Distributed Systems

Indirect Communication

Why?

Point-to-point communication

- Participants need to exist at the same time
  - Establish communication
- Participants need to know address of each other and identities
- Not a good way to communicate with several participants

Indirect communication

- Communication through an intermediary
  - no direct coupling between the sender and the receiver(s)
- Space uncoupling – no need to know identity of receiver(s) and vice versa
  - participants can be replaced, updated, replicated, or migrated
- Time uncoupling – independent lifetimes
  - requires persistence in the communication channel

Good for …

- scenarios where users connect and disconnect *very* often
  - Mobile environments, messaging services, forums
- event dissemination where receivers may be unknown and change often
  - RSS, events feeds in financial services
- scenarios with very large number of participants
  - Google Ads system, Spotify
- Commonly used in cases when change is anticipated
  - need to provide dependable services

… but there are also some disadvantages

- performance overhead introduced by adding a level of indirection
  - reliable message delivery, ordering → (-) effect on scalability
- more difficult to manage because lack of direct coupling
- difficult to achieve end-to-end properties
  - real time behavior
  - security

Examples of Indirect communication

Commonalities

- Some processes write information into an abstraction and some other reads from that abstraction

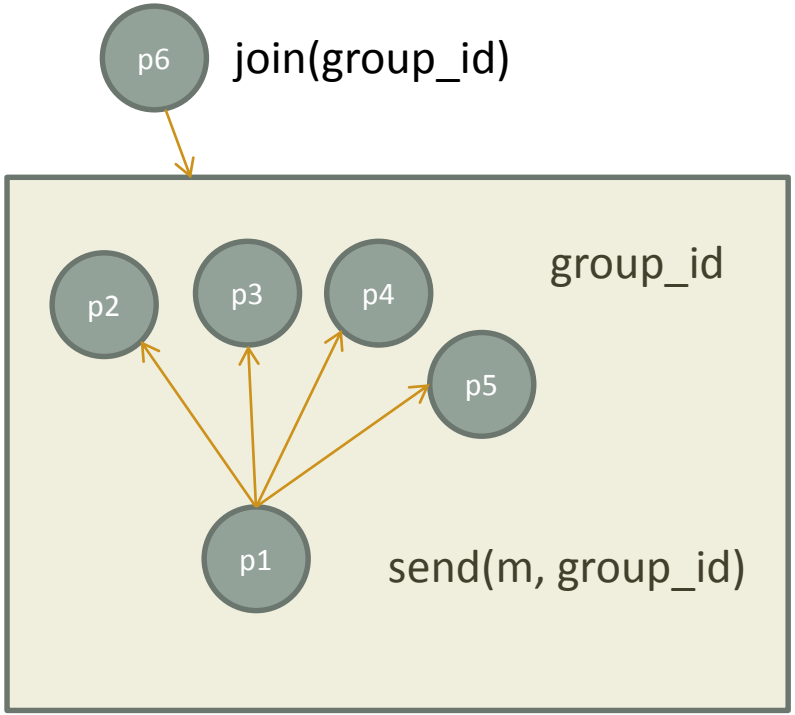
Communication-based
- a queue
- a group
- a channel

Potential to scale to very large systems
- Key is routing infrastructure

State-based
- an array of memory
- a space (whiteboard)

