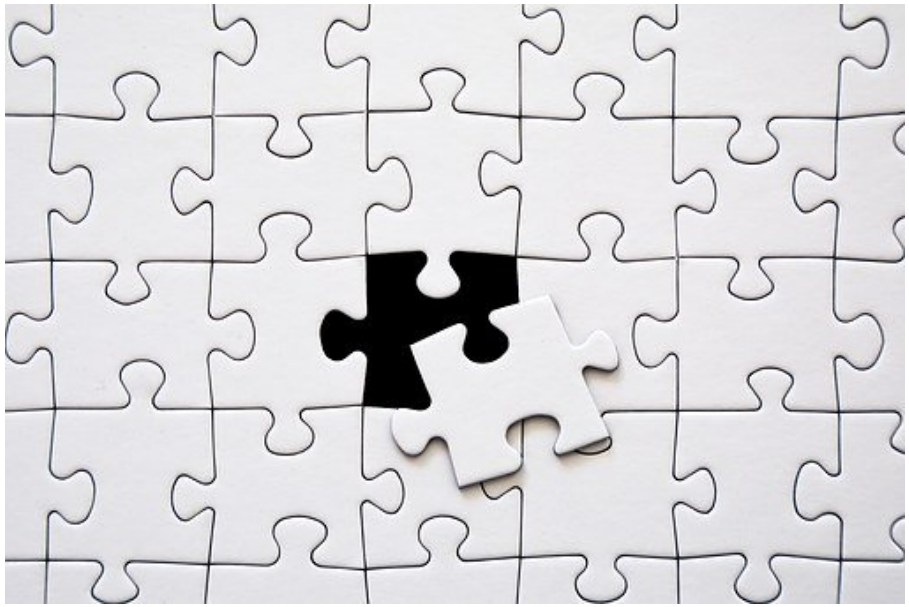
amazon web services™

Google Cloud Pub/Sub

PubNub

Summary of Pub/Sub

- Pub/sub provides **many-to-many** communication between publishers and subscribers
- Communication is decoupled with respect to at least **time, space, and synchronization**
- Many applications are **event-based** or require **event notifications**, which leverage pub/sub
- Many flavors of pub/sub, which differ in **matching** and **routing**



Pixabay.com

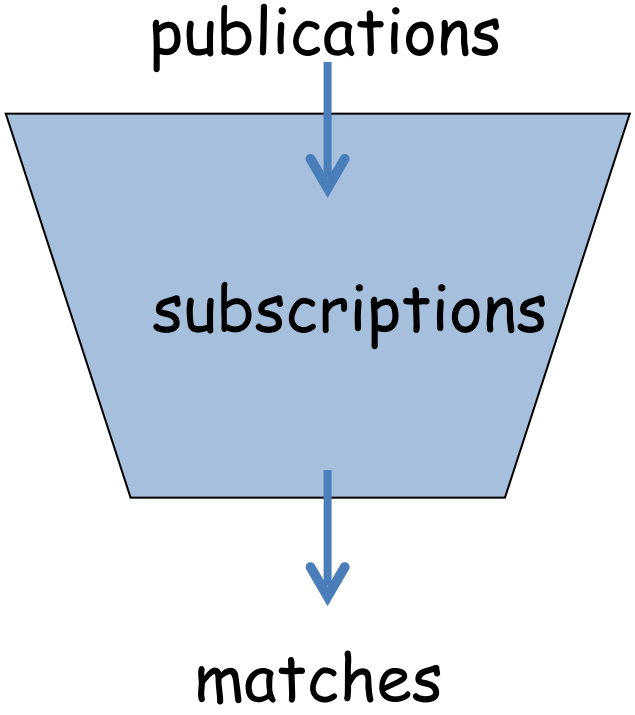
PUBLISH/SUBSCRIBE

MATCHING MODELS

Matching & Filtering Problem in Pub/Sub

- **Matching problem**: Given a publication, e, and a set of subscriptions, *S*, determine all subscriptions, $s \in S$, that match *e*.

- Different for each model
  - Topic-based filtering
  - Content-based filtering
  - Type-based filtering
  - …



Matching Model

- The matching model describes **how** subscriptions and publications are **expressed**.
- Each subscription contains **filters** which determine whether a publication matches or not.
- When a publication is sent, it is compared against each subscription to see which ones **match**.
- The publication is then delivered to **each subscriber with a matching subscription**.

Topic-Based Matching

- A.k.a. subject-based, group-based or channel-based event filtering (yet, there are subtle differences!)
- Publish(data, metadata) → Publish(data, topic)
- Subscribe(interest) → Subscribe(topic)
- A publication **matches** a subscription if it is published on the **same topic** as the subscription.
- Example:

  subscribe("IBM"),

  publish("price:175.31","IBM")

Topic-Based Matching II

- Static publisher & subscriber relationship
  - At publication time, the subscriber recipient set is known
  - For a given topic, unless topics or subscriptions change, every message published on the topic, is delivered to the same recipient set
- Topics can be hierarchically organized
  - E.g., `sports/basketball/NBA`
  - E.g., publish message to `sports/basketball/NBA`
- Topic subscriptions may contain wildcards
  - E.g., Subscribe to `sports/basketball/*`

Content-Based Matching

- **Subscription**: Conjunction of **predicates** (i.e., an attribute-operator-value triple)

  (subject = news) ∧ (topic = travel) ∧ (date > 29.11.2011)

- **Publication (a.k.a. event)**: Sets of attribute-value pairs

  (subject, news), (topic, travel), (date, 21.2.2011), …

Content-Based Matching II

- Publish(data, metadata) → Publish(data, list(attribute-value))
- Subscribe(interest) → Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

  subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

  publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

  publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

  publish("stock = IBM", "price = 180.0", "market = NYSE")

  publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	**Tree-structured data**	**Graph-structured data**	**Unstructured data**	**Regular languages**	**Relational model**
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) → Publish(data, list(attribute-value))
- Subscribe(interest) → Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	**Tree-structured data**	**Graph-structured data**	**Unstructured data**	**Regular languages**	**Relational model**
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) → Publish(data, list(attribute-value))
- Subscribe(interest) → Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

✓ publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	**Tree-structured data**	**Graph-structured data**	**Unstructured data**	**Regular languages**	**Relational model**
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) → Publish(data, list(attribute-value))
- Subscribe(interest) → Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

✓ publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

✗ publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	**Tree-structured data**	**Graph-structured data**	**Unstructured data**	**Regular languages**	**Relational model**
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) → Publish(data, list(attribute-value))
- Subscribe(interest) → Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

✓ publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

✗ publish("stock = IBM", "price = 180.0", "market = NYSE")

✓ publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	**Tree-structured data**	**Graph-structured data**	**Unstructured data**	**Regular languages**	**Relational model**
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Language Categories

Simple

- Channel-based
  - OMG CORBA Event Service, group communication …
- Topic-based (Subject-based)
  - WS Notifications, OMG Data Dissemination Service …
- Type-based
  - OMG Data Dissemination Service (partially), …
- Content-based
  - The PADRES ESB (see below), …

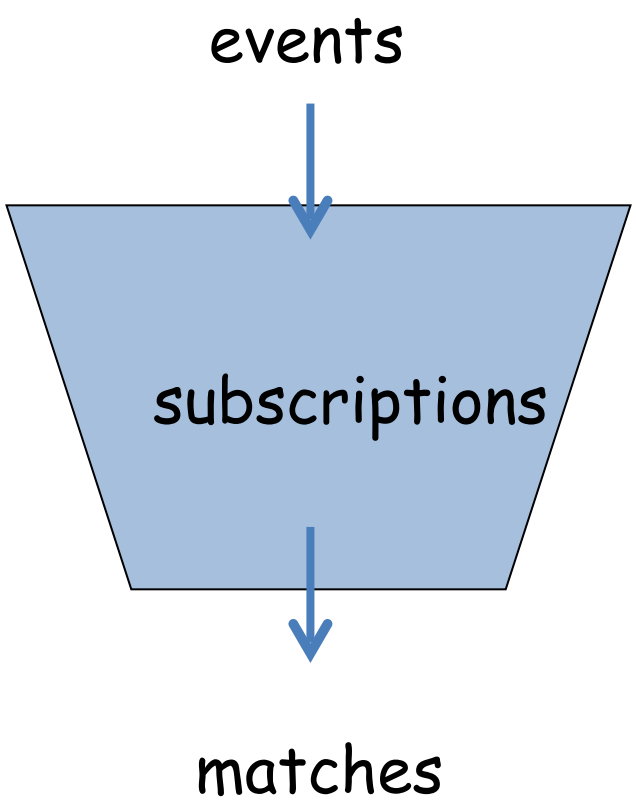
Stateless

Stateful

- State-based
  - Many rule-based systems & tuple spaces
- Subject Spaces (state persistent)
  - Still, a research idea, concept & prototype

Complex

Matching problem: Given an event, *e*, and a set of subscriptions, *S*, determine all subscriptions, $s \in S$, that match *e*.



CONTENT-BASED MATCHING ALGORITHMS

Matching Algorithms

- Example algorithms
  - **Counting algorithm** [origins unknown to me; guess: 1970s]
  - *Clustering algorithm [Fabret, Jacobsen et al., 2001]*

- Both are **two-phased** matching algorithms
  1. Match all predicates (**predicate matching phase**)
  2. Match subscriptions from results of Phase 1 (**subscriptions matching phase**)

- Let us look at both phases separately
  - Predicate matching
  - Subscription matching

Two-Phased Matching

- **Decompose** subscriptions into predicates

s_1 $(\text{subject} \overset{p1}{=} \text{news}) \wedge (\text{topic} \overset{p2}{=} \text{travel}) \wedge (\text{date} \overset{p3}{>} 29.11.2011)$

s_2 $(\text{subject} \overset{p1}{=} \text{news}) \wedge (\text{topic} \overset{p4}{=} \text{stock}) \wedge (\text{date} \overset{p5}{>} 30.11.2011)$

- **Convert** subscriptions to reference predicates

  S1: P1 ∧ P2 ∧ P3

  S2: P1 ∧ P4 ∧ P5

Break down subscriptions into predicates, store and match them individually

PHASE 1: PREDICATE MATCHING

Predicate Matching Problem

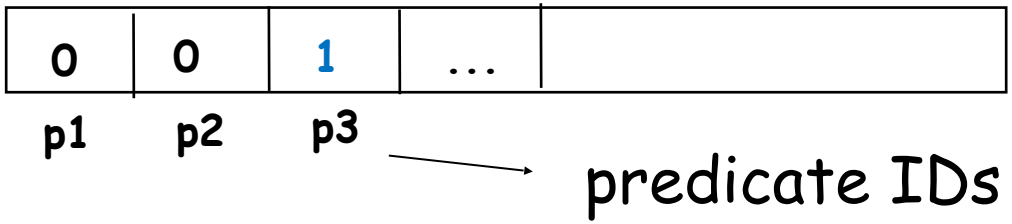
- Given a set ***P*** of predicates and a publication ***e***, ***identify all predicates p of P which evaluate to true under e***.
- Example:

e = {…, (price, 5), (color, white), …}

p1: price > 5; p2: color = red; p3: price < 7

p1 is false p2 is false p3 is true

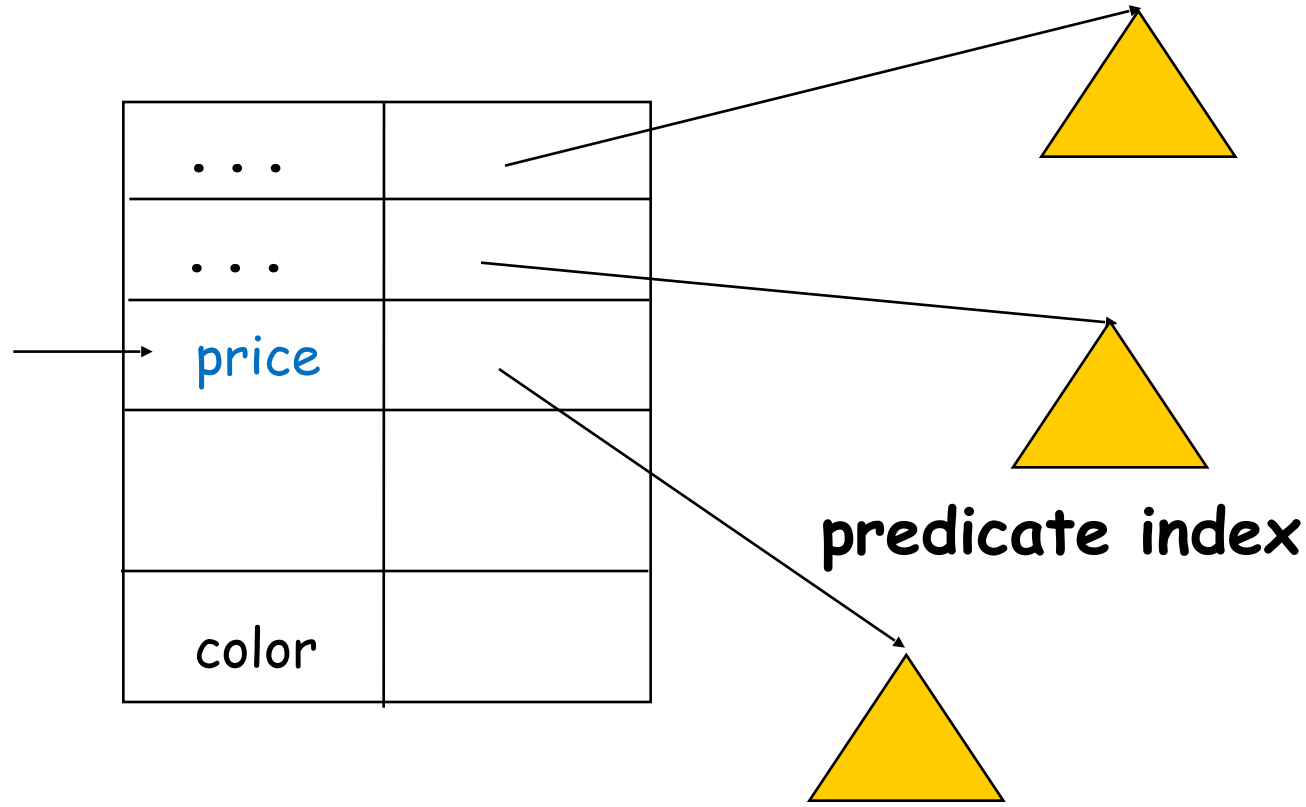
Predicate bit vector:



Predicate Matching:
Top-level Data Structure

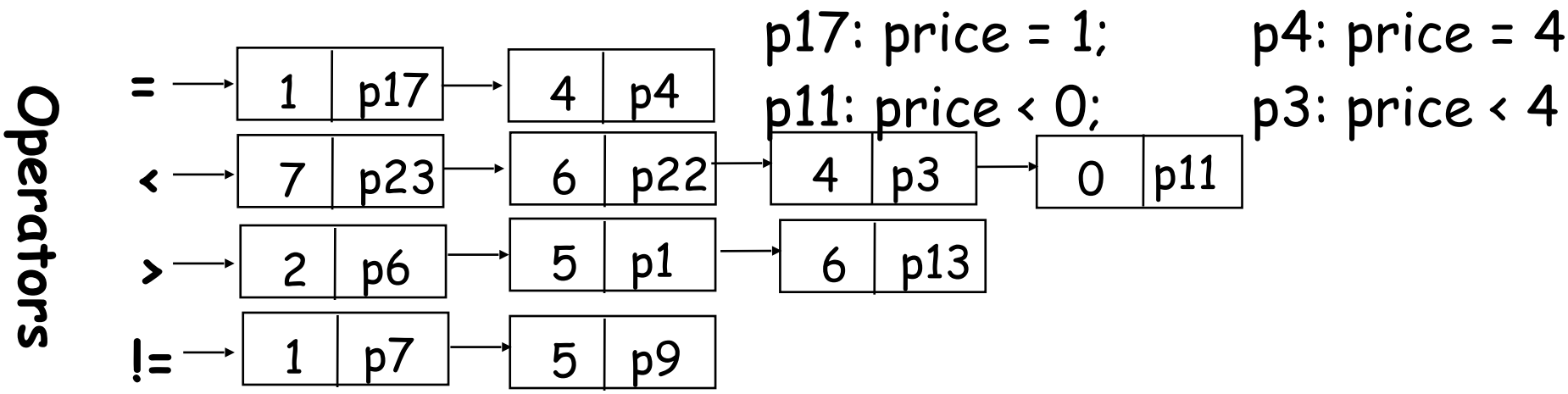
- Hash table on attribute name

e = {…, (**price**, 5), …}



Predicate Index: General Purpose Data Structure

e = {…, (**price**, 5), …}



- One ordered linked list per operator
- Insert, delete, match are O(n)-operations (per attribute name in ***e*** and per operator)
- Alternatively, use a B-tree or B+-tree etc.

Finite Predicate Value Domain Types

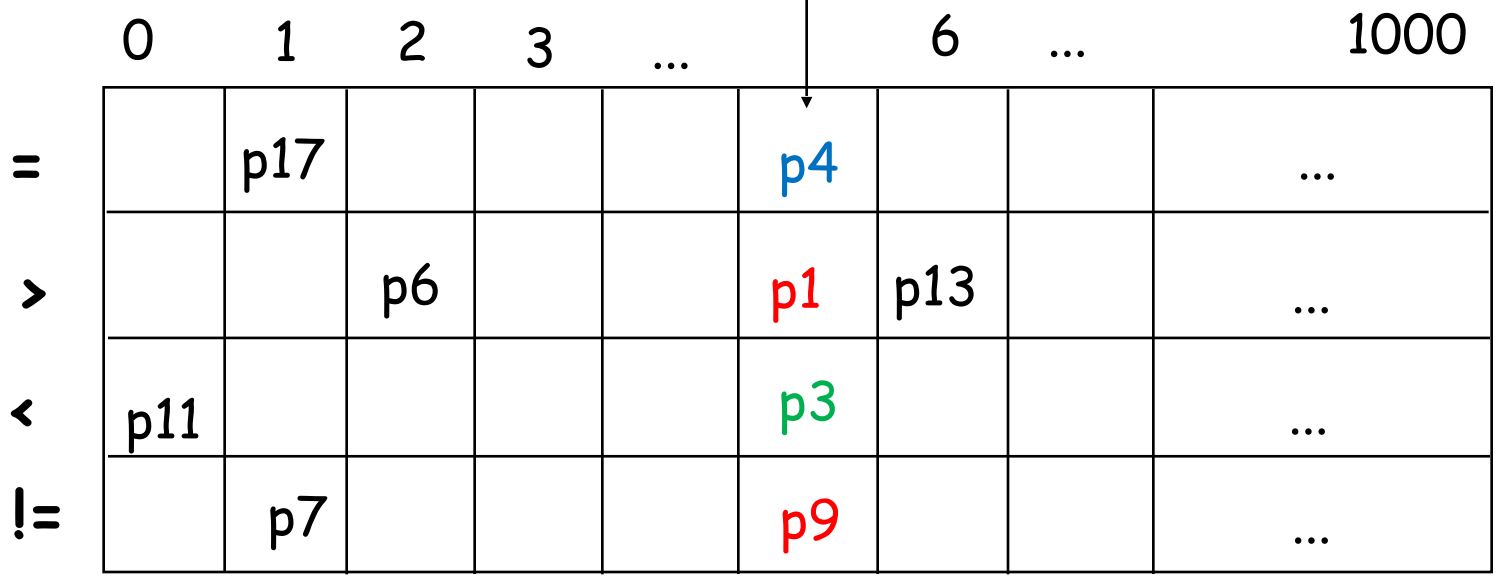
- Countable domain types with small cardinality
  - Integer intervals
  - Collections (enums)
  - Set of tags (e.g., in XML)

- Examples
  - Price : [0, 1000], models variety of prices
  - Color, city, state, country, size, weight
  - All tags defined in a given XML schema
  - Predicate domain often context dependant, but limited in size
    - Prices of cars vs. prices of groceries

Predicate Matching For Finite Domains

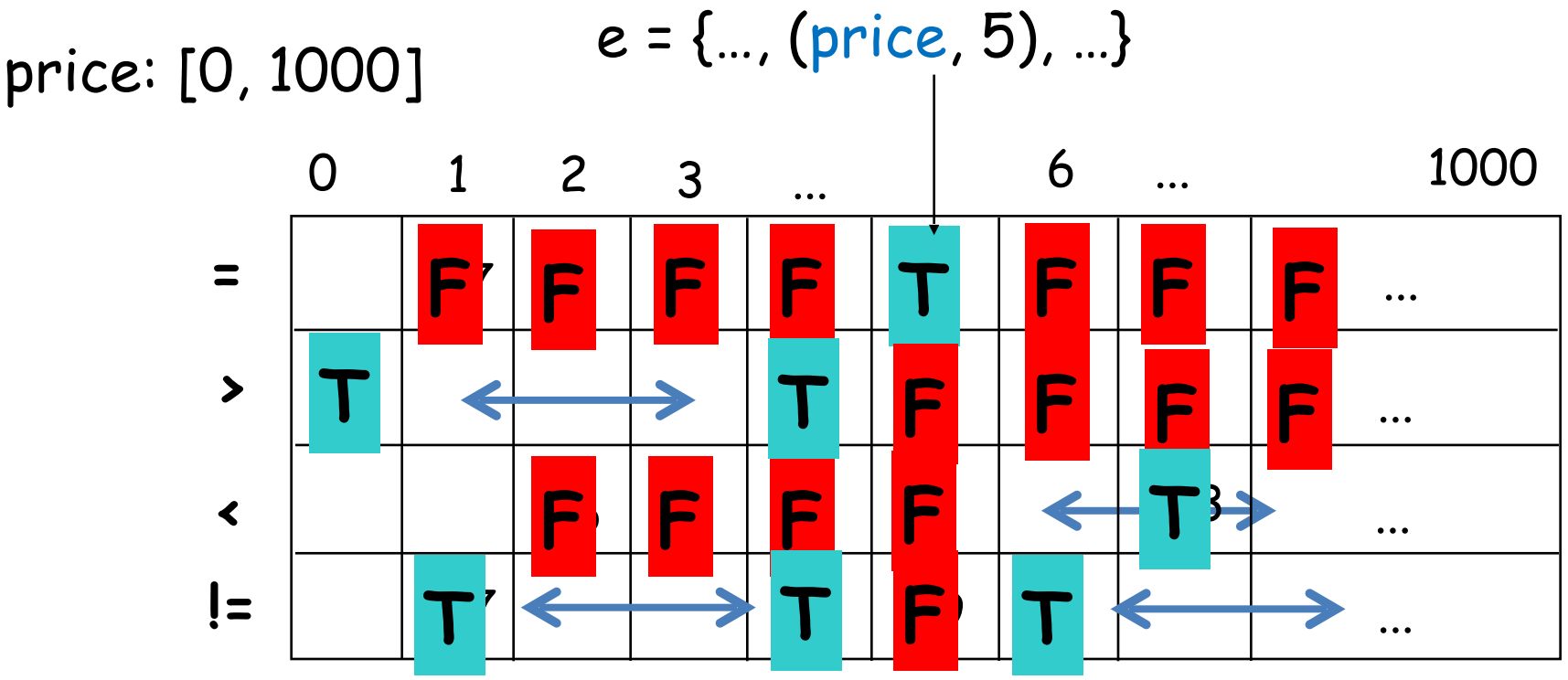
price: [0, 1000]

e = {…, (price, 5), …}



p1: price > 5; p3: price < 5; p4: price = 5; p9: price != 5

Predicate Matching Symmetries



PHASE 2: SUBSCRIPTION MATCHING

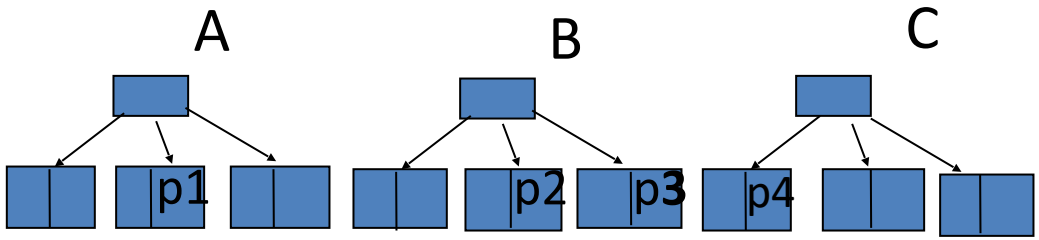
Subscription Matching: Counting Algorithm