Need to maintain consistent view Of shared state

Group communication



One-to-many communication

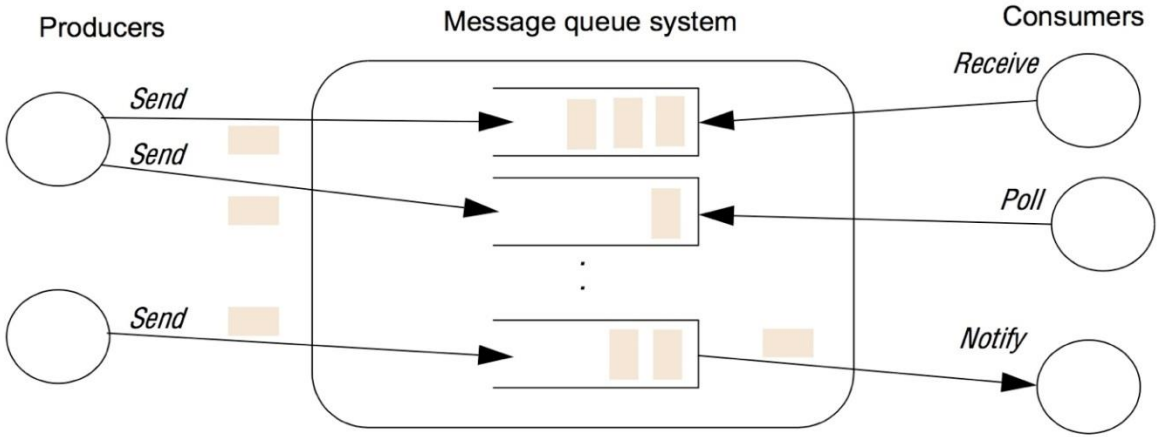
Management functionality

- Maintain membership and a mechanism to deal with failure of member(s)

Message queue systems

Message queues offer a point-to-point service in which producer processes send messages to a specified queue and consumer processes receive messages from the queue or are notified of message arrivals.

Programming model



Styles for receiving messages:

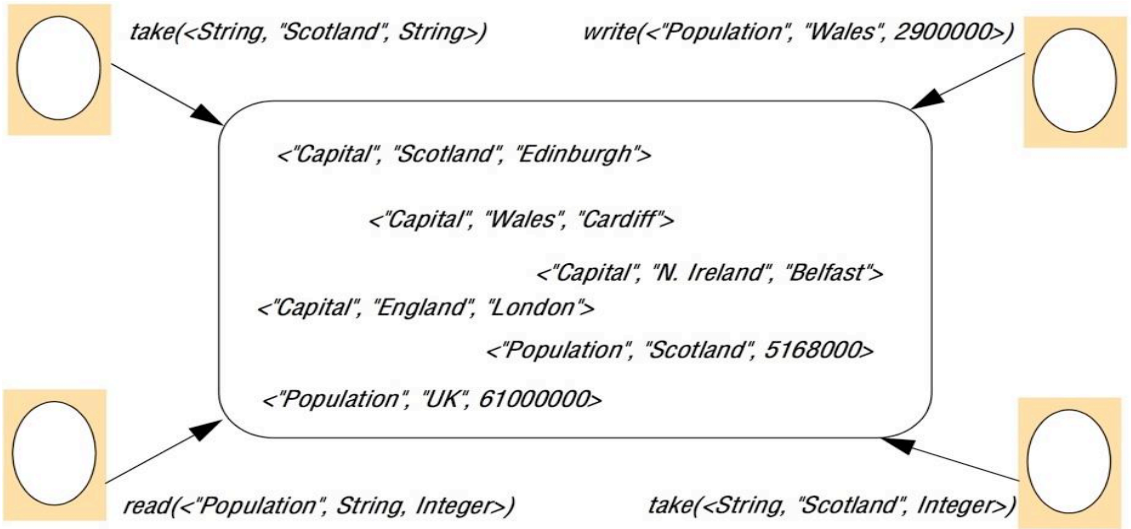
- blocking
- non-blocking *(polling)*
- notify operation

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 6.14**

Tuple space systems (generative communication)

Processes place items of structure data (*tuples*) and other processes can read or remove the tuples by specifying patterns of interest not an address. The tuple space is persistent, readers and writers do not need to exist at the same time.

Programming model



read and *take* block until there is a matching tuple in the space

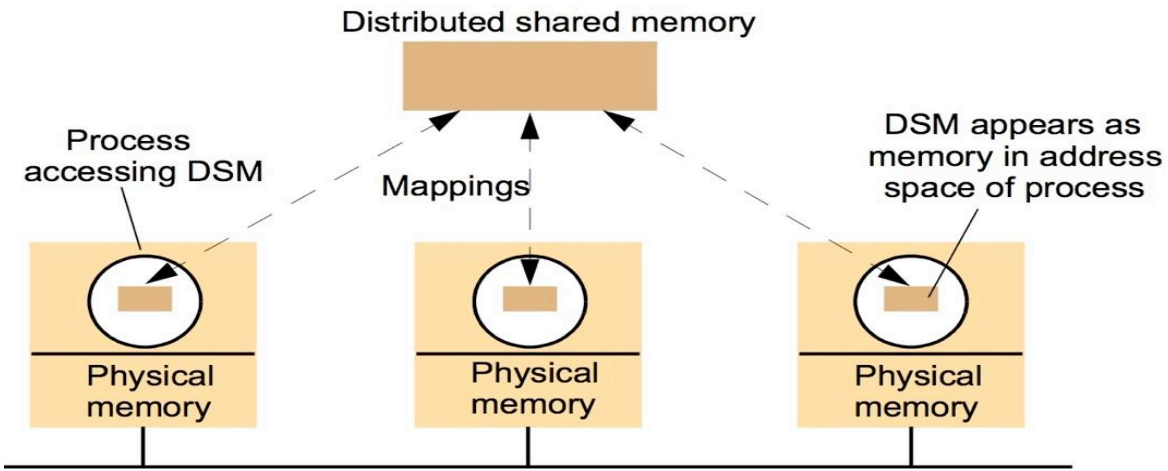
tuples are *immutable*

A tuple specification includes the number of fields and the required values or types of the fields

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 6.20**

Distributed shared memory systems

Provides an abstraction to sharing data between processes that do not share physical memory. Programmers operate as if they were in their own local address spaces. The infrastructure must ensure *timeliness*, *synchronization*, and *consistency* of data.



Individual shared data items can be accessed directly

Underlying runtime system ensures that processes executing at different computers observe the updates made by one another

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 6.19**

Publish-subscribe systems (distributed event-based systems)

A large number of publishers (*producers*) publish structured events to an event service and a large number of subscribers (*consumers*) express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.

Applications

- Financial information systems
- Live feeds of real-time data
  - RSS feeds
- Cooperative working
  - Events of shared interests
- Ubiquitous computing
  - Location services
- Monitoring applications
  - Network monitoring, internet of things