- Subscriptions consist of a set of predicates
  - S1: (2< **A**<4) & (**B**=6) & (**C** >4) ⇒ p1: (2<A<4), p2: (B=6), p3: (C>4)
  - S2: (2< **A**<4) & (**C**=3) ⇒ p1: (2<A<4), p4: (C=3)
- Subscription matches the event if all its predicates are satisfied
- **Idea**: Count the number of satisfied predicates per subscription and compare with actual number

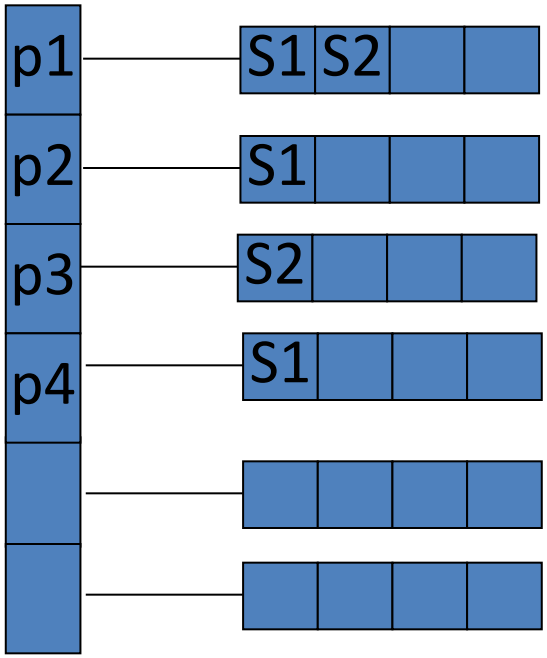
Counting Algorithm: Subscription Insertion

S1: p1, p2, p4

S2: p1, p3



Predicate vector



predicate-to-subscription association

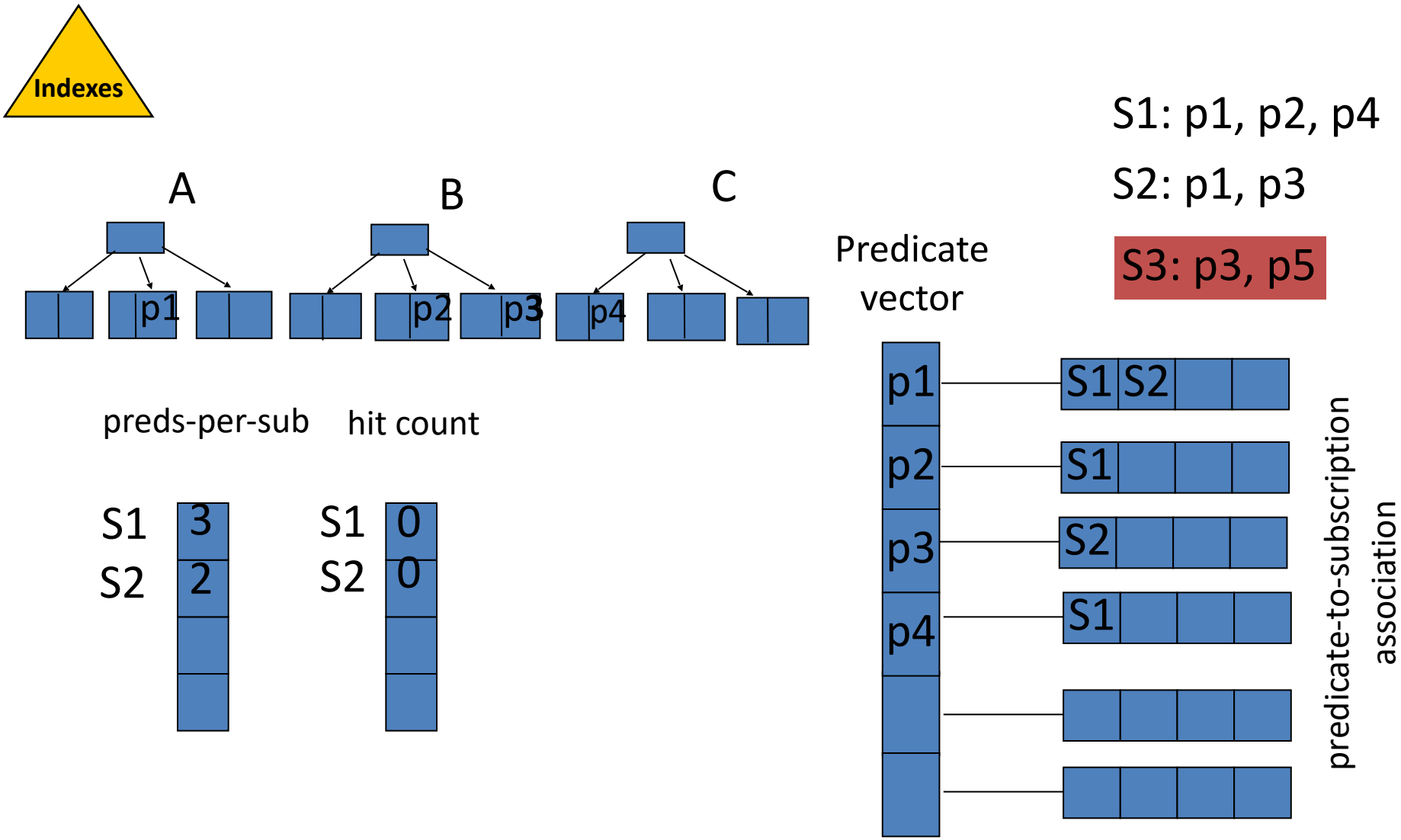
preds-per-sub

S1	3
S2	2

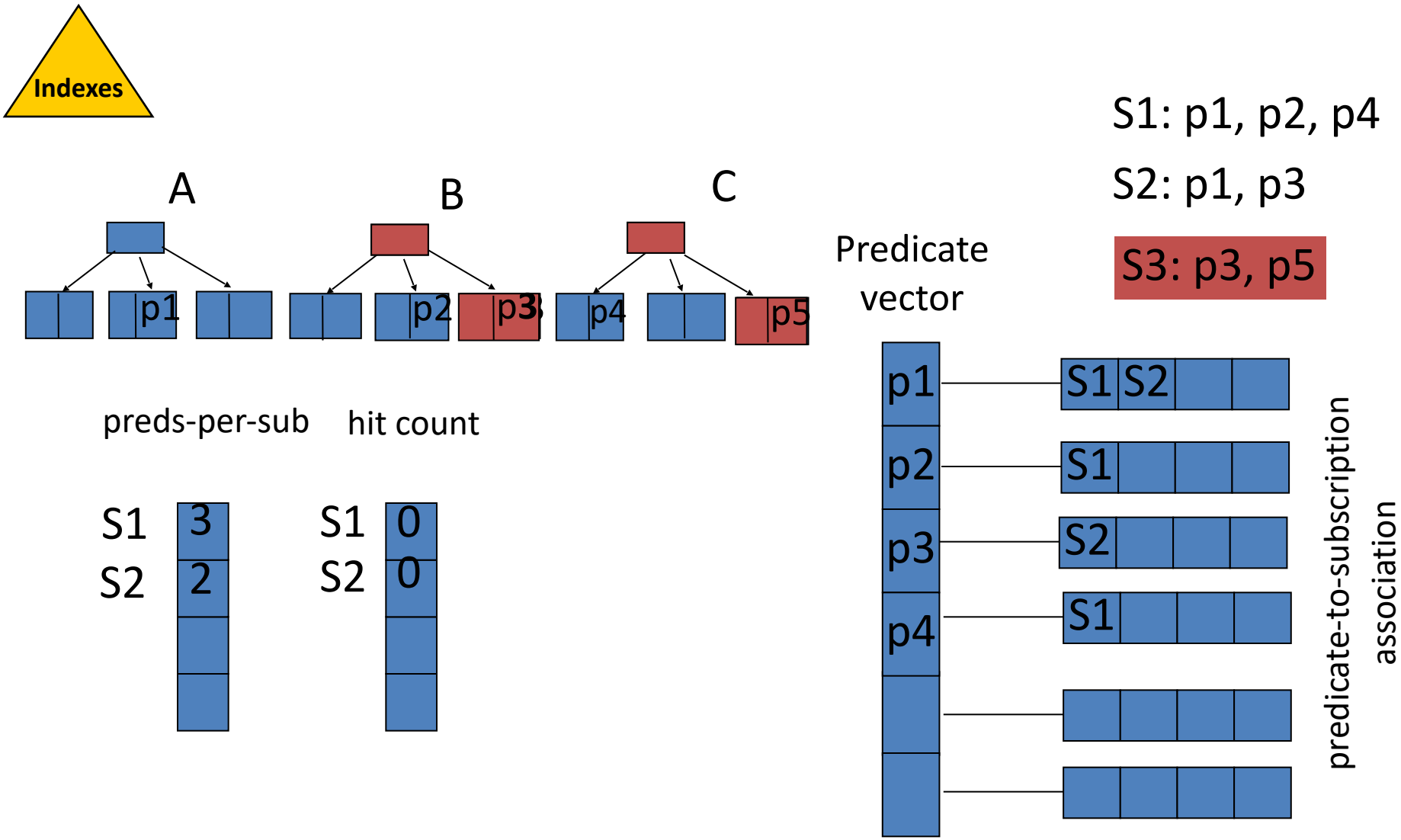
hit count

S1	0
S2	0

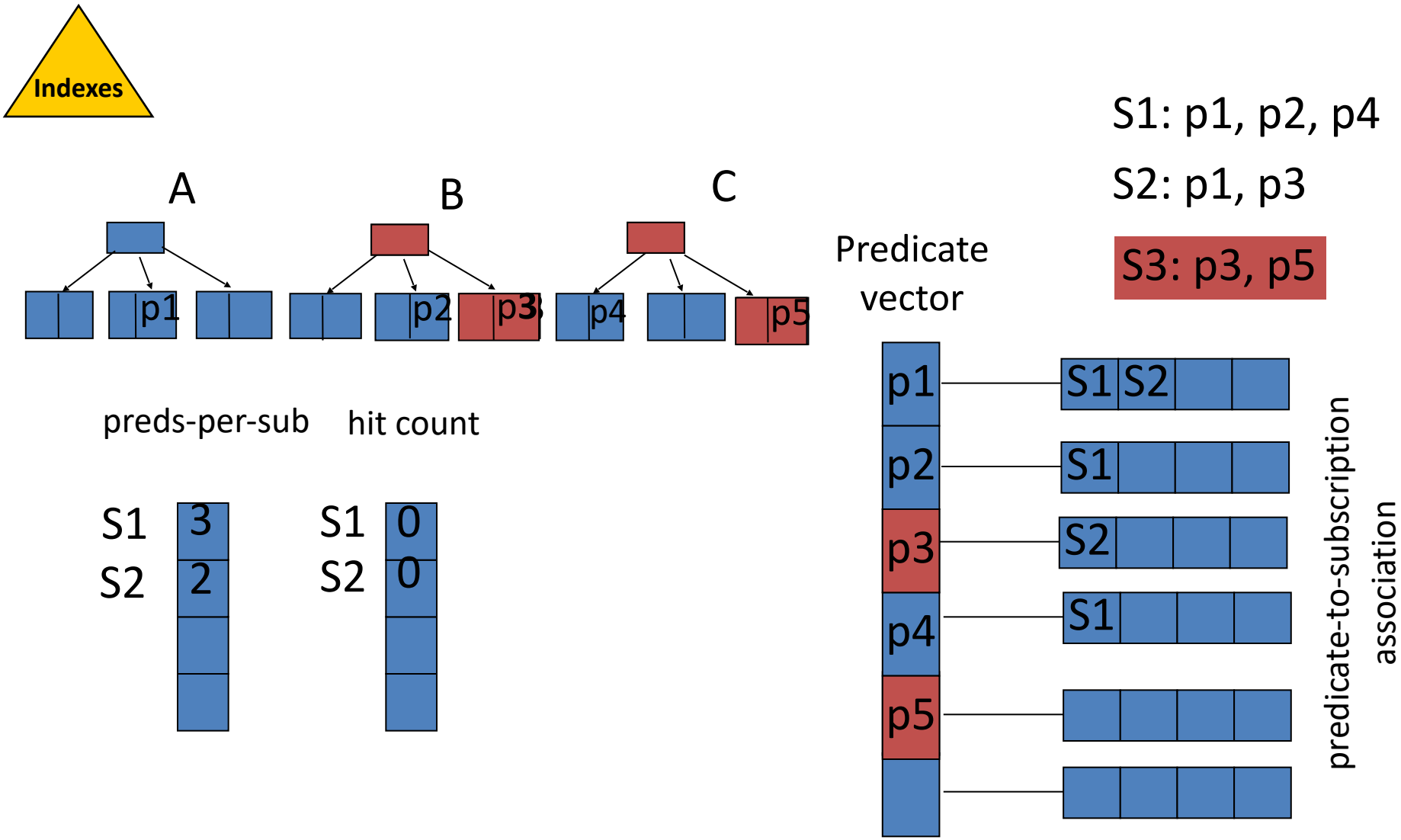
Counting Algorithm: Subscription Insertion



Counting Algorithm: Subscription Insertion



Counting Algorithm: Subscription Insertion

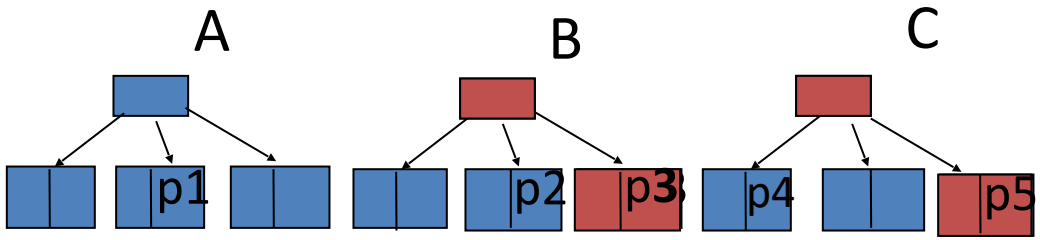


Counting Algorithm: Subscription Insertion

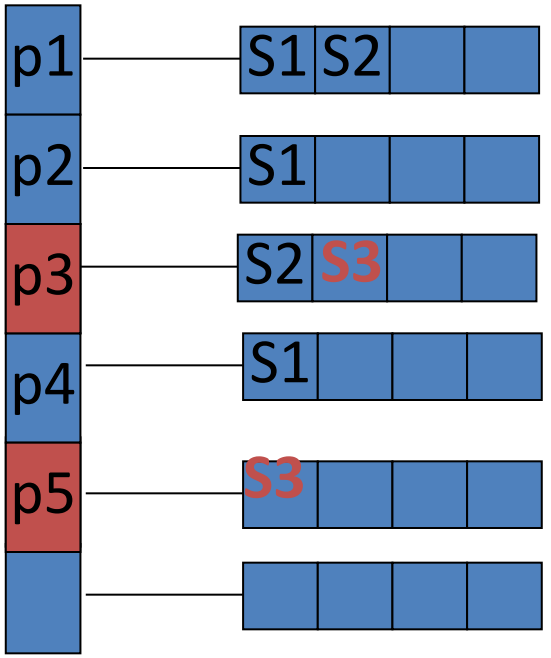
S1: p1, p2, p4

S2: p1, p3

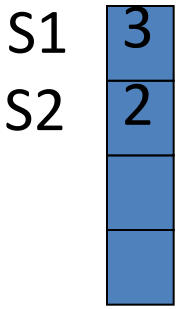
S3: p3, p5



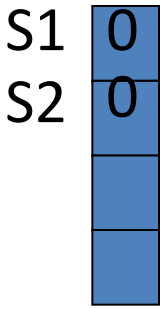
Predicate vector



preds-per-sub



hit count

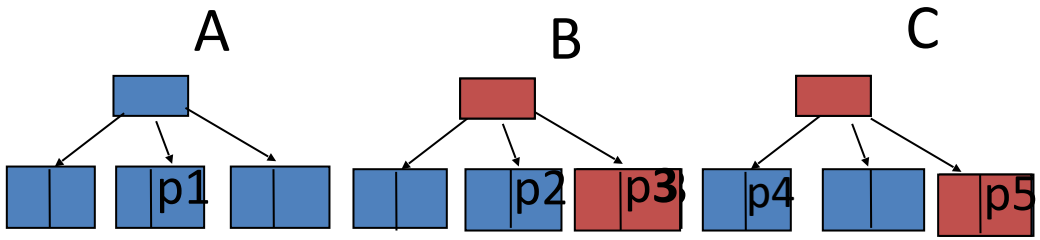


Counting Algorithm: Subscription Insertion

S1: p1, p2, p4

S2: p1, p3

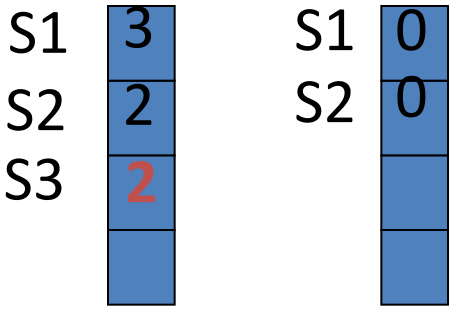
S3: p3, p5

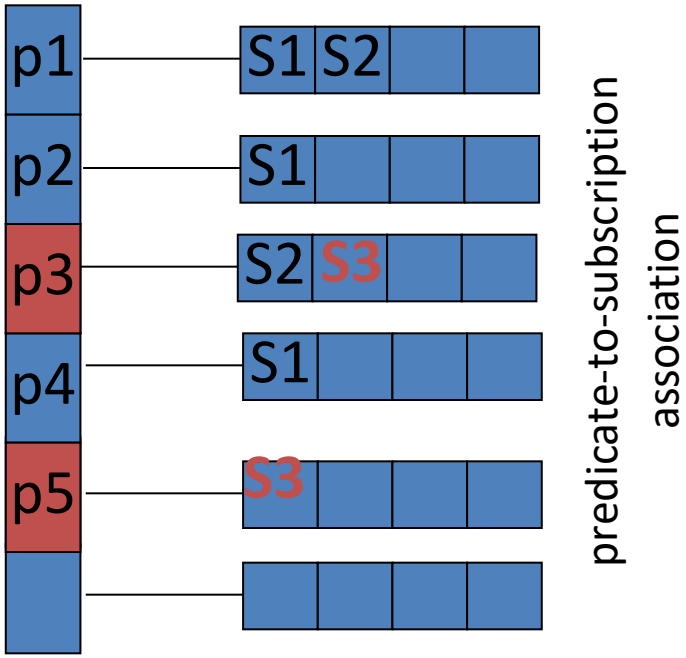


Predicate vector

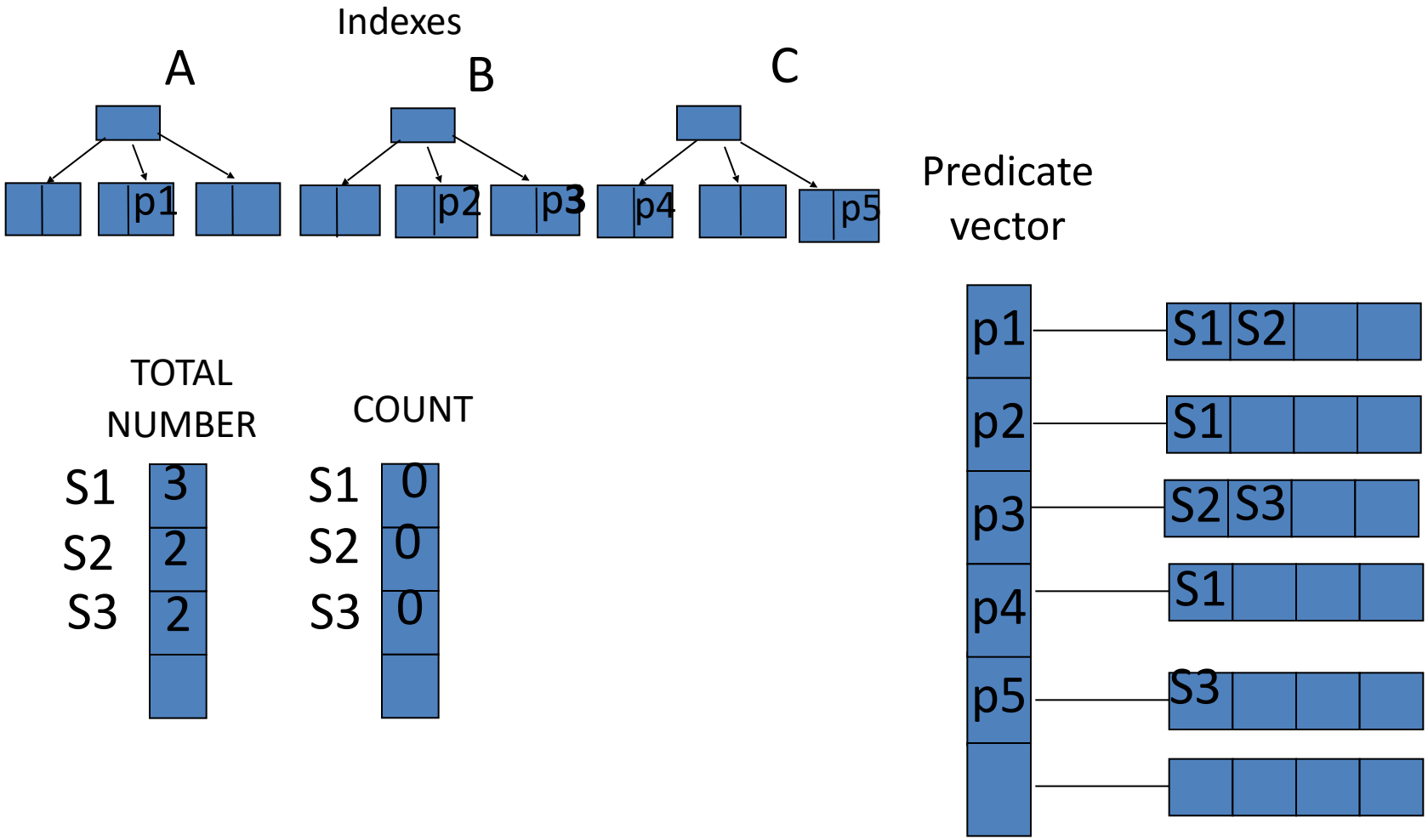
preds-per-sub

hit count

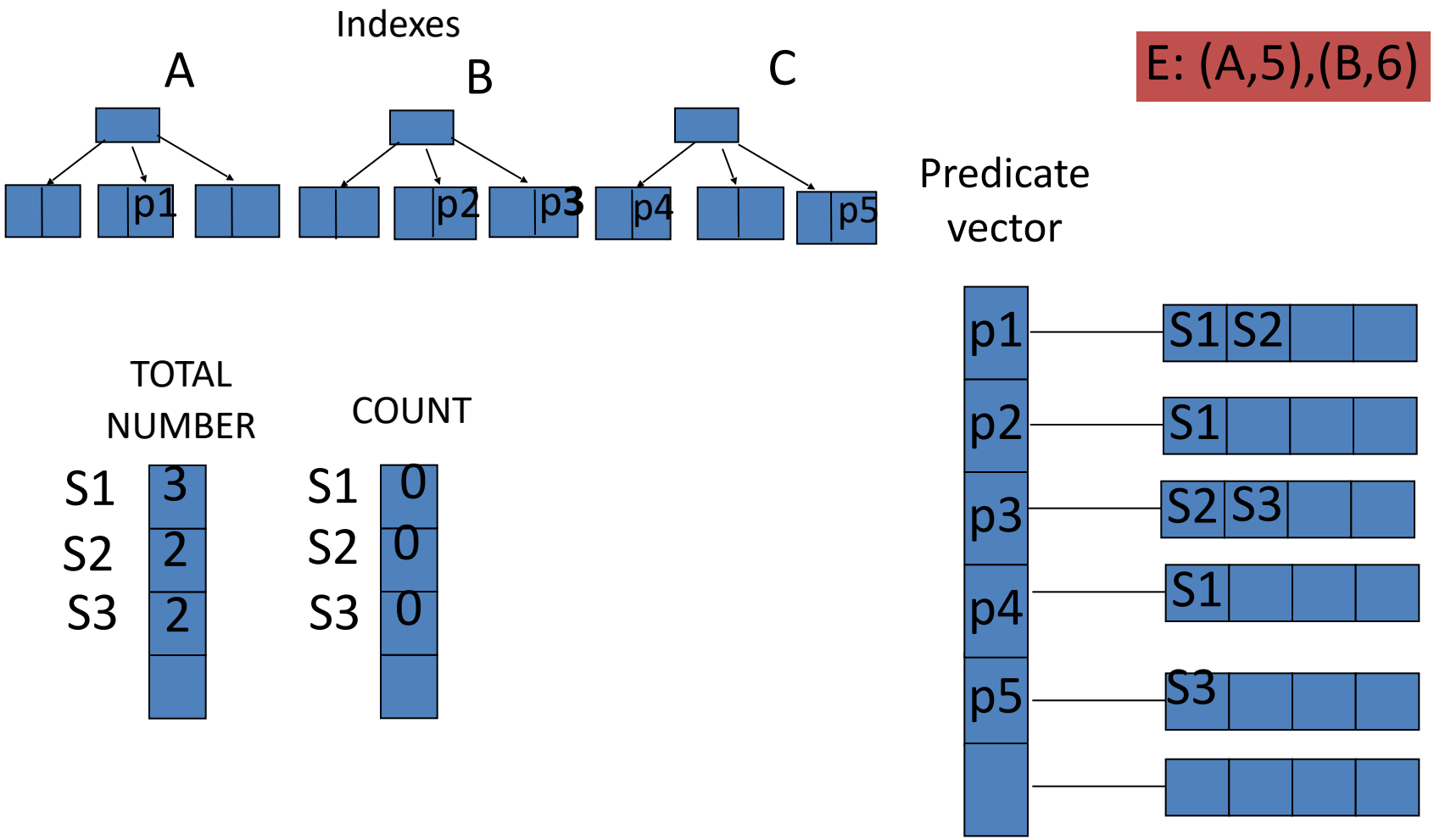




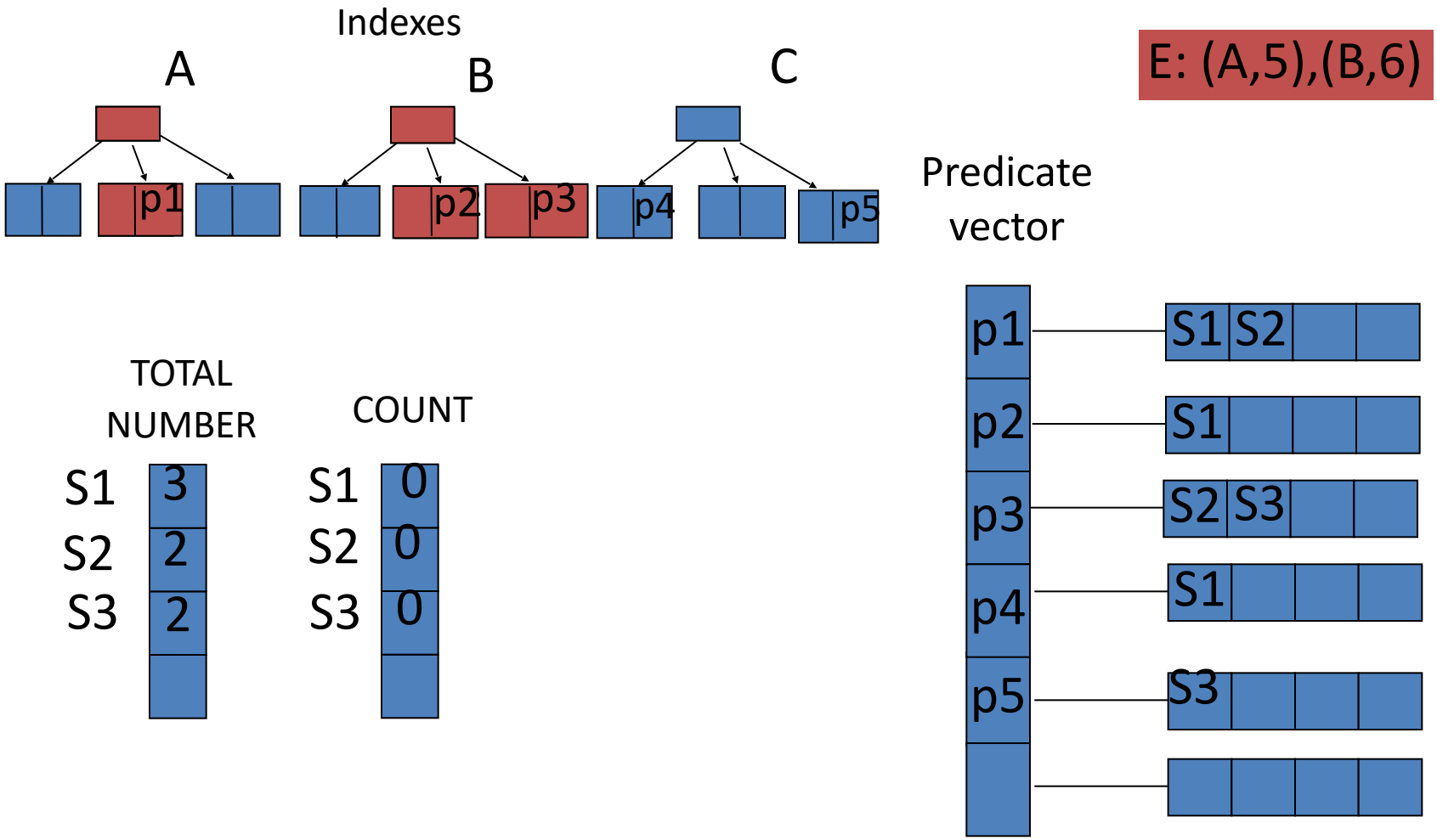
Counting Algorithm: Event Matching



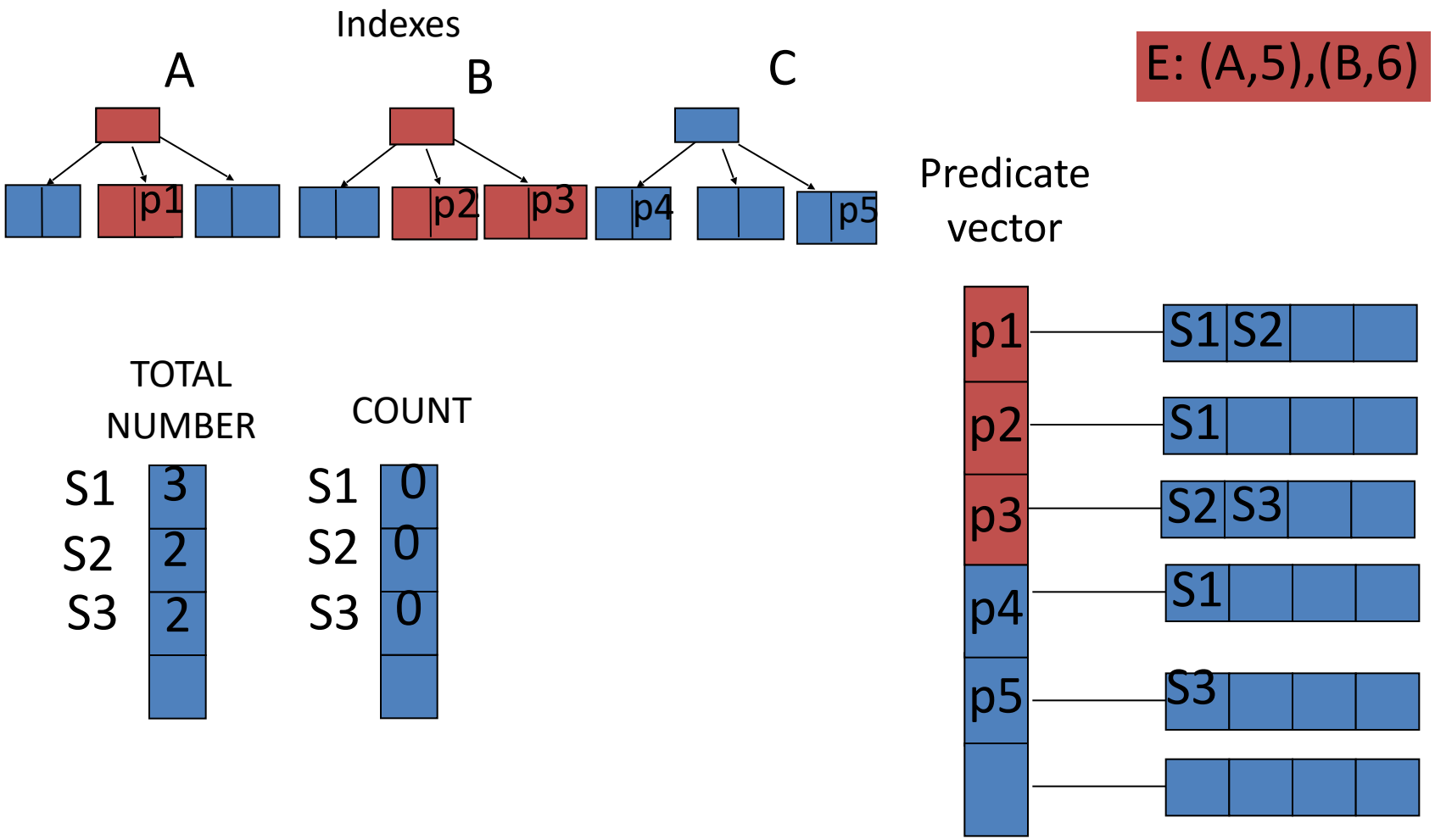
Counting Algorithm: Event Matching



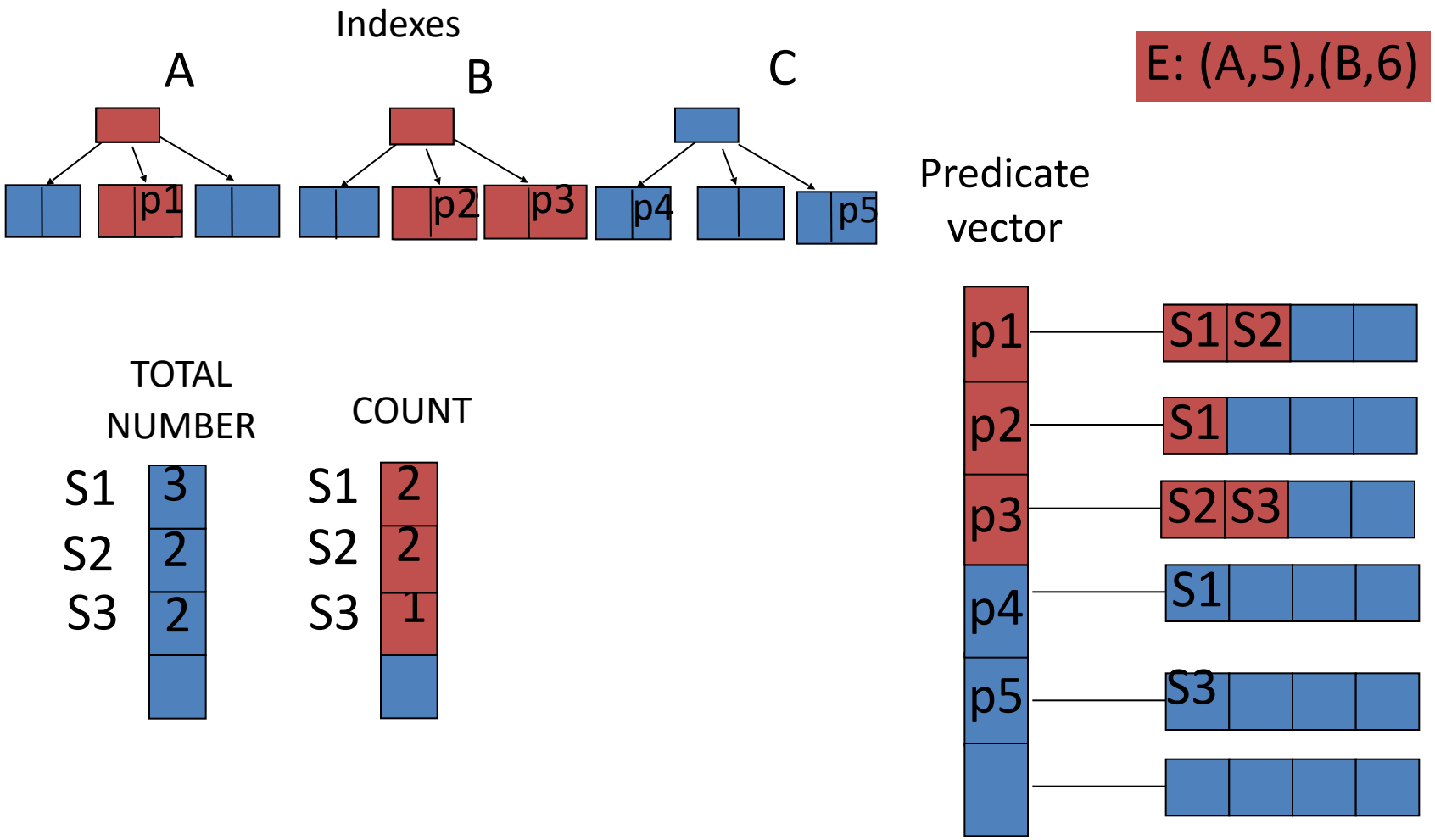
Counting Algorithm: Event Matching



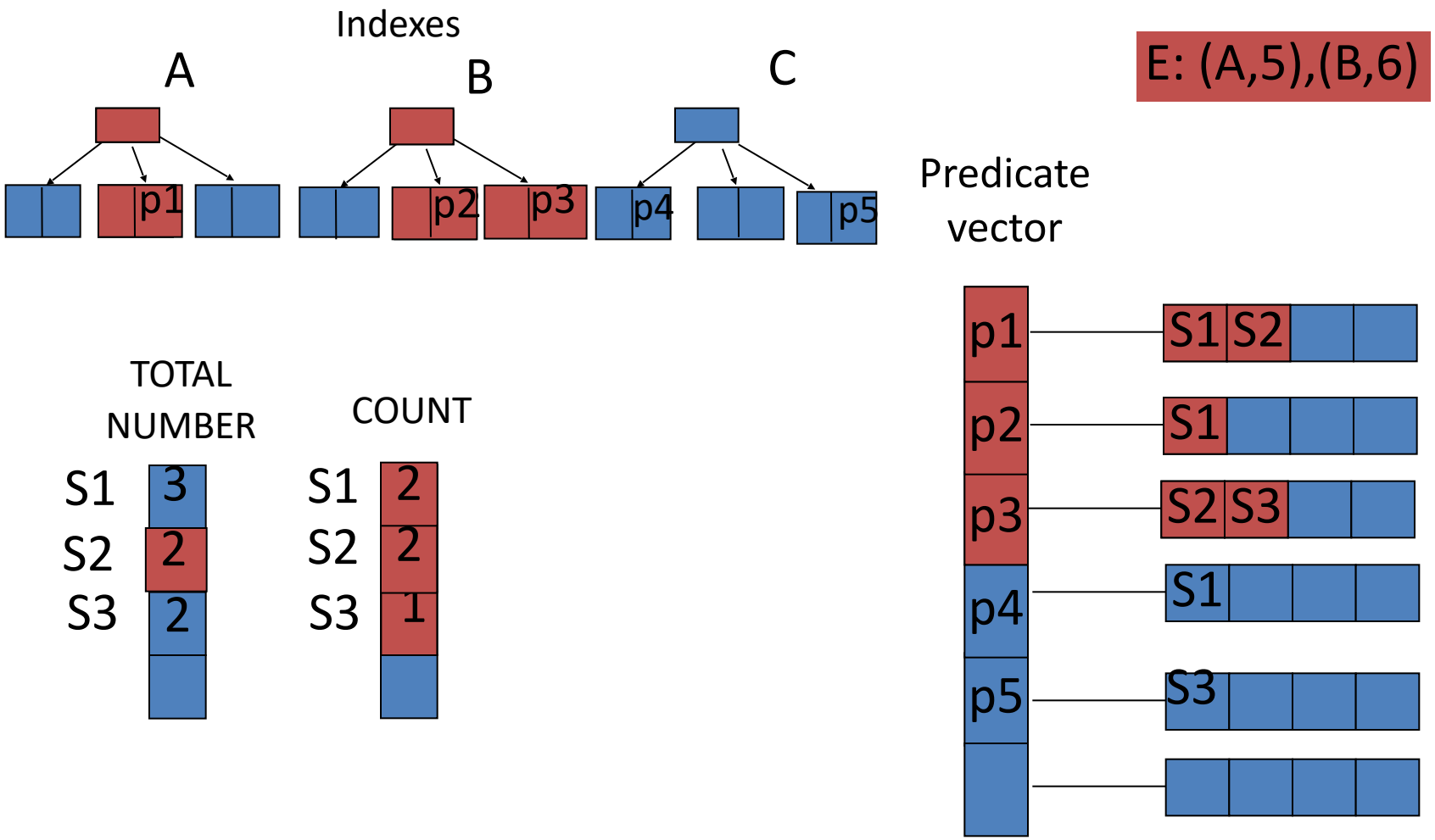
Counting Algorithm: Event Matching



Counting Algorithm: Event Matching



Counting Algorithm: Event Matching



Matching Models Summary

- **Matching model:** The language used to express pubs and subs in order to match them.
- **Matching problem**: Given a publication, p, and a set of subscriptions, *S*, determine all subscriptions, $s \in S$, that match *e*.
- **Topic-based** vs. **content-based** matching
- **Tradeoff**: computation vs. communication
- **Two-phase matching** is an efficient technique for quickly finding matches in content-based model
- **Counting** algorithm

Pixabay.com

PUBLISH/SUBSCRIBE

ROUTING MODELS

Pub/Sub Architecture

- Pub/Sub Middleware
  - Comprised of a dedicated **pub/sub middleware**
  - Receives all operations
  - Stores subscriptions
  - Performs **publication matching and dissemination**
- Infrastructure-less
  - No dedicated middleware
  - Clients (publishers and subscribers) perform the tasks
  - **Peer-to-peer model**

Centralized vs. Distributed

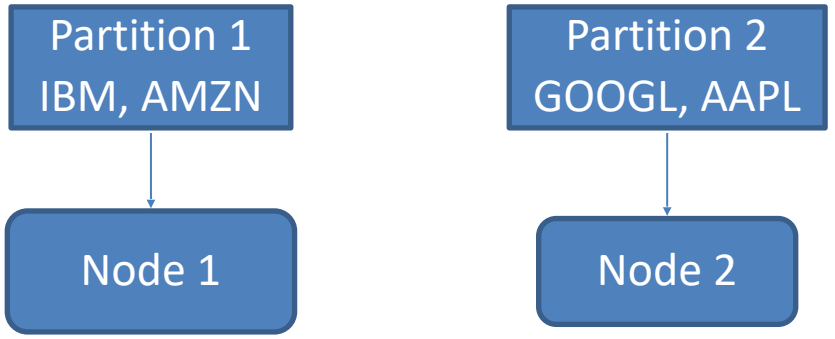
- Centralized
  - A single pub/sub entity
  - Client-server interactions
  - CORBA, JMS, …
- **Decentralized**
  - Matching and dissemination tasks **distributed**
  - Either, multiple pub/sub entities (called **brokers**)
  - Or, P2P model without brokers (e.g., DHT)

RENDEZVOUS-BASED ROUTING

Rendezvous-based Routing

- The "publication space" is partitioned
- Each node is in charge of a partition
- Publications and subscriptions belonging to a partition are sent to the corresponding node