Programming model

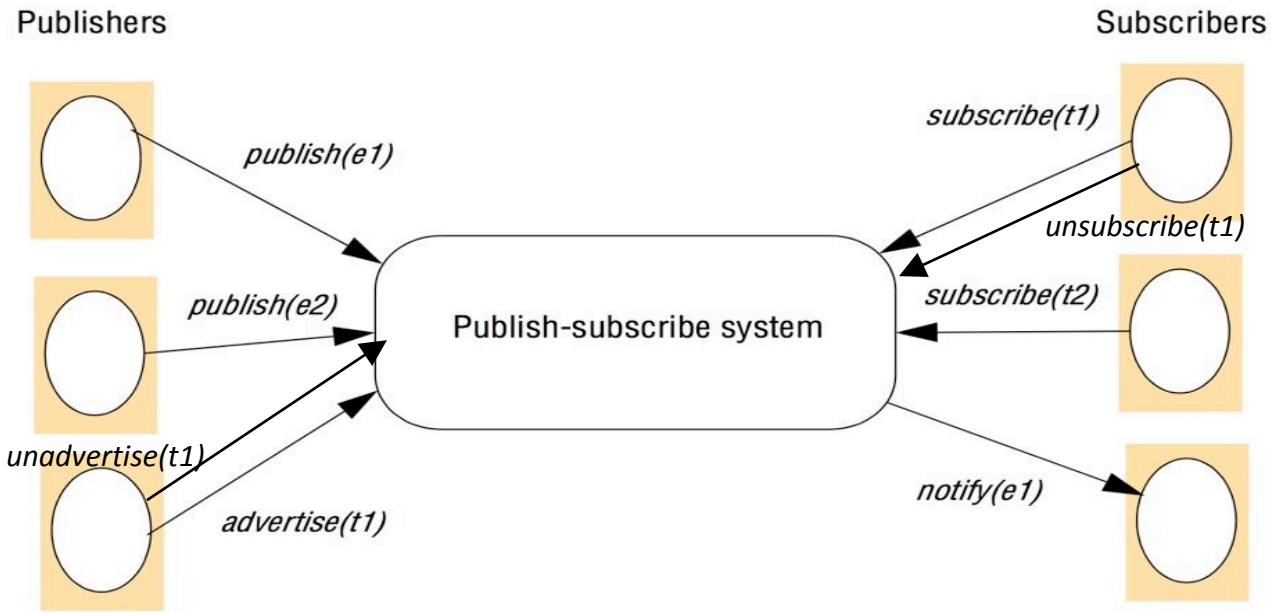


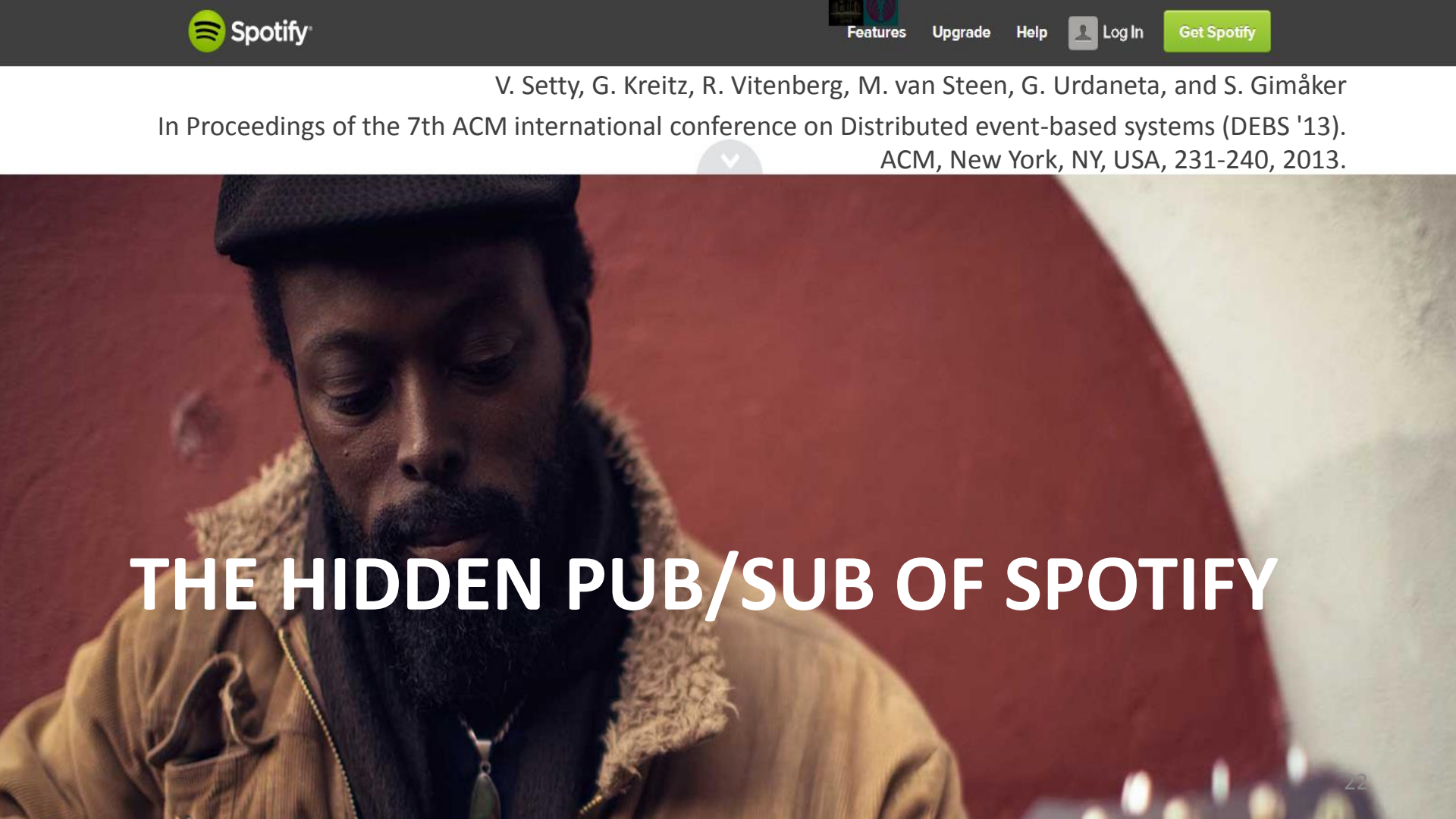
Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 6.8**

Subscription models – from course to fine grain filters

- Channel based: only physical channel
- Topic (subject) based: fields and one is the topic, can build hierarchies
- Content based: queries over range of fields
- Type based: types of events, matching over types or sub-types
- Objects of interest: focus on changes in state of objects
- Context based: associate events to locations
- Complex event processing: “inform me if *A* happens concurrently to *B* but not to *C*”

Main concern

- Deliver events efficiently to all subscribers that have filters that match the events
  - Security
  - Scalability
  - Failure handling
  - Quality of Service (QoS)
- Tradeoffs:
  - Latency/reliability
  - Ease in implementation / expressive power to specify events of interest