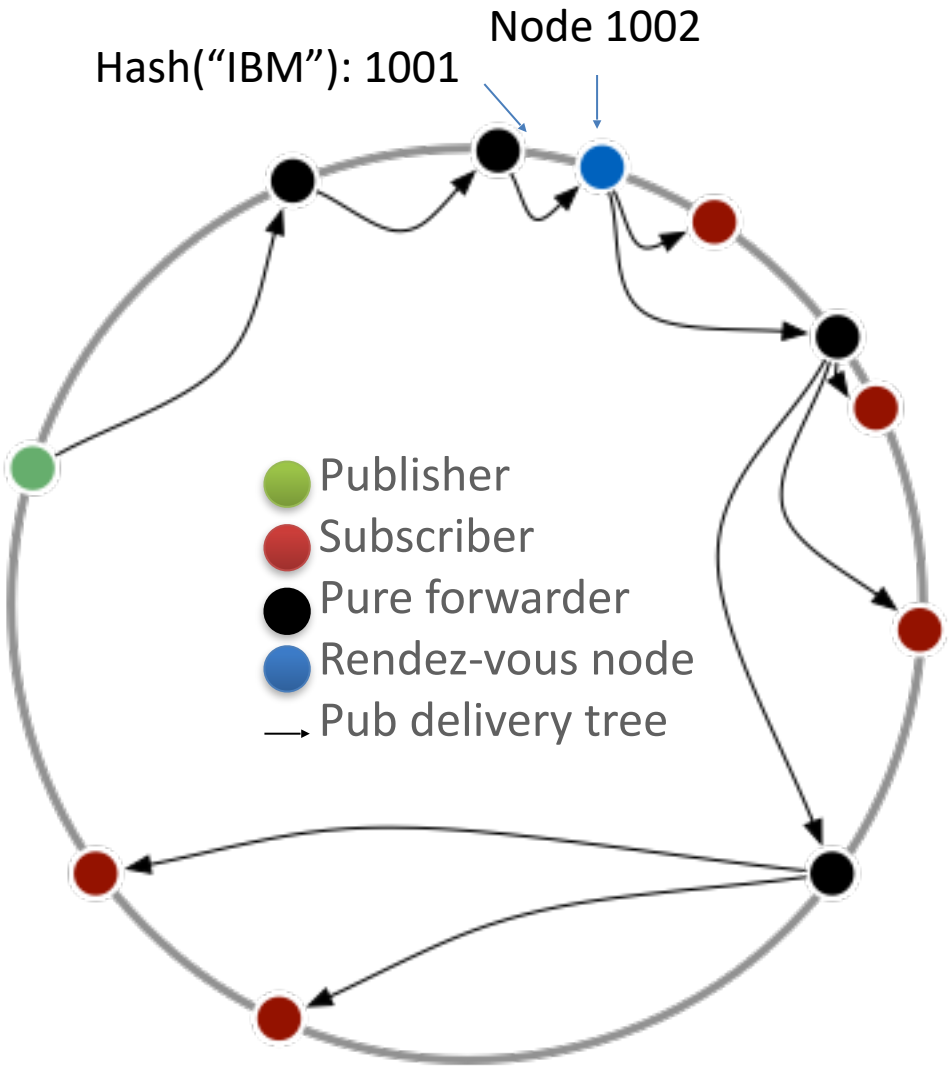
Example for topic-based:



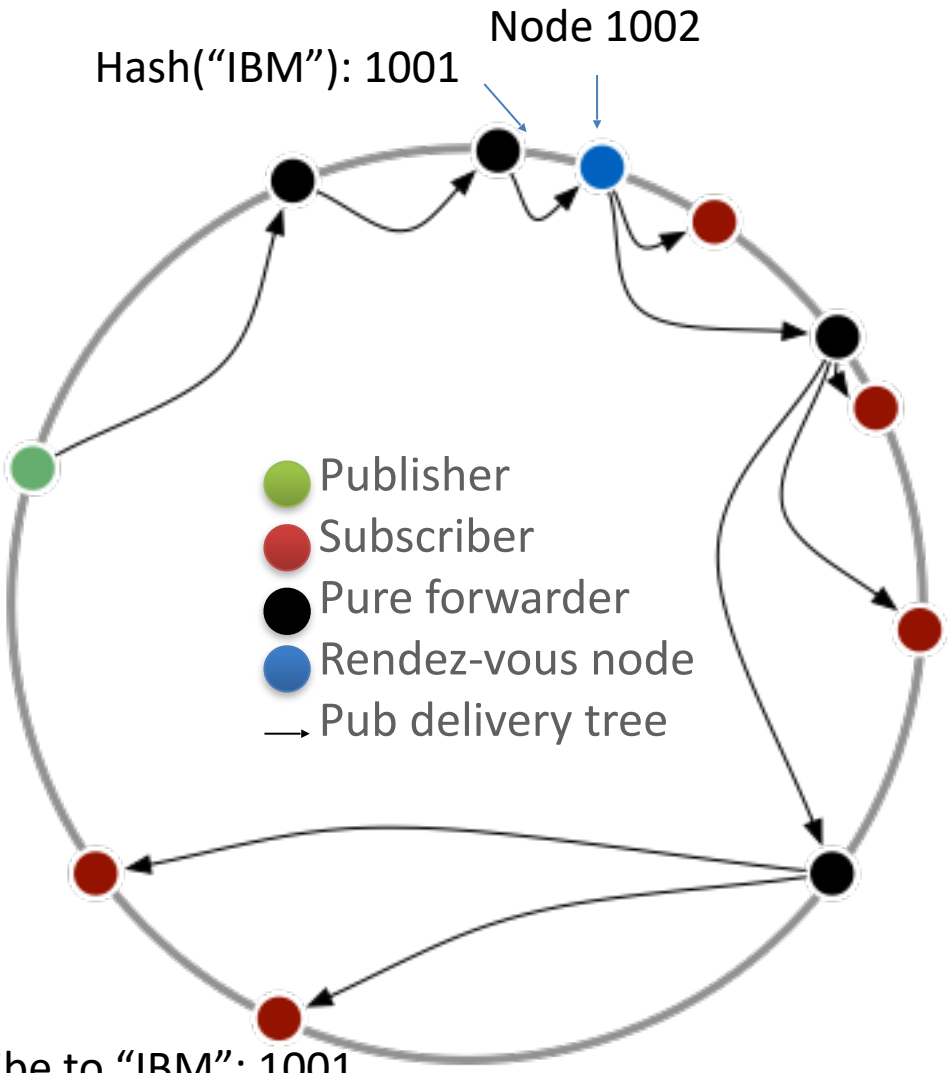
P2P Pub/Sub: Scribe (2001)

- Infrastructure-less, distributed, pub/sub network built on top of a P2P network
- Relies on a structured P2P DHT (Pastry)
- Supports topic-based pub/sub API on top of Pastry
- Topics are hashed in the DHT

Scribe Routing

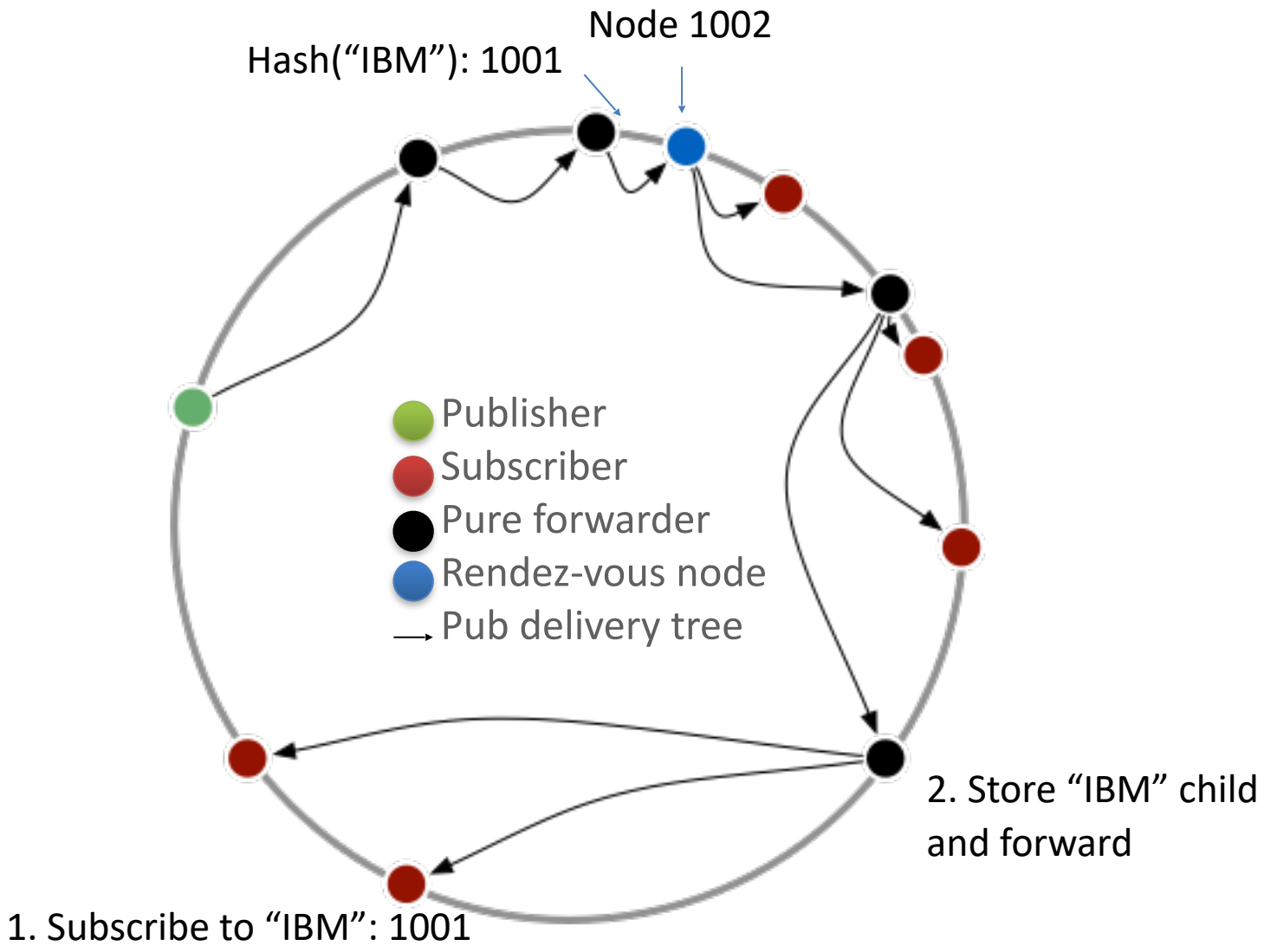


Scribe Routing

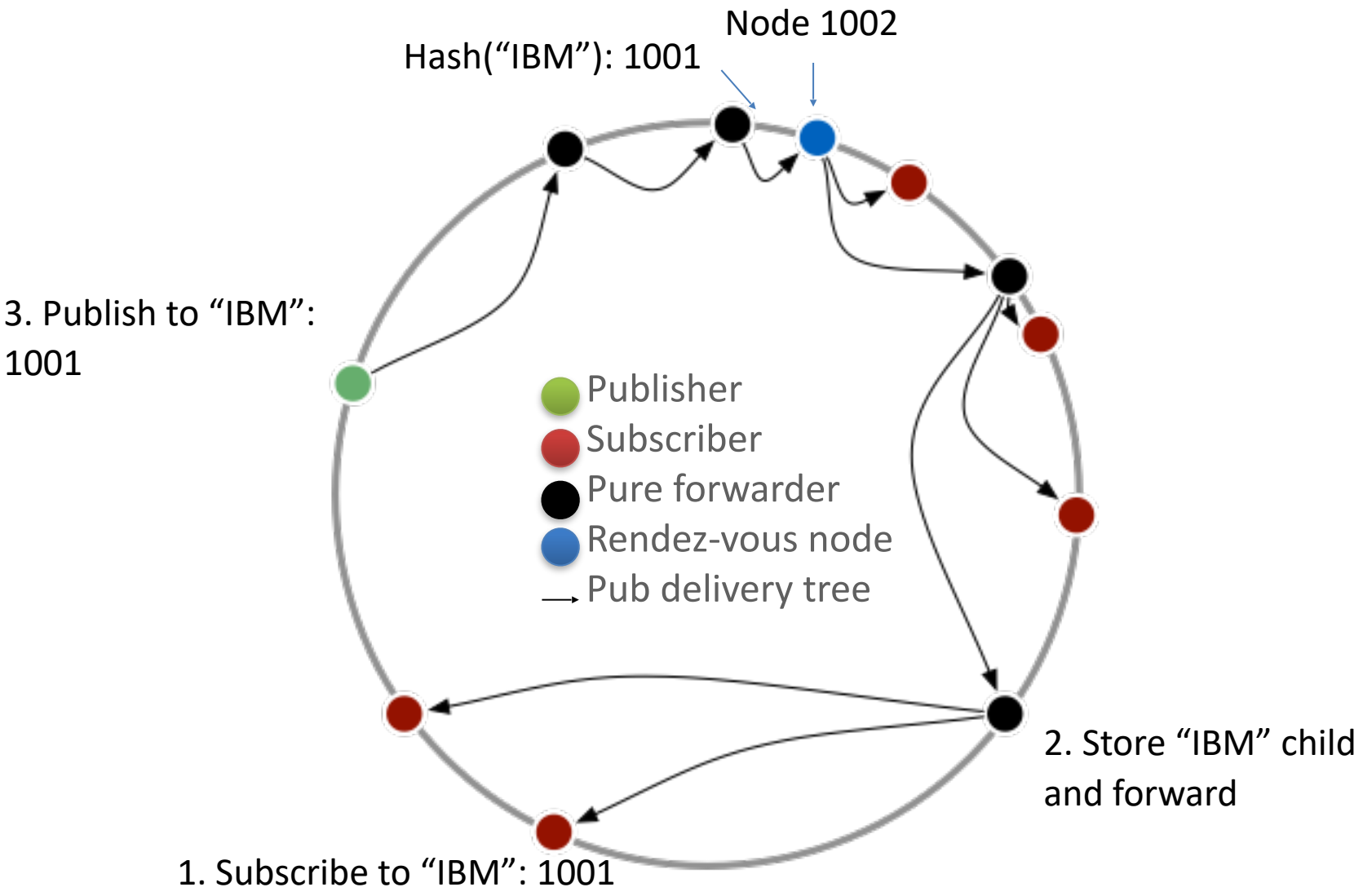


1. Subscribe to "IBM": 1001

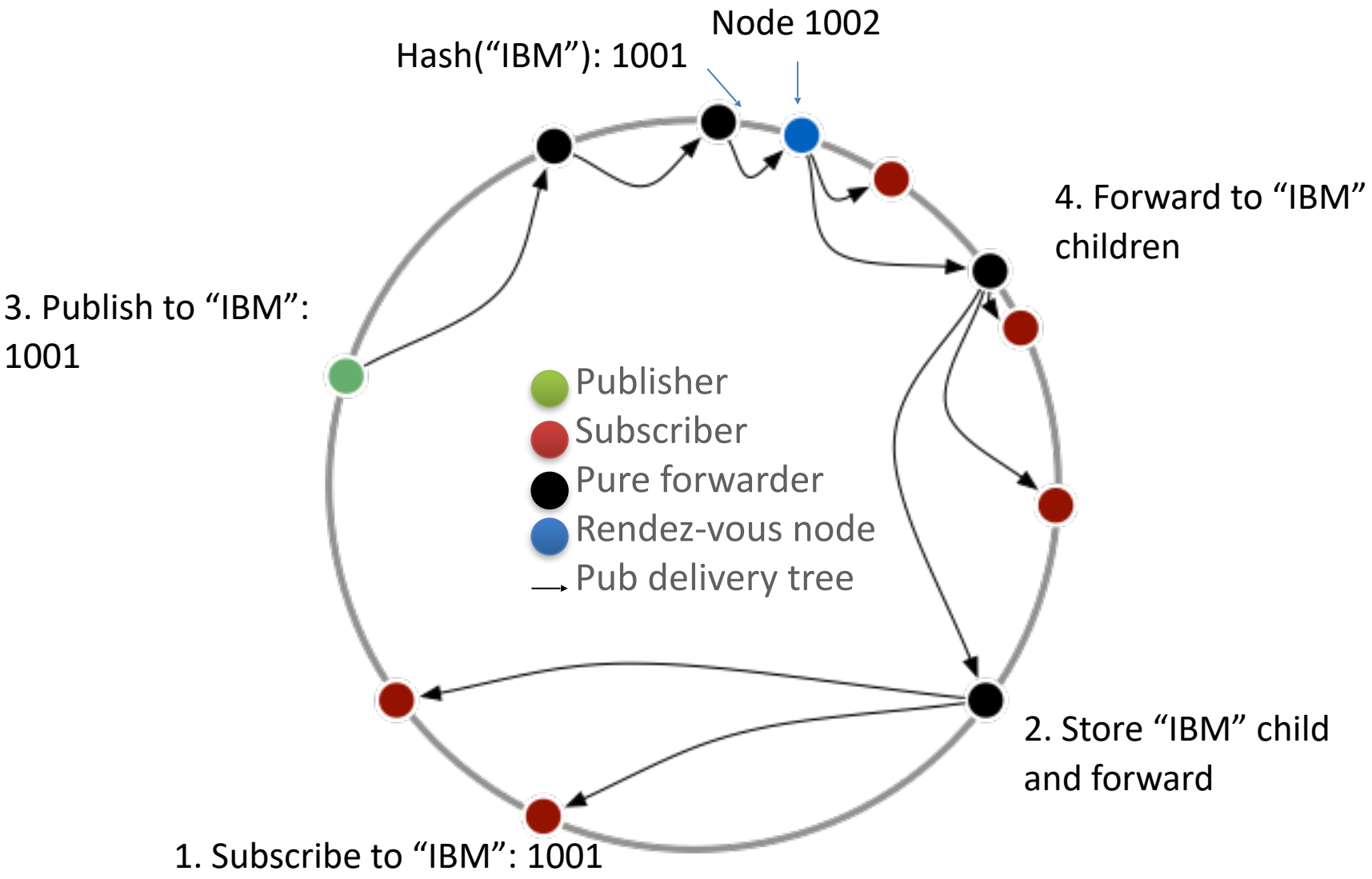
Scribe Routing



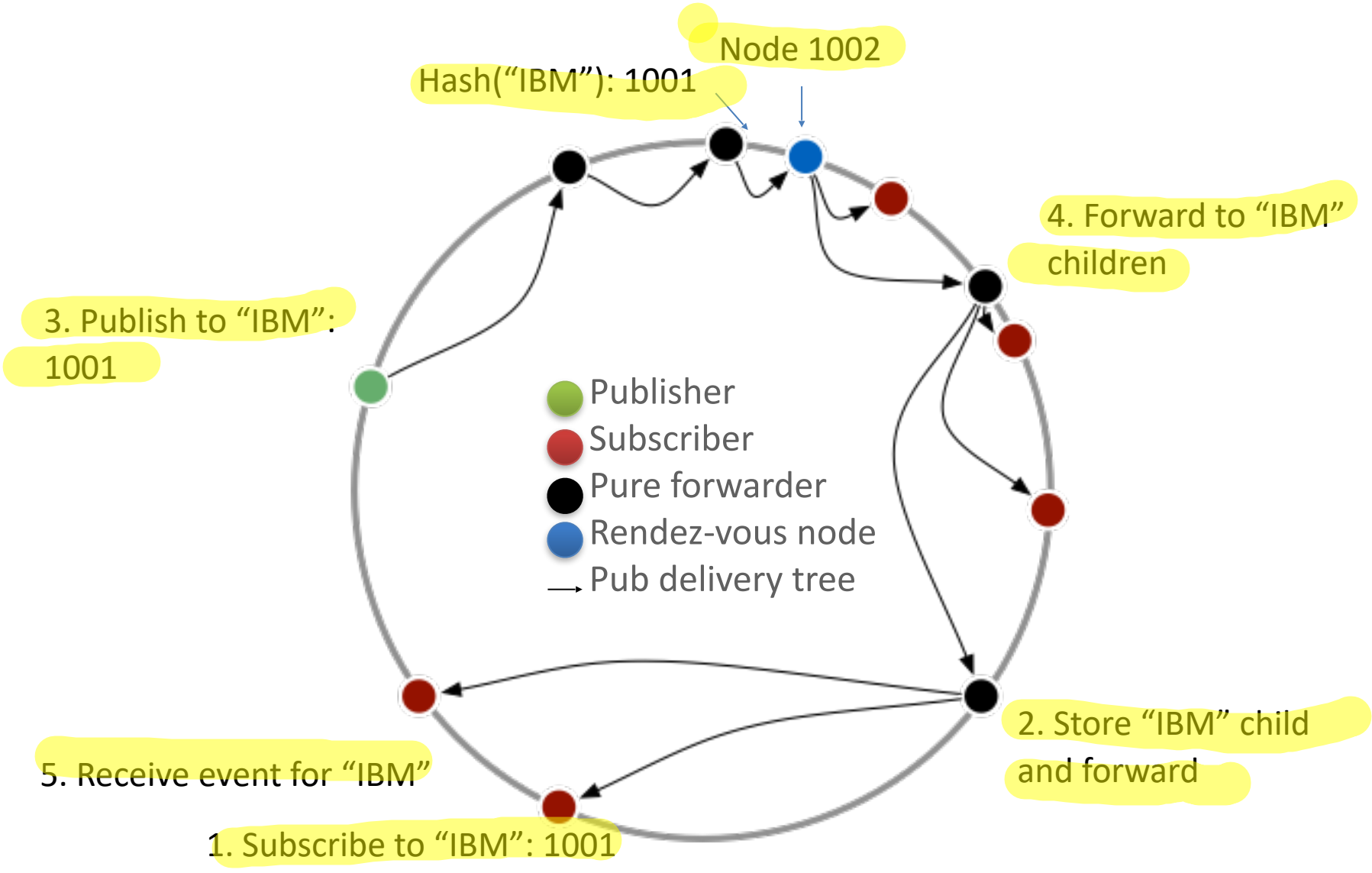
Scribe Routing



Scribe Routing



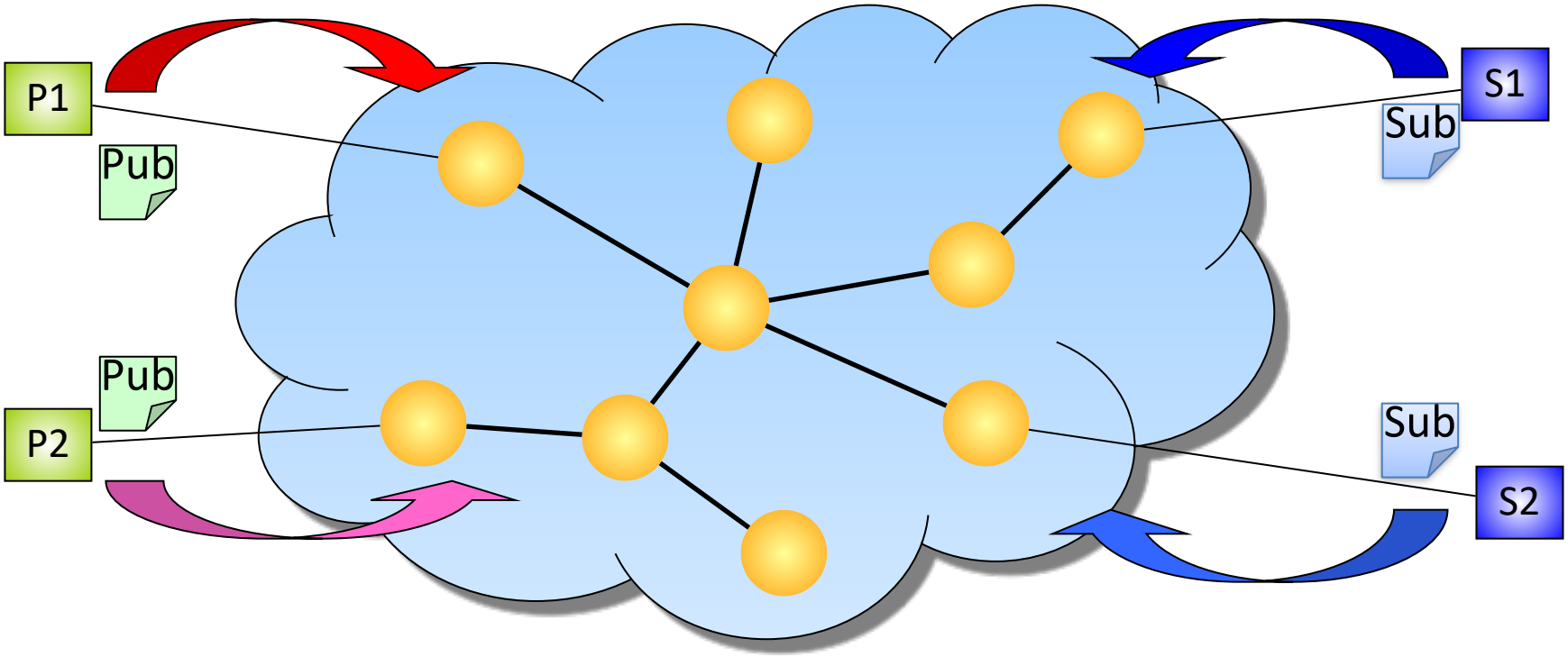
Scribe Routing



OVERLAY-BASED ROUTING

Overlay Broker Networks

- Messages flow based on overlay links
- Clients connect to any edge broker

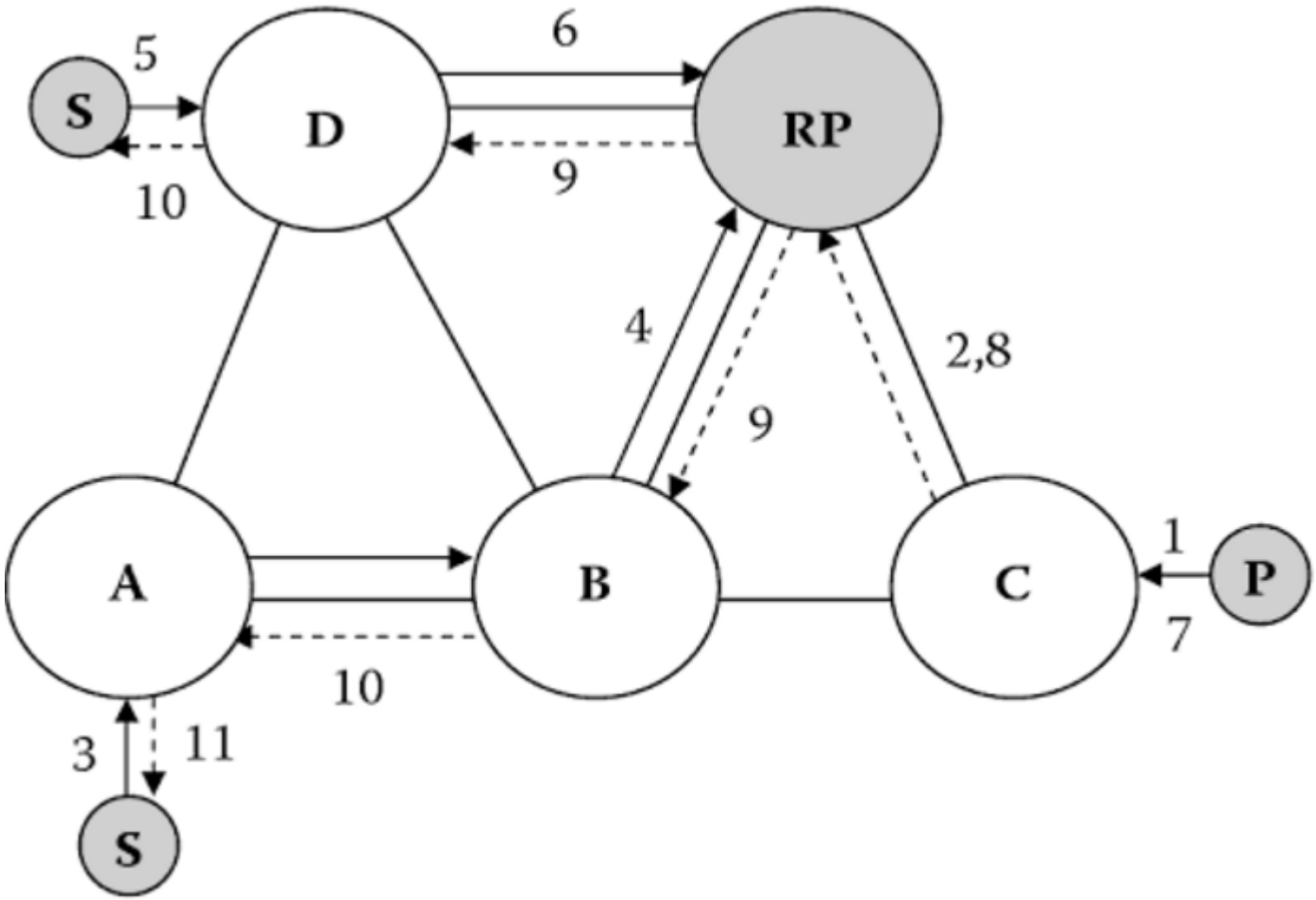


Matching and dissemination are performed inside the overlay

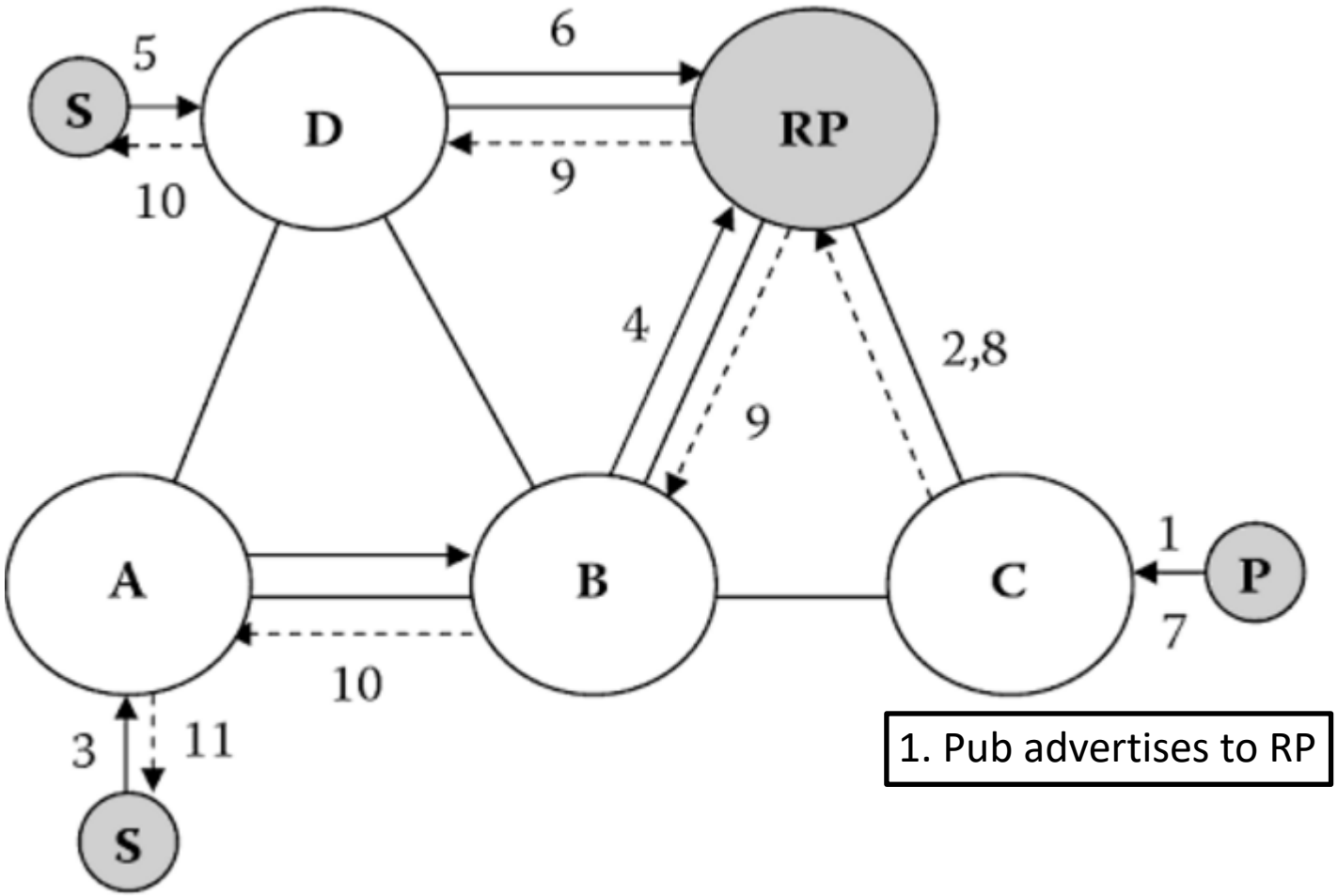
RDV-based Overlay: Hermes (2004)

- Supports **content-based** matching with advertisements
- One broker is the **rendezvous point**
- Each broker knows how to reach the rendezvous point
- Rendezvous point performs **matching** and **subscriber notification**

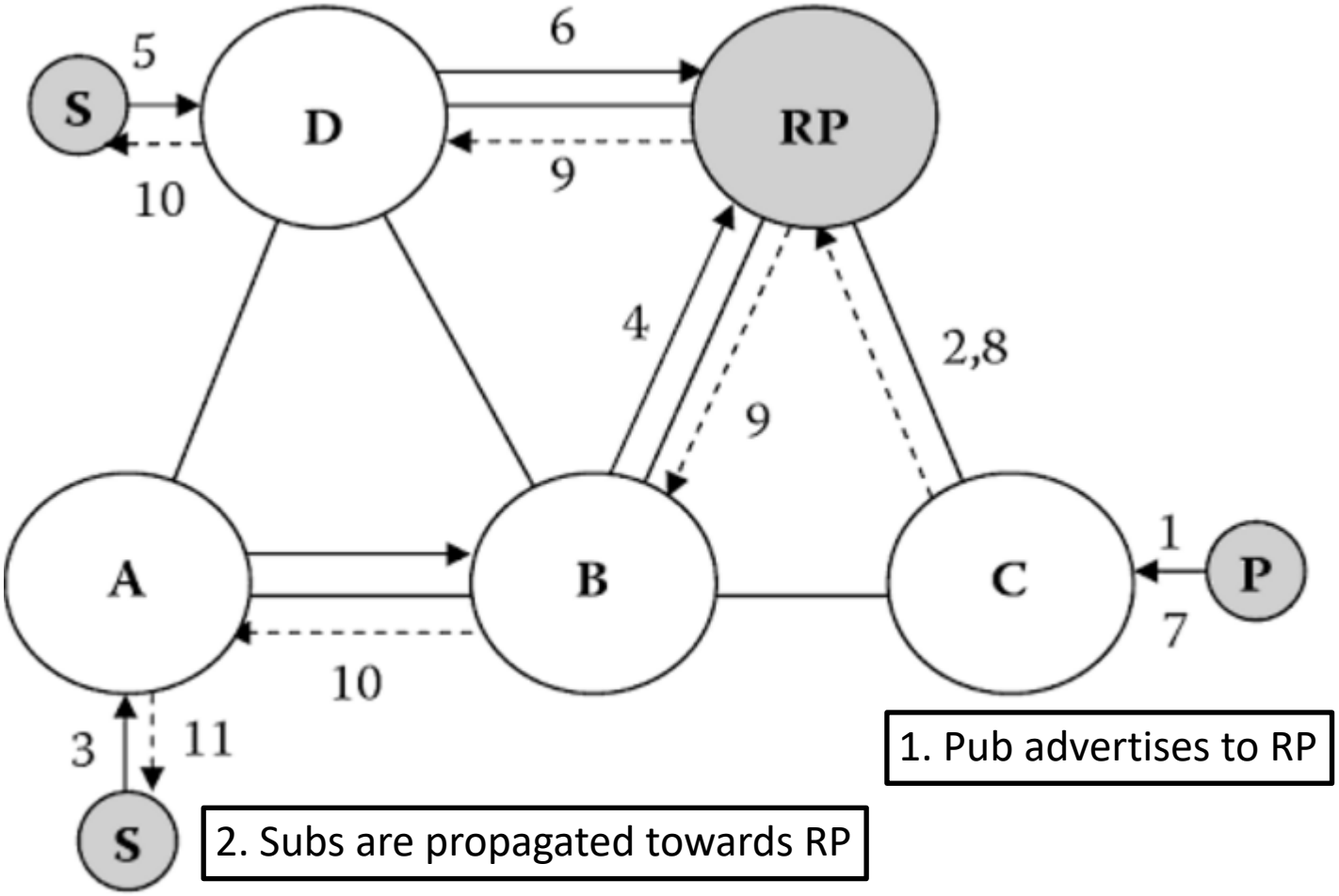
Hermes Routing



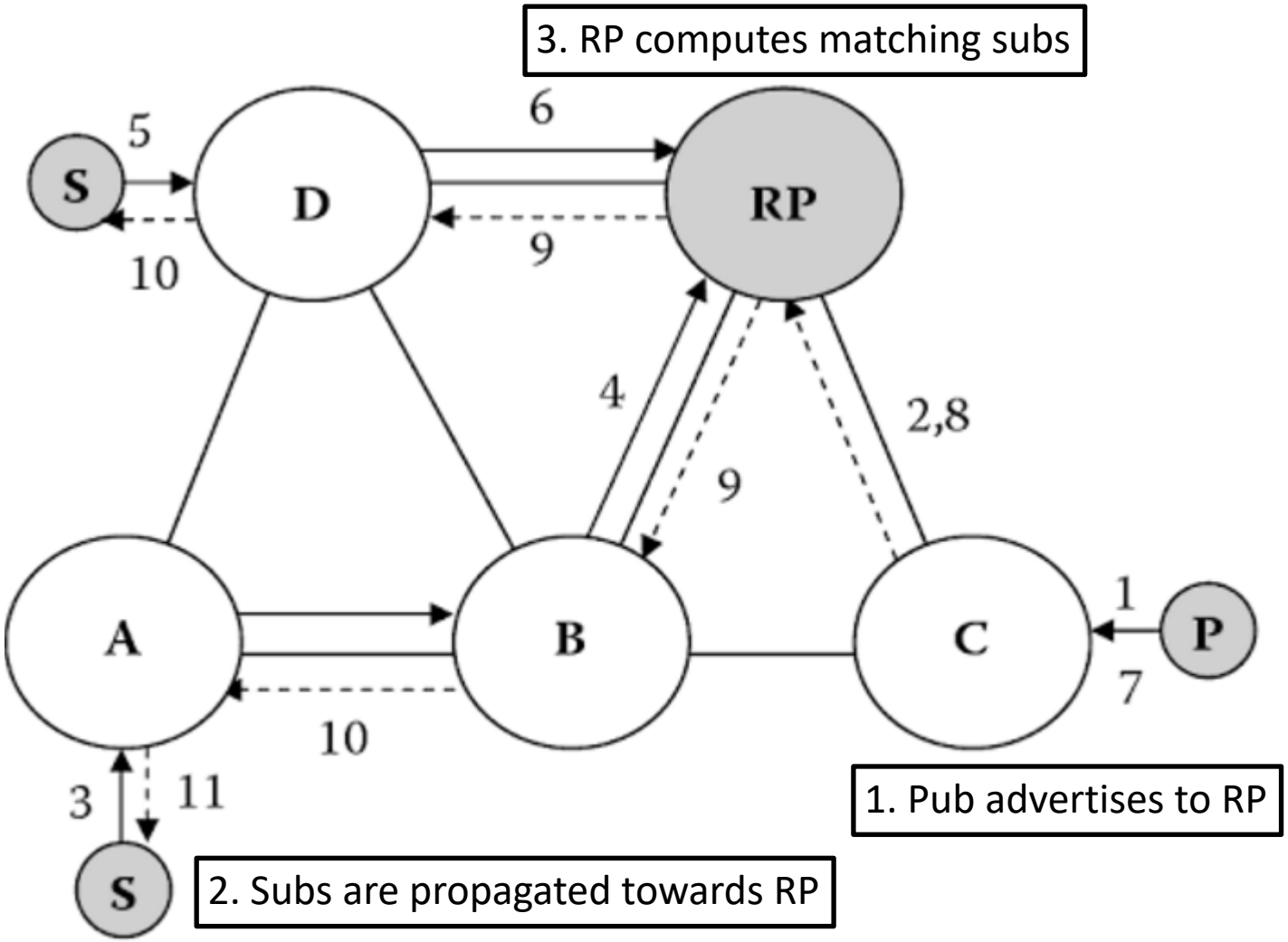
Hermes Routing



Hermes Routing



Hermes Routing



Possible Optimizations

- More than one rendezvous point
  - Use space partitioning
  - Increases state complexity of forwarding brokers
- **Bloom filters** using Link IDs
  - Reduces state complexity of forwarding brokers
  - Introduces false positive dissemination
  - Increases processing time at the RP

Bloom Filter

- Efficient data structure for representing whether elements are in a set
- Represented as a **bit vector** with **fixed size**
- Elements can be inserted into the Bloom filter (the set)
- Element can be tested to identify whether it is in the set
- Cannot obtain a list of all elements in the Bloom filter
- **False positives** (elements that would show as members but are not) are possible – i.e., can incorrectly identify an element as part of the set
- **False negatives** (elements that are in the set but don't show as such) **are not possible**

Based on k *hash functions.*

Bloom Filter Operations

Start with an m bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in set S, k times. If $H_i(x_j) = a$, set $B[a] = 1$

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

To check if y is in S, check B at $H_i(y)$. All k $H_i(y)$ must be 1

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Possible to have a *false positive*; all k values are 1, but y is not in S!

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Example of Bloom Filter

Start with a *16* bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash1("id1") = 7, Hash2("id1") = 2, …, Hash8("id1") = 5

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Hash1("id2") = 7, Hash2("id2") = 11, Hash3("id2") = 4

B

0	1	0	1	1	0	1	0	0	1	1	1	0	1	1	0

Hash1("id3") = 5, Hash2("id3") = 5,…, Hash8("id3") = 7

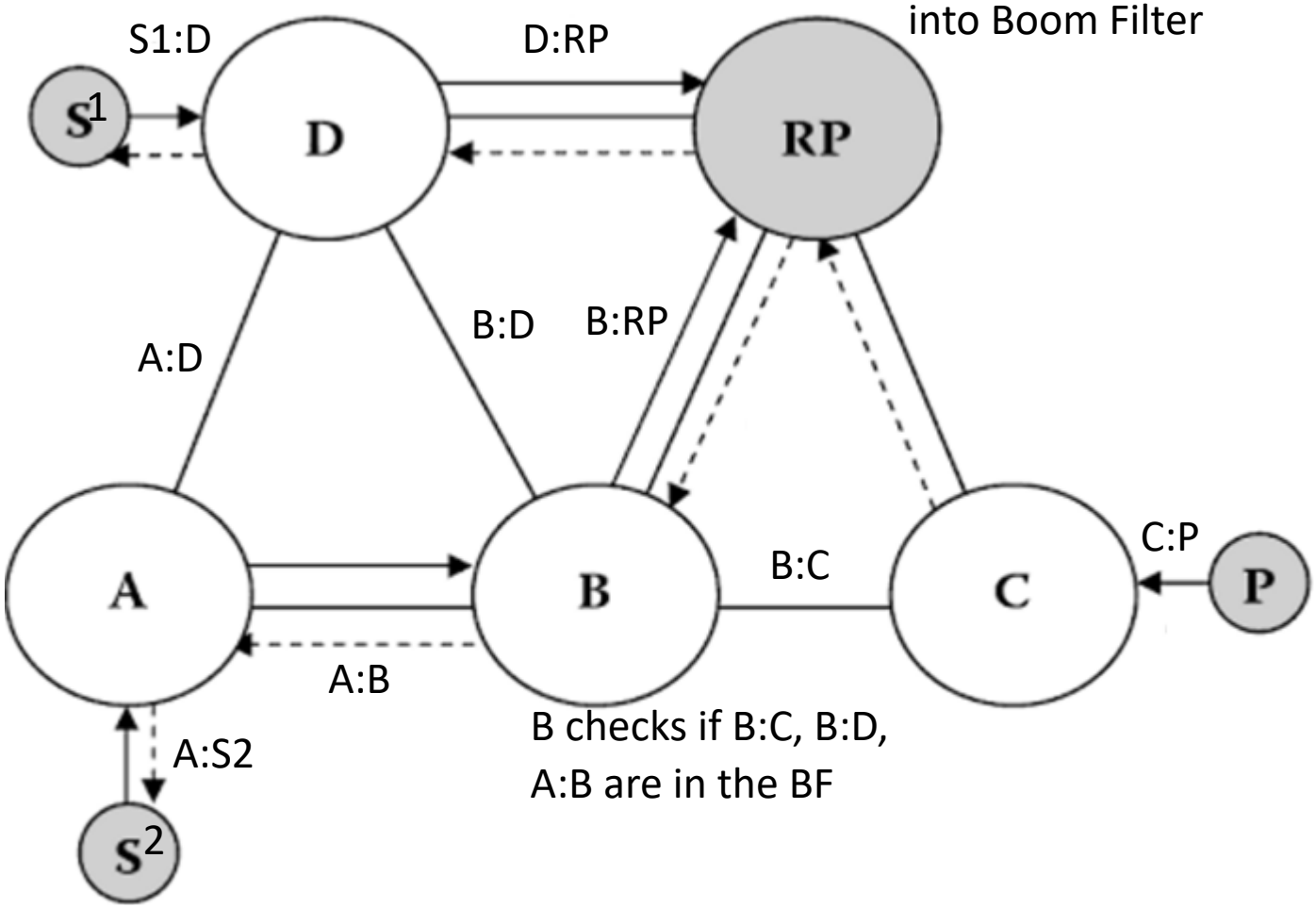
B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

8 hash functions

Link IDs encoding in Bloom Filters (LIPSIN 2009)

Insert
{D:RP, S1:D, B:RP, A:B, A:S2}
into Boom Filter



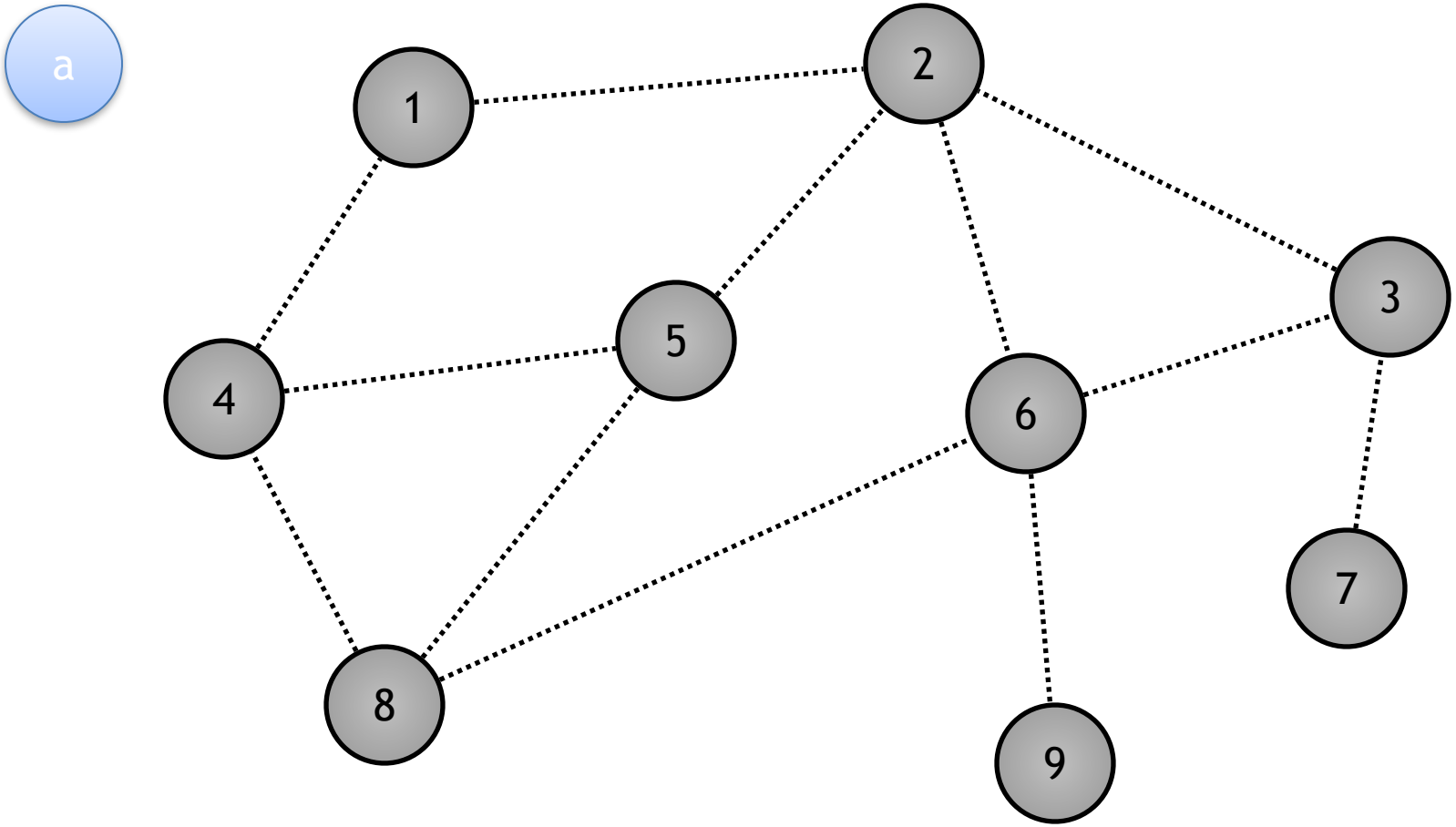
Filtering-Based Routing: PADRES (2003)

- Each broker performs filtering
- Based on the filtering, each broker disseminates publications towards matching subs
- The next hops repeat the process until subs are reached
- Supports any matching model (e.g. content-based)
- Subs are flooded to initialize routing paths
- **Optimization**:
  - Subscription covering
  - Advertising-based routing

http://padres.msrg.org

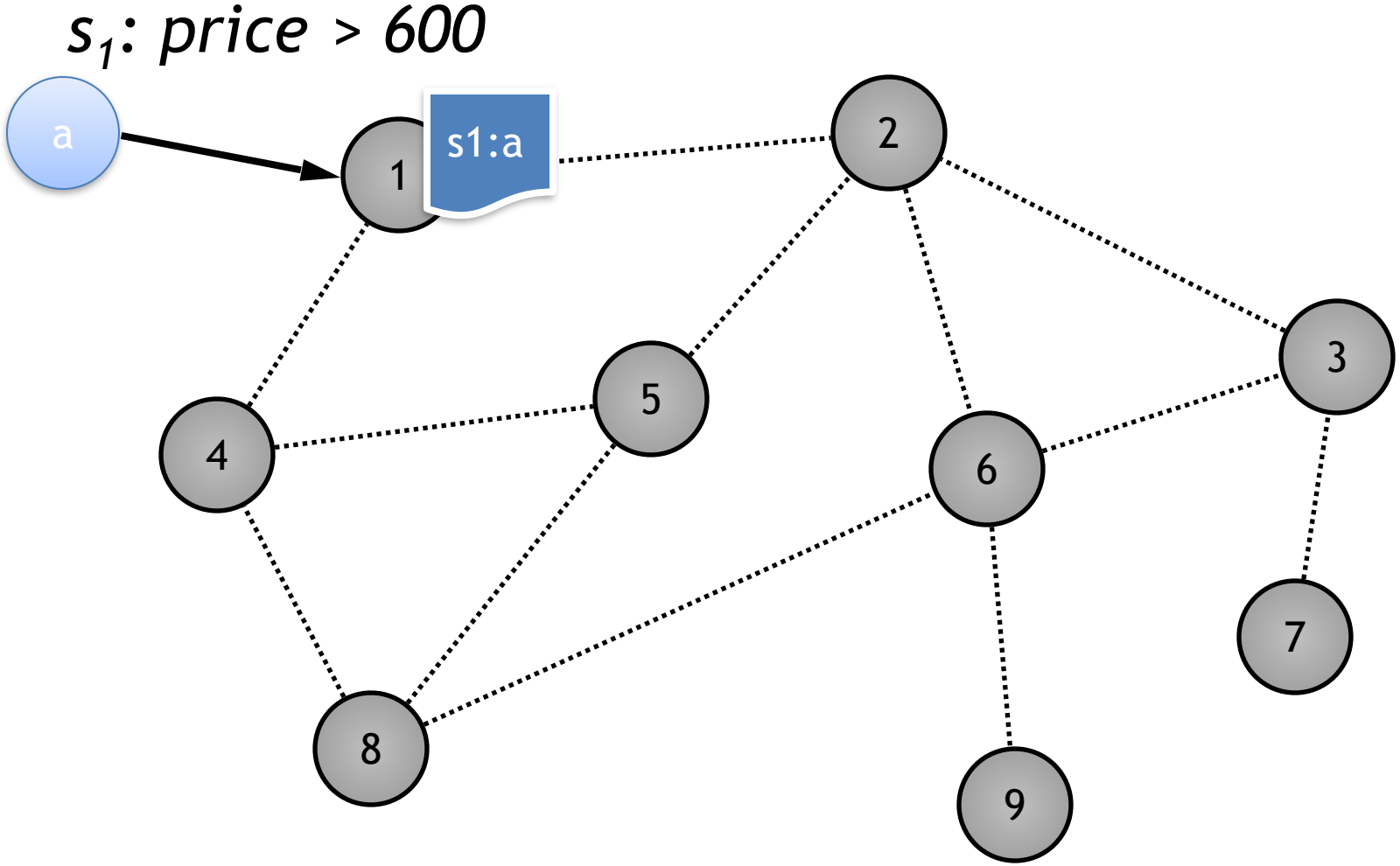
Filtering-Based Routing

Subscription Flooding



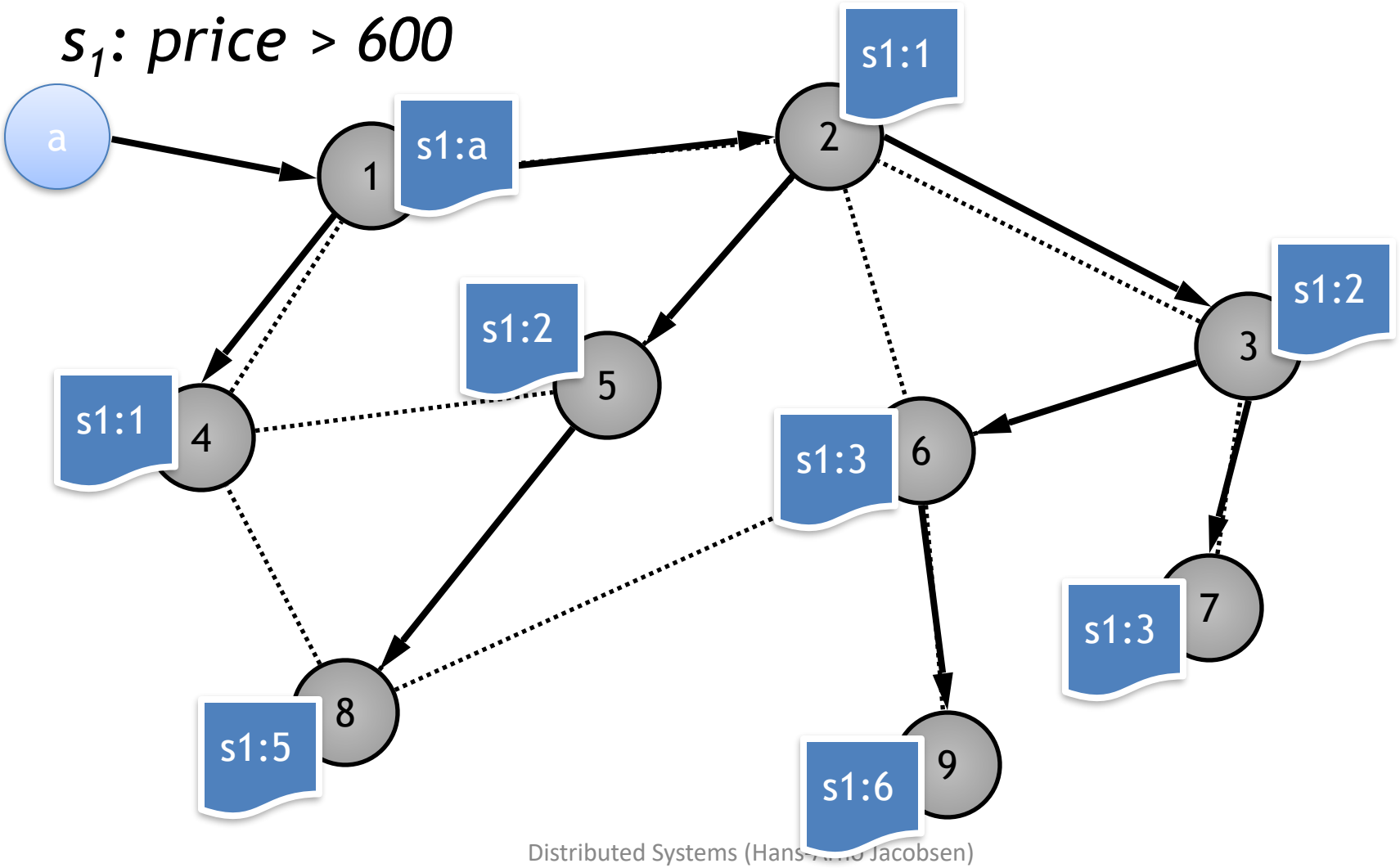
Filtering-Based Routing

Subscription Flooding



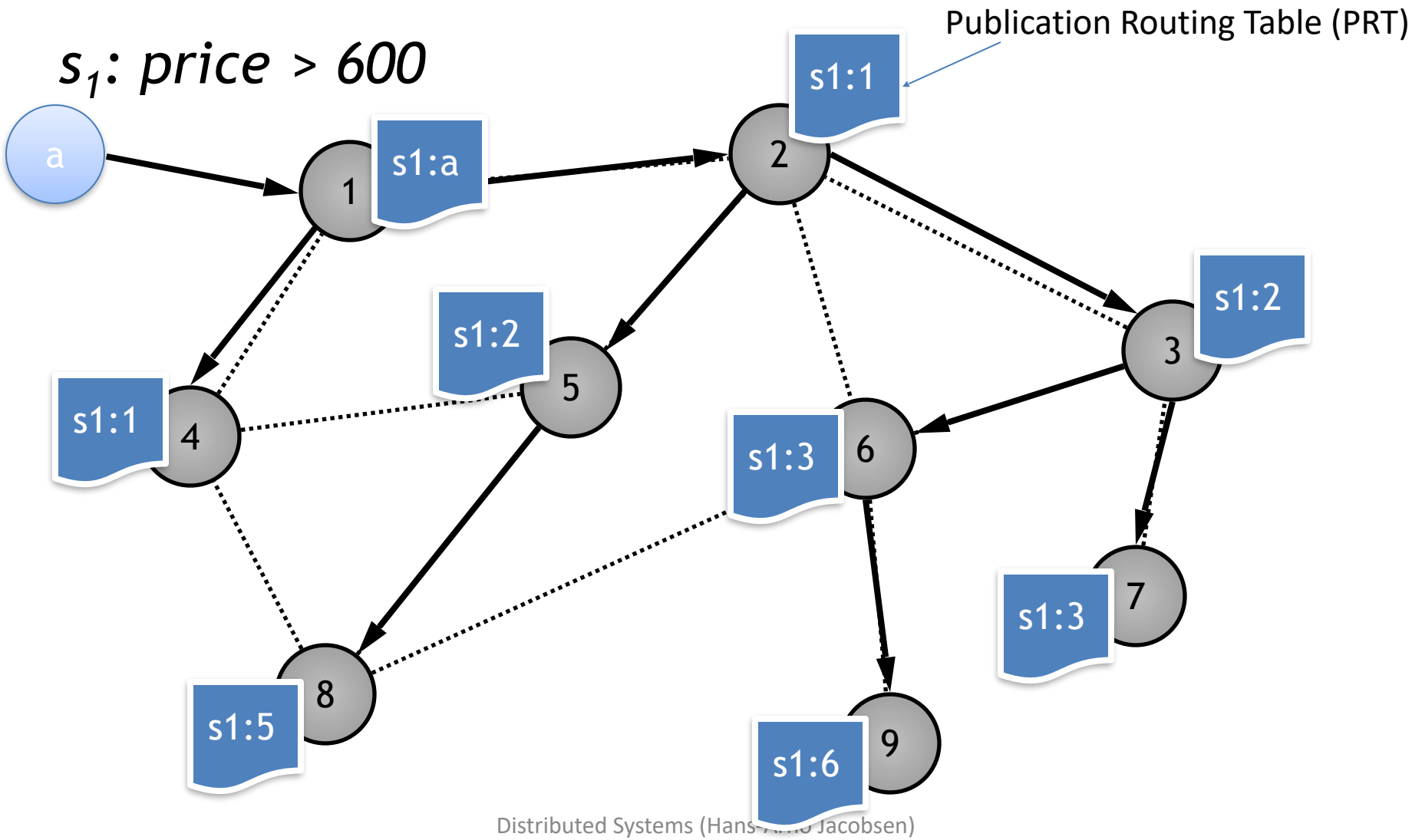
Filtering-Based Routing

Subscription Flooding



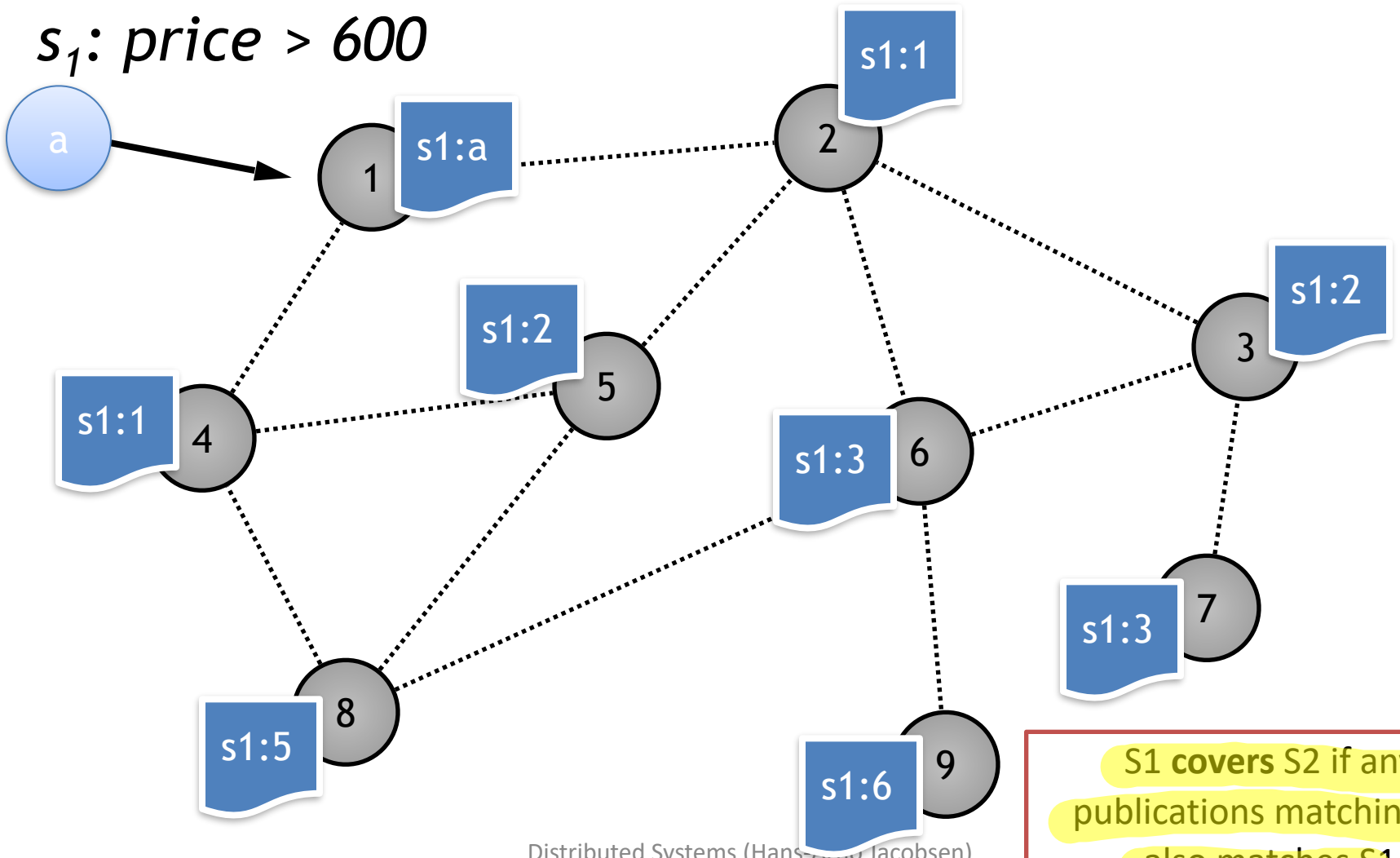
Filtering-Based Routing

Subscription Flooding



Filtering-Based Routing

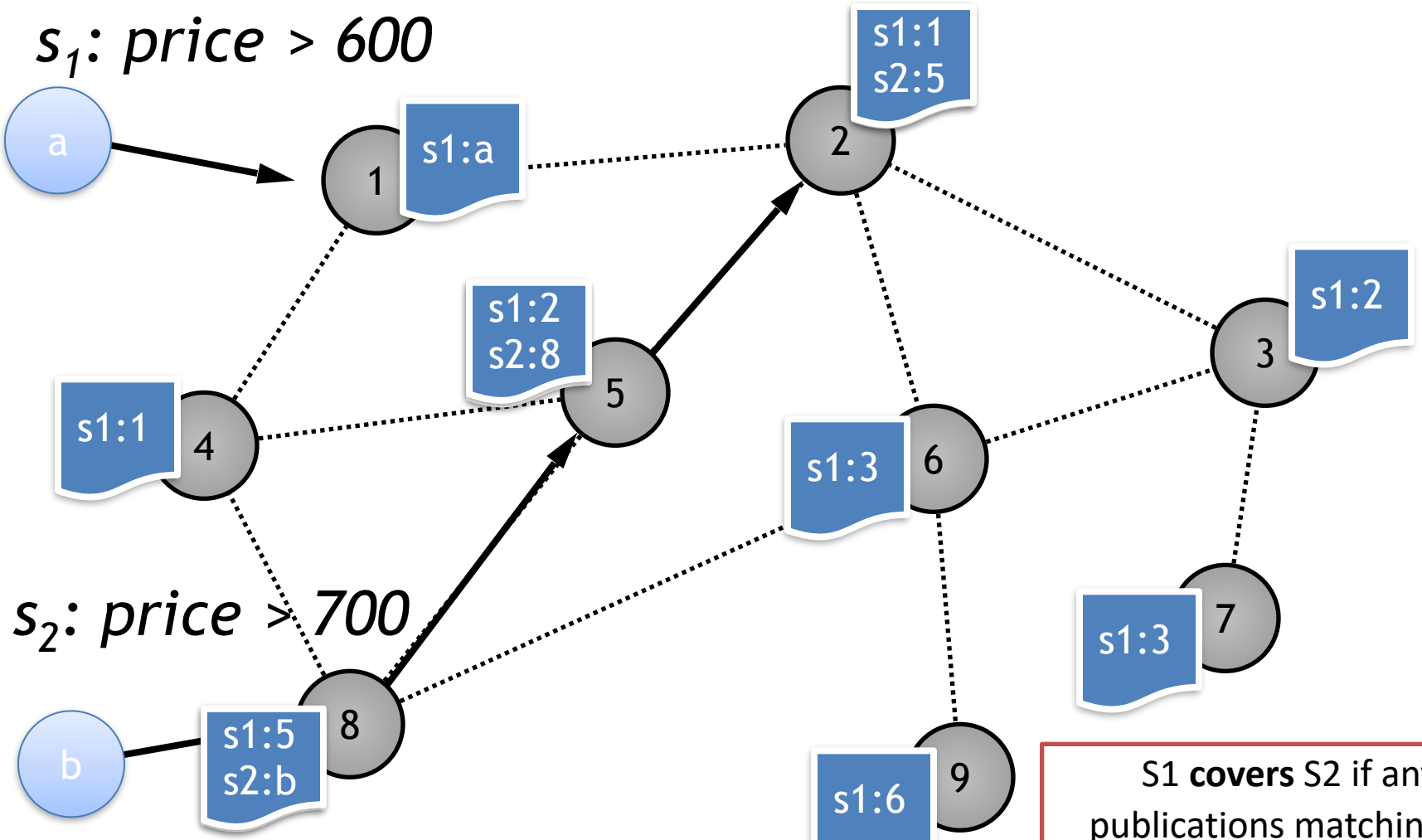
Subscription Covering



S1 **covers** S2 if any publications matching S2 also matches S1

Filtering-Based Routing

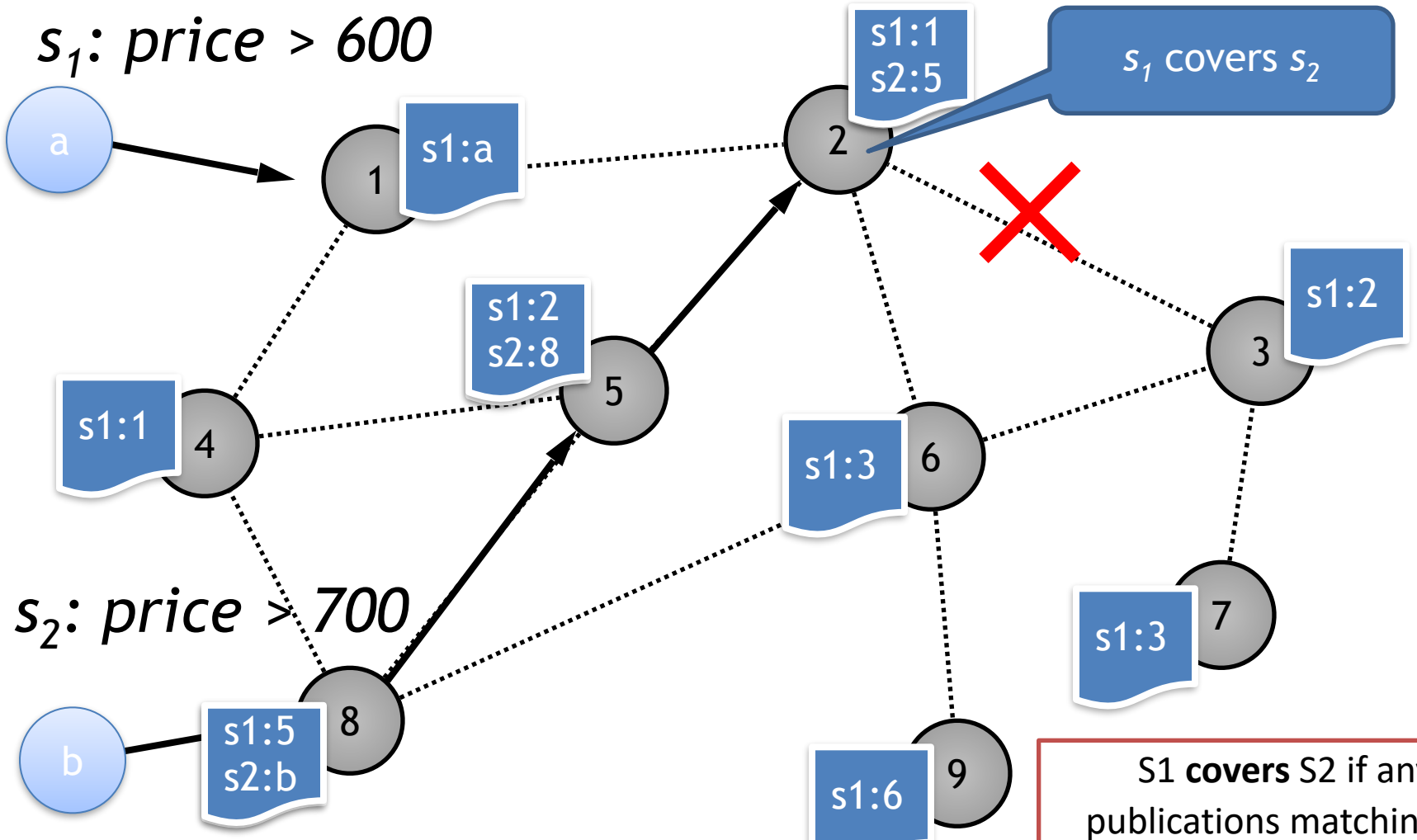
Subscription Covering



S1 **covers** S2 if any publications matching S2 also matches S1

Filtering-Based Routing

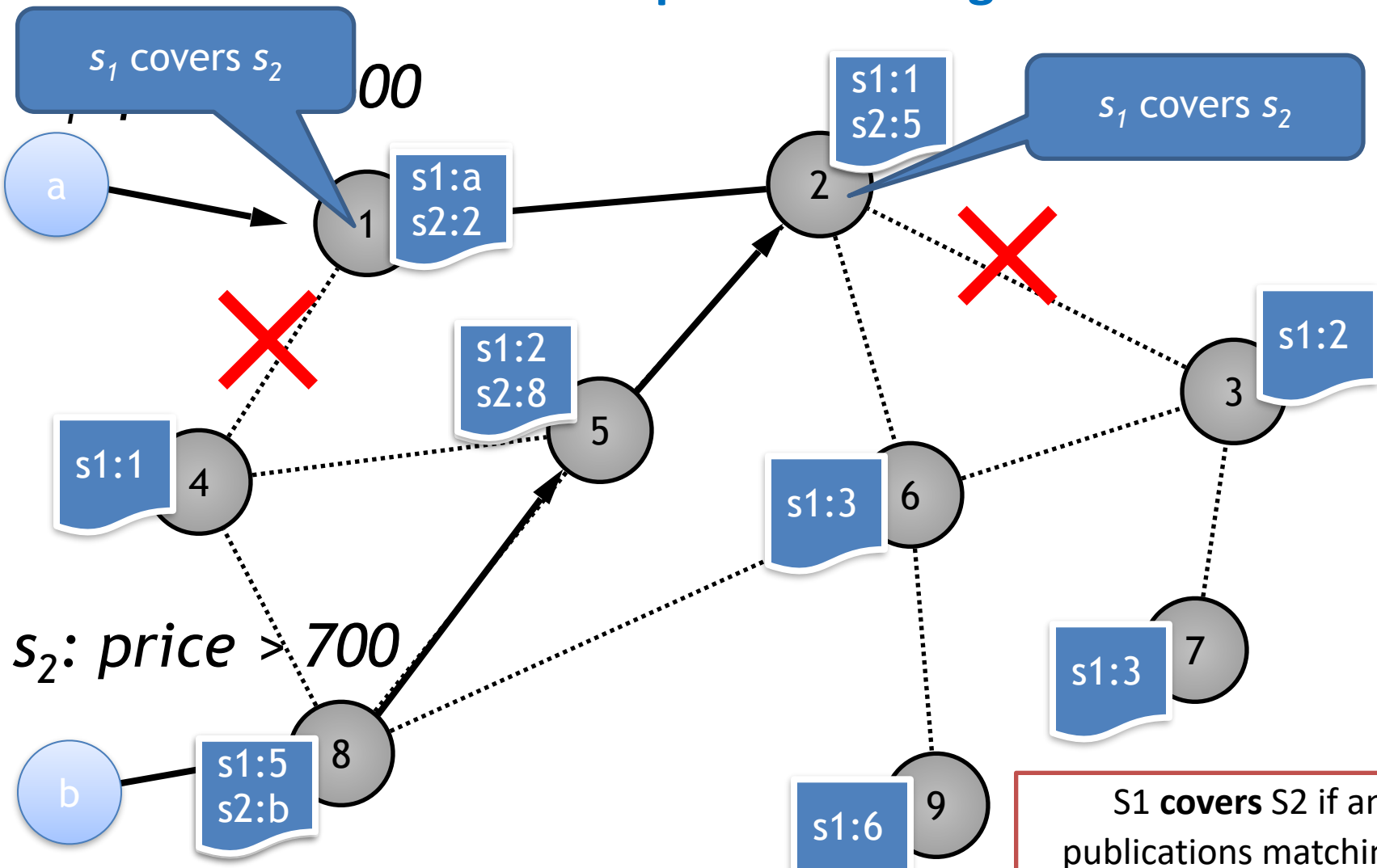
Subscription Covering



S1 **covers** S2 if any publications matching S2 also matches S1

Filtering-Based Routing

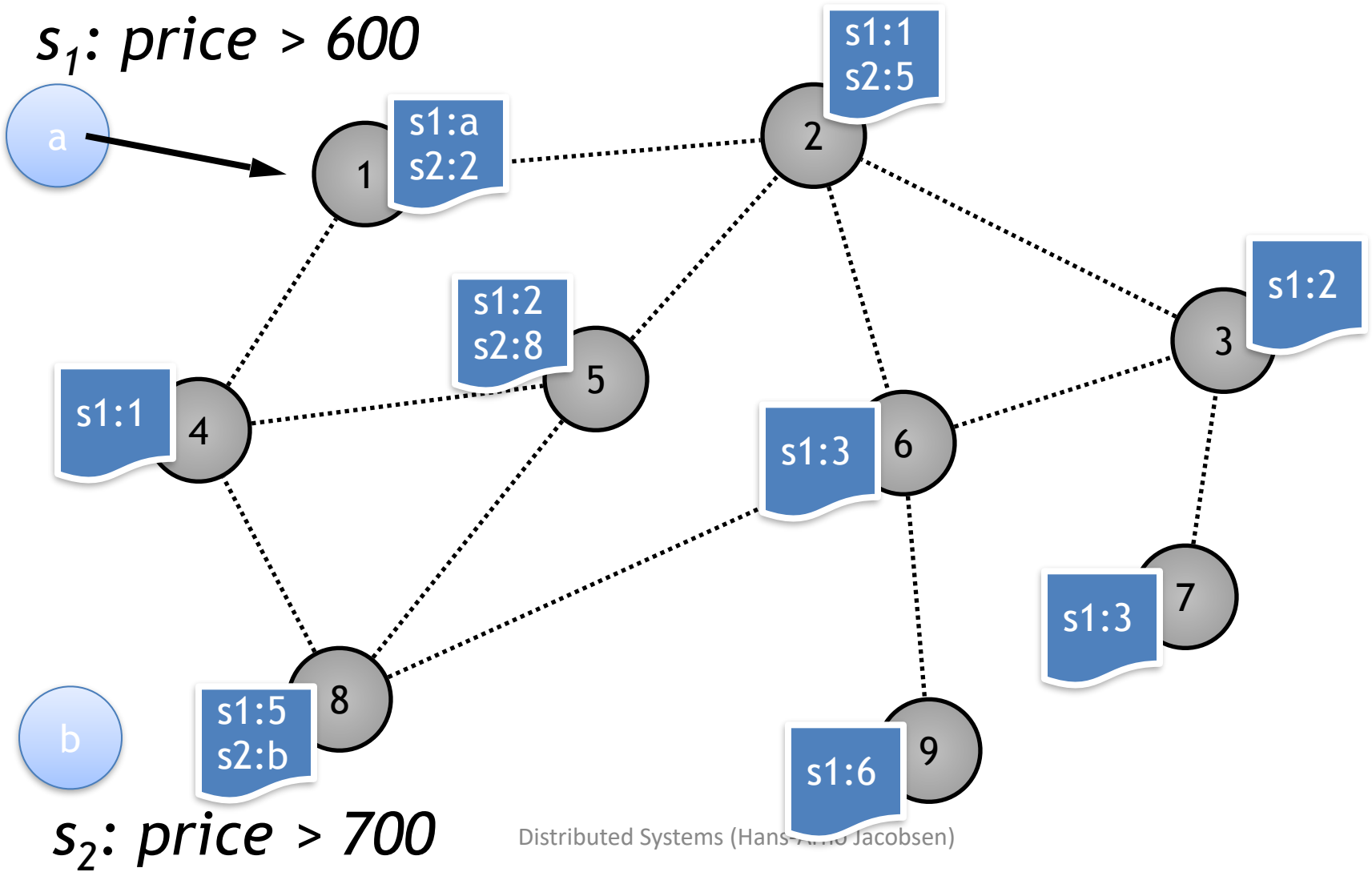
Subscription Covering



S1 **covers** S2 if any publications matching S2 also matches S1

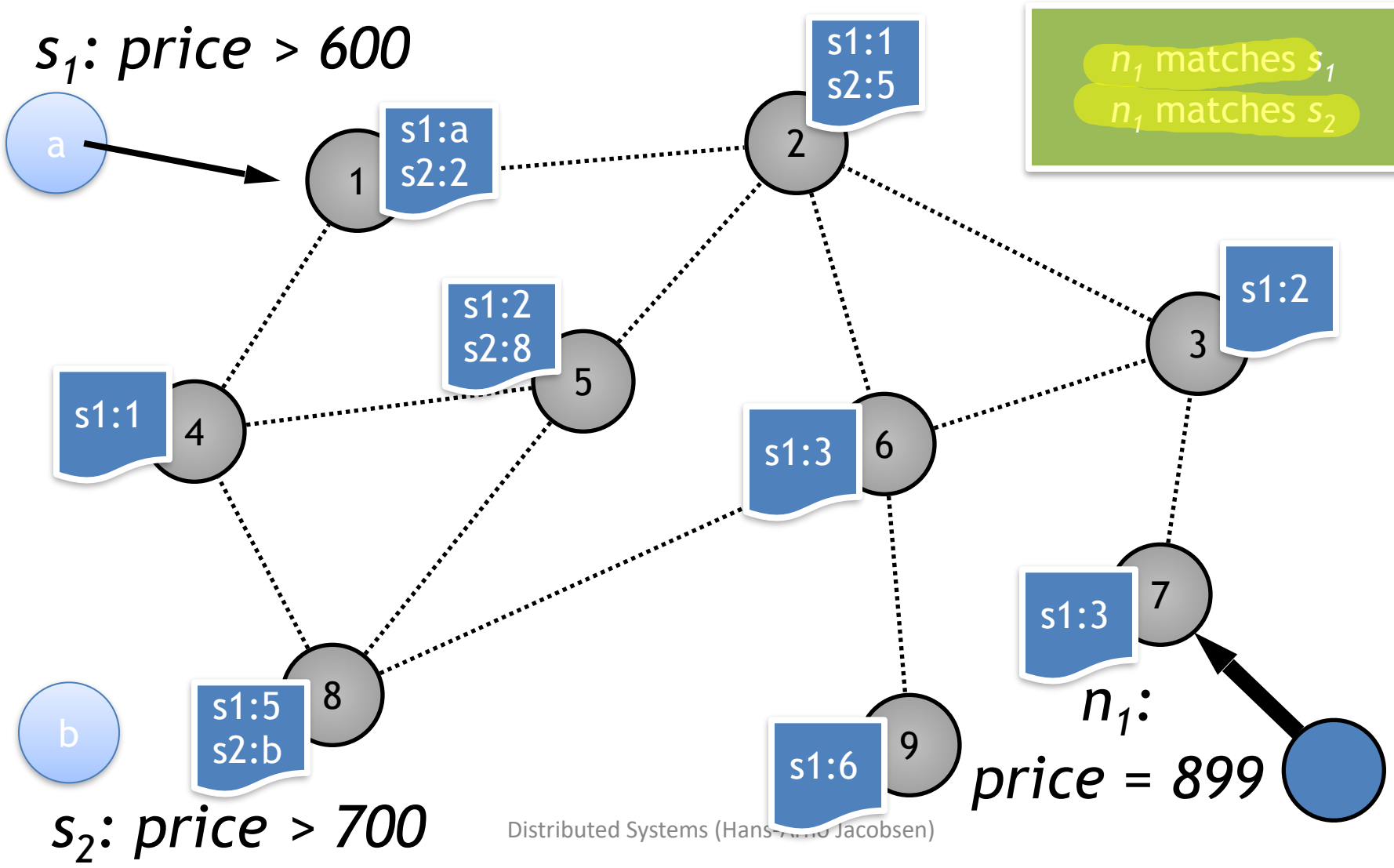
Filtering-Based Routing

Notification Delivery



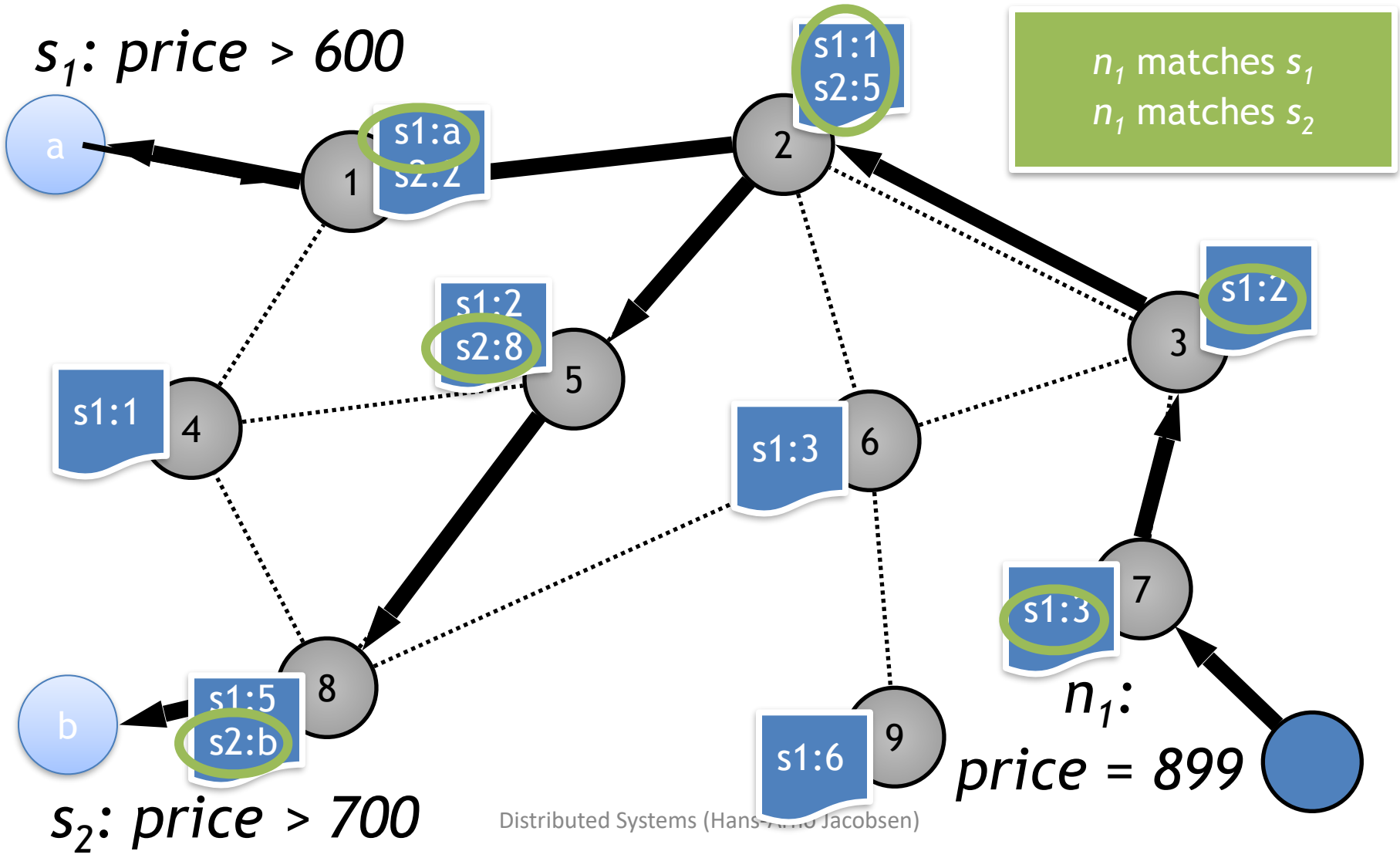
Filtering-Based Routing

Notification Delivery



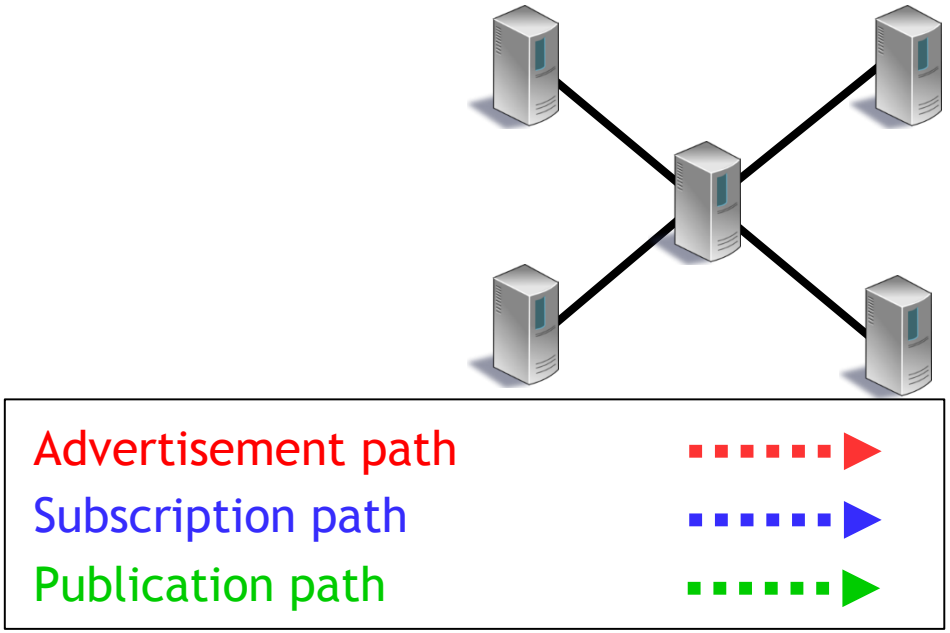
Filtering-Based Routing

Notification Delivery



Advertisement-Based Routing

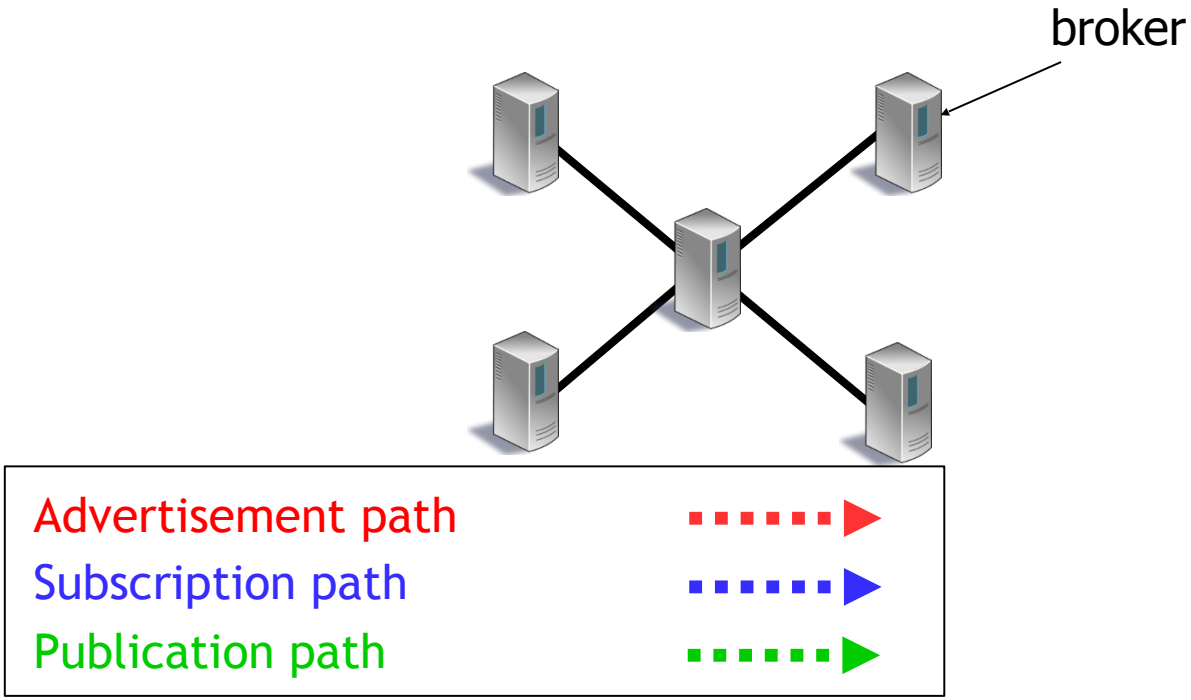
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

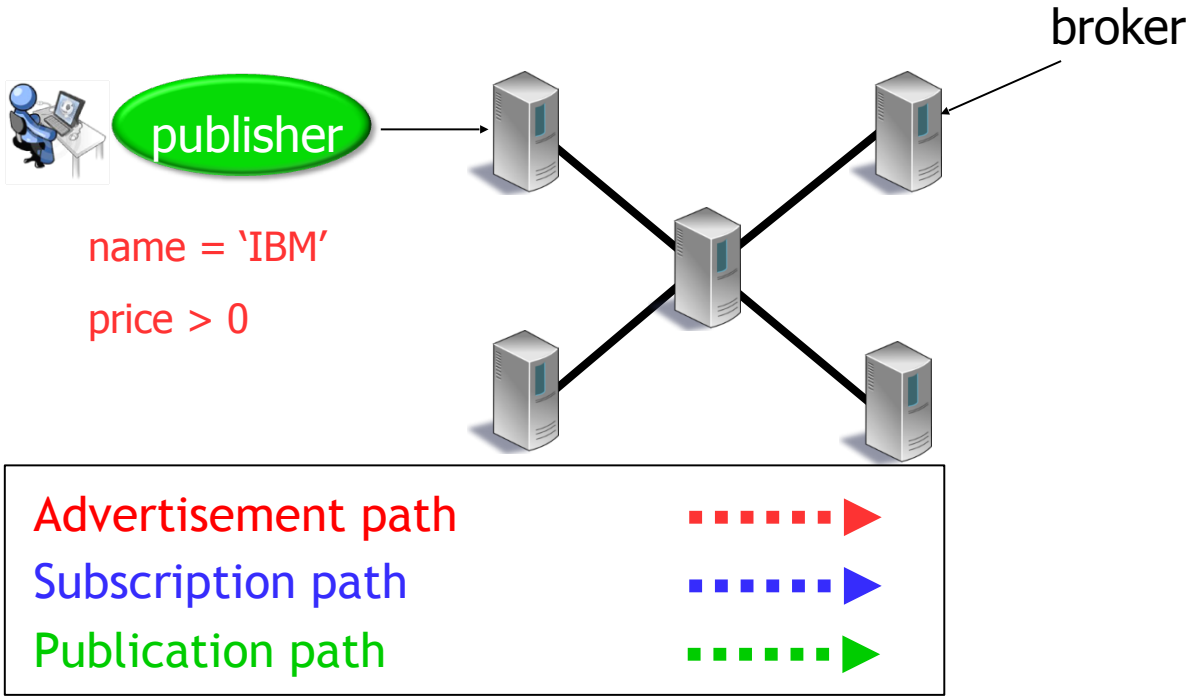
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

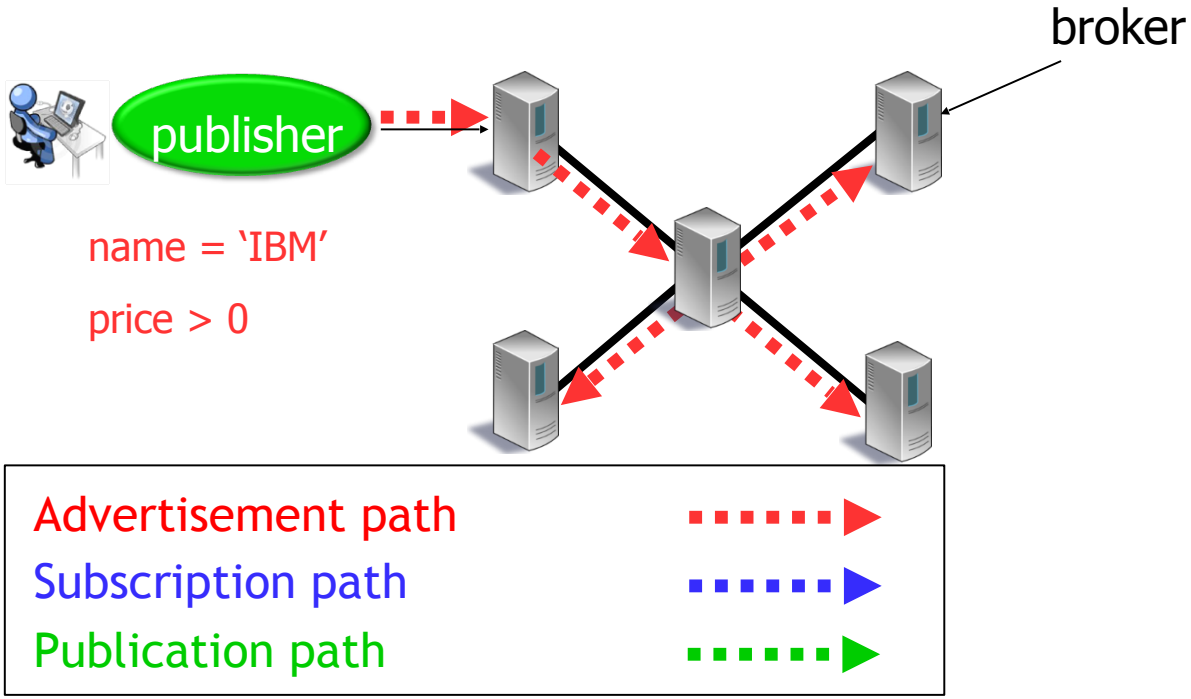
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

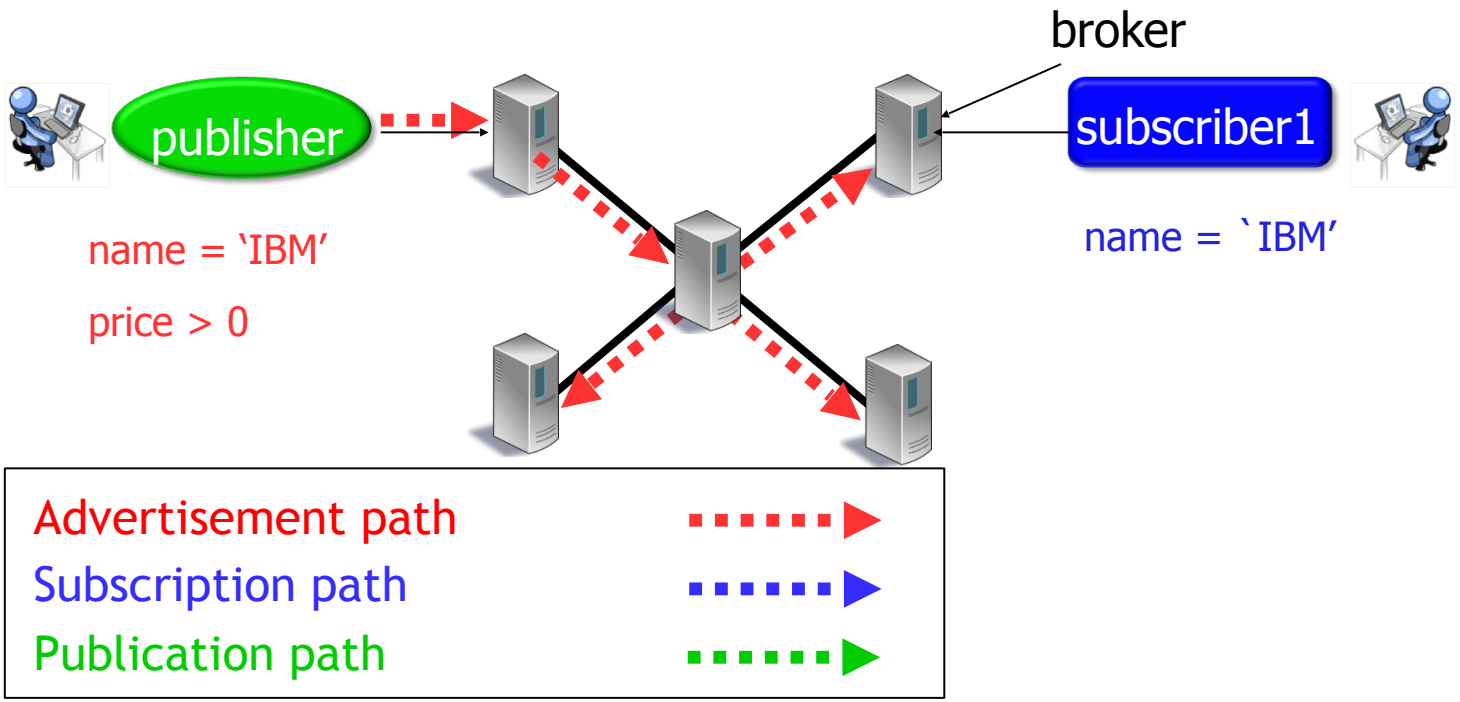
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

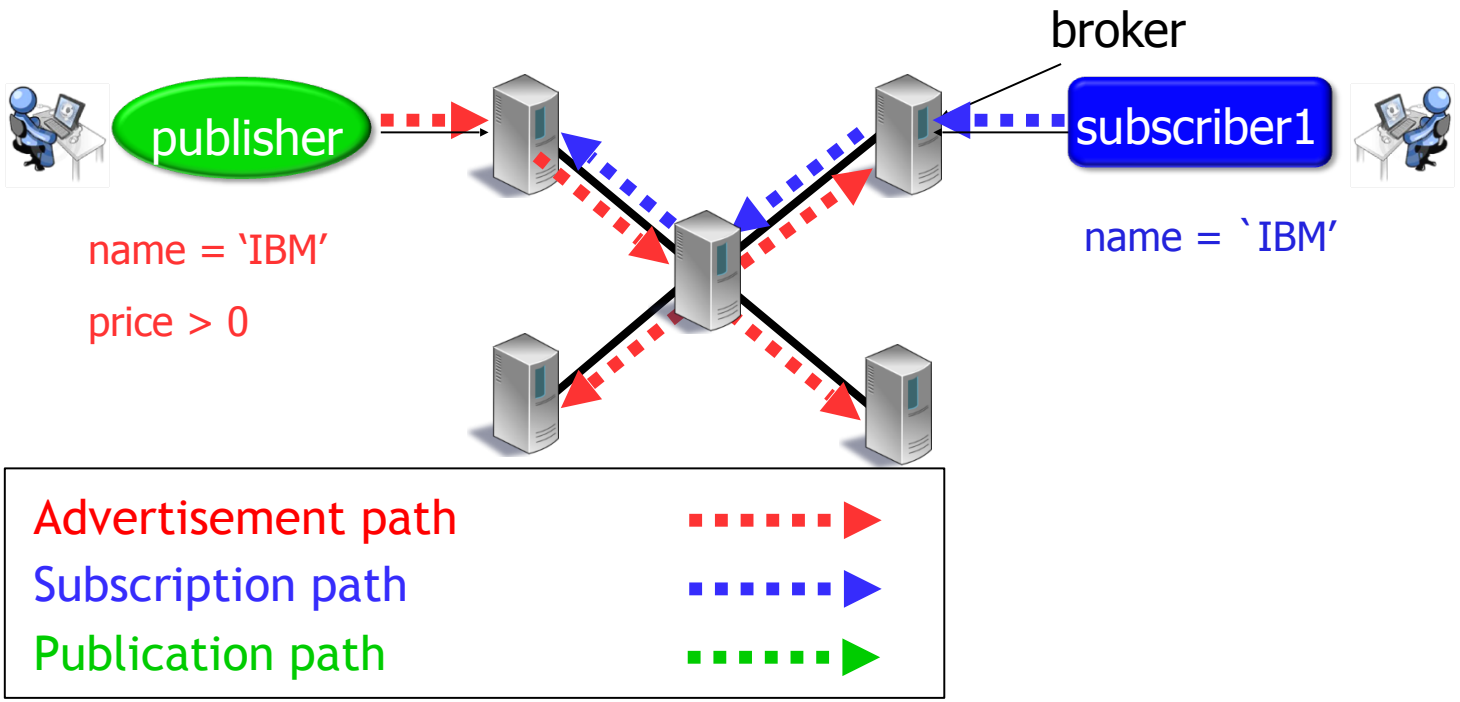
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

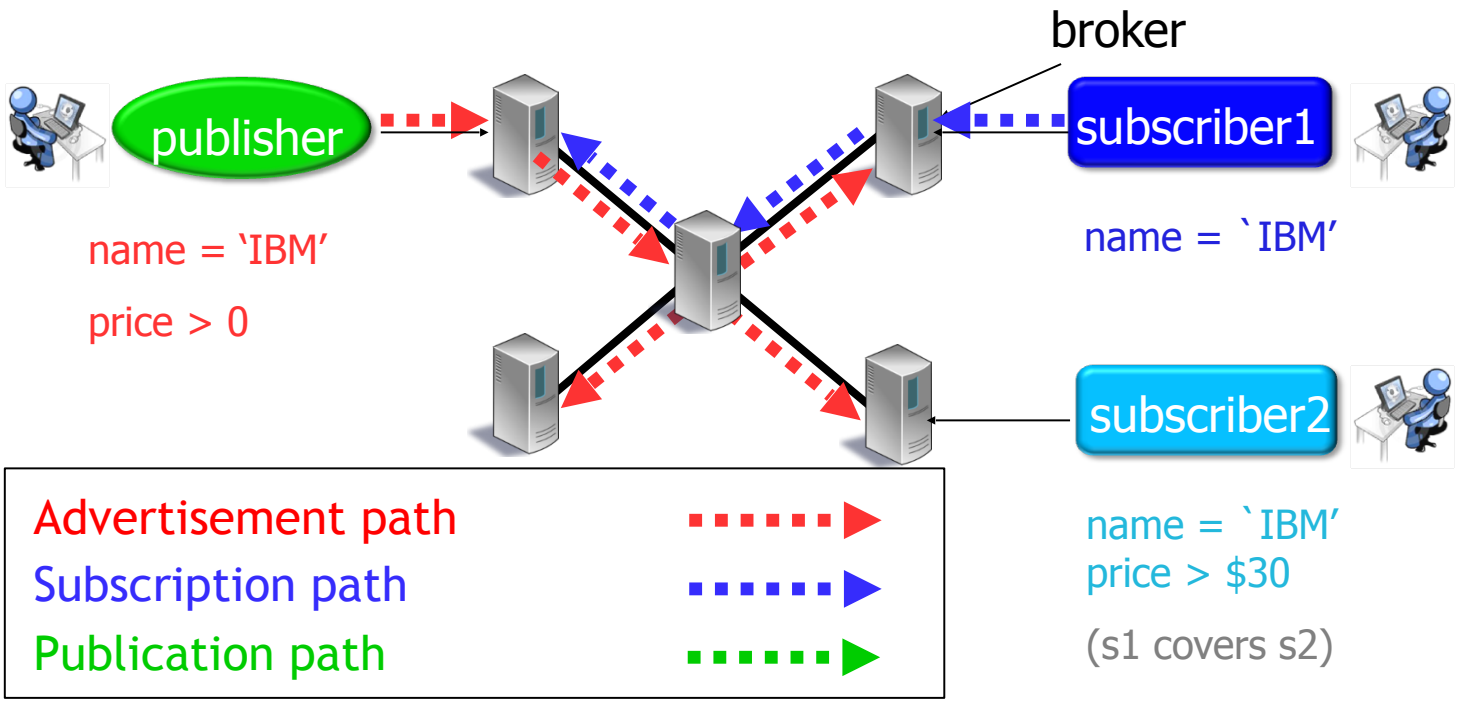
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

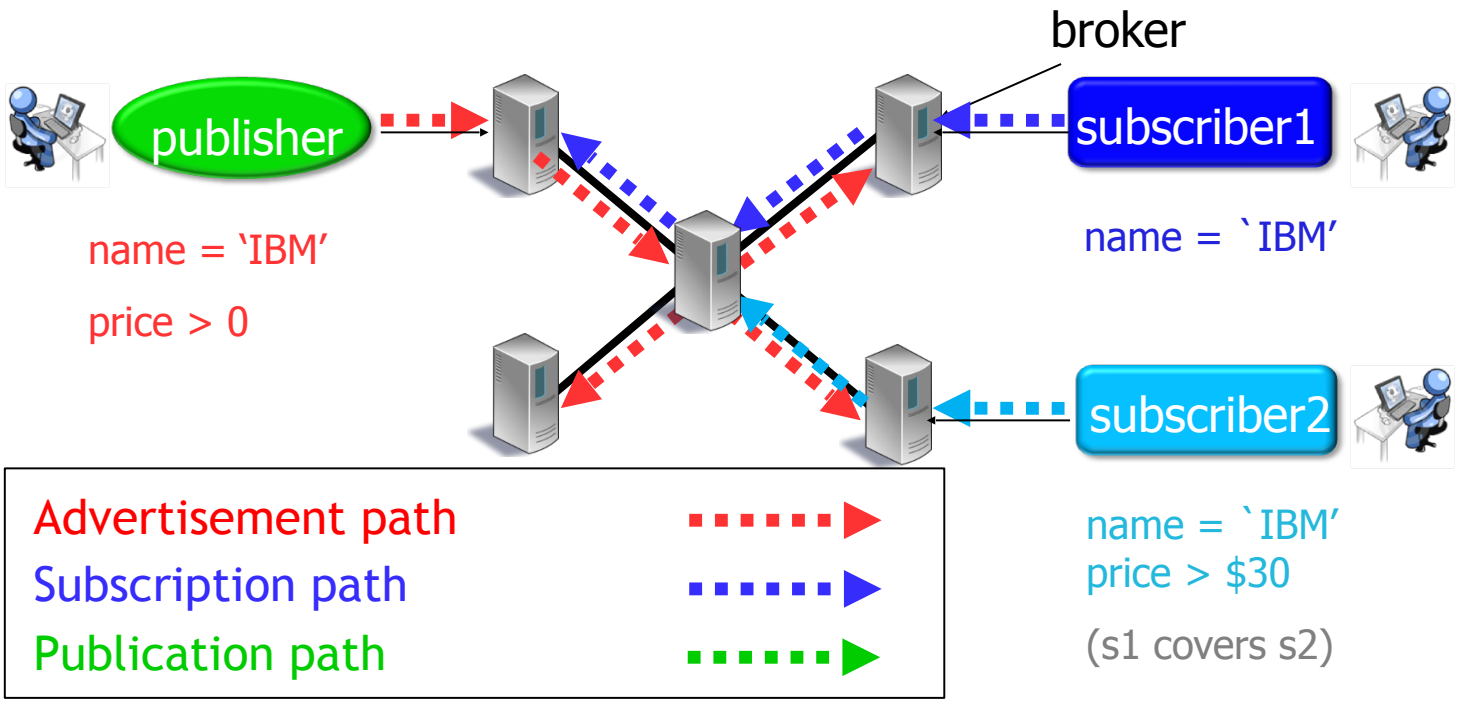
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

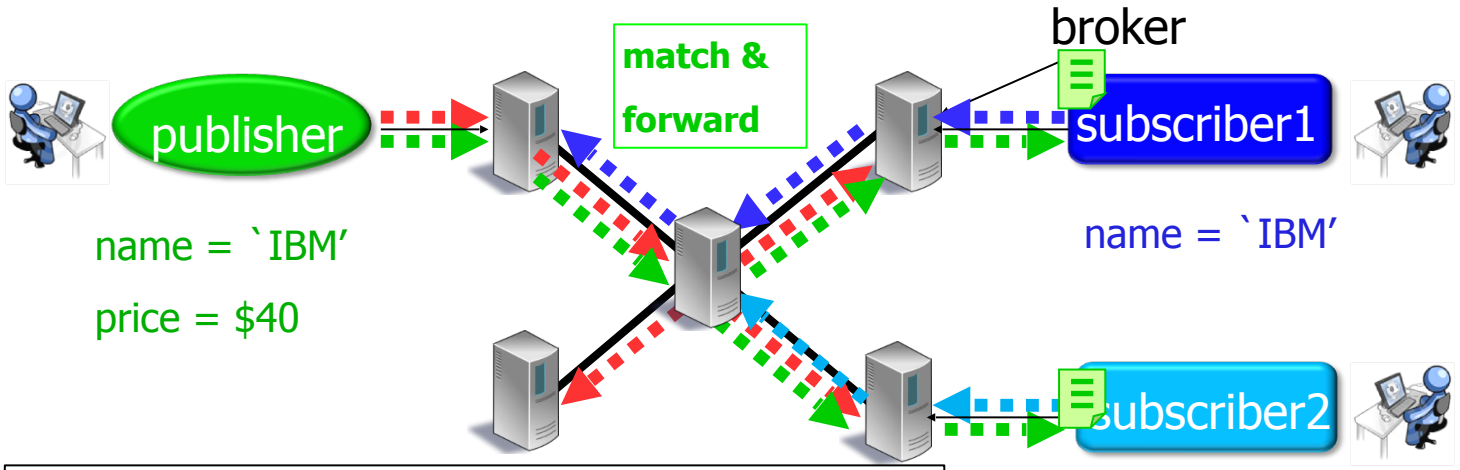
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Routing Algorithms Summary

- Rendezvous-based
  - Partition-based: Simple, requires master
  - DHT-based: log(n) hops, structured network
- Overlay-based
  - Rendezvous-based: Efficient matching, inefficient communication, RPs are bottlenecks
  - Filtering-based: Computation intensive, efficient communication
  - Advertisement flooding: Useful if the number of publishers is much smaller than subscribers