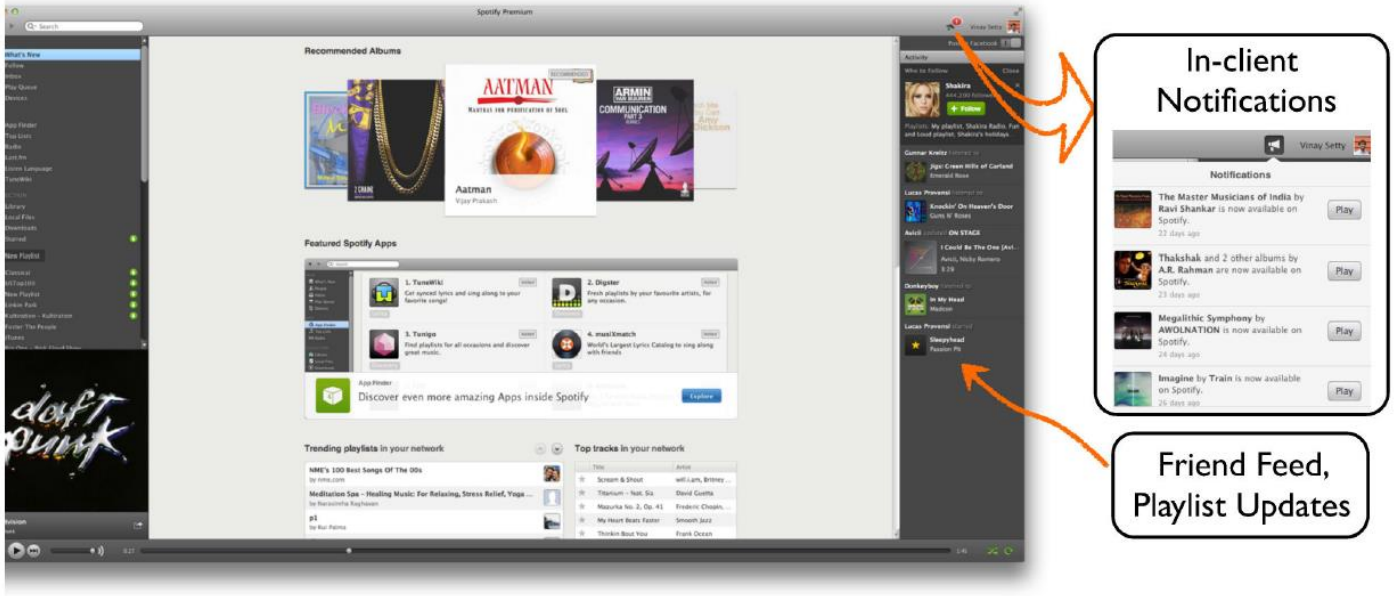
V. Setty, G. Kreitz, R. Vitenberg, M. van Steen, G. Urdaneta, and S. Gimåker

In Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS '13). ACM, New York, NY, USA, 231-240, 2013.

Some details

- End-to-end architecture to support social interaction
- Topic-based subscriptions
- Hybrid engine
  - Relay events to online users in real time
  - Store and forward selected events to offline users
- DHT based overlay
  - 3 sites: Stockholm Sweden, London UK, Ashburn USA
- Design to scale
  - Stores approx., 600 million subscriptions at any given time
  - Matches billions of publication events every day

Desktop client



Friend Feed, Playlist Updates

Push notification

Topic-based subscription

subscription(user_name, topic_name)

- Types of topics
  - Friends (Spotify + Facebook): FB friends who are Spotify users and by sharing music
  - Playlists (URI): other users playlists (updates), “Collaborative” playlists or only modifiable by creator
  - Artists pages (follow artist): new albums or news related to artist

Publication events

- All events delivered in real time (best effort and guaranteed delivery) to online users
- Some notifications are sent by email to retrieve in the future
- Example, new album from famous artist added
  - Instant notification sent to online followers
  - Email notification to offline followers
  - Event persisted so that (new) followers can retrieve it in the future (e.g., from another device)

Publication events

- Friend feed
  - Event notification to all friends following user
    - Play a track, create or modify playlist, add a favorite(artist, track, album)
    - Publish event on Facebook wall (optional)

Publication events

- Playlist updates
  - Event notifications when
    - A playlist is modified (adding or removing track, renaming playlist) via friend feed
  - Synchronize playlist across all devices of all subscribers of the playlist

Publication events

- Artist pages
  - Notification sent to followers of artist when
    - New album added in *Spotify*
    - Playlist created by artist

Publish-subscribe Architecture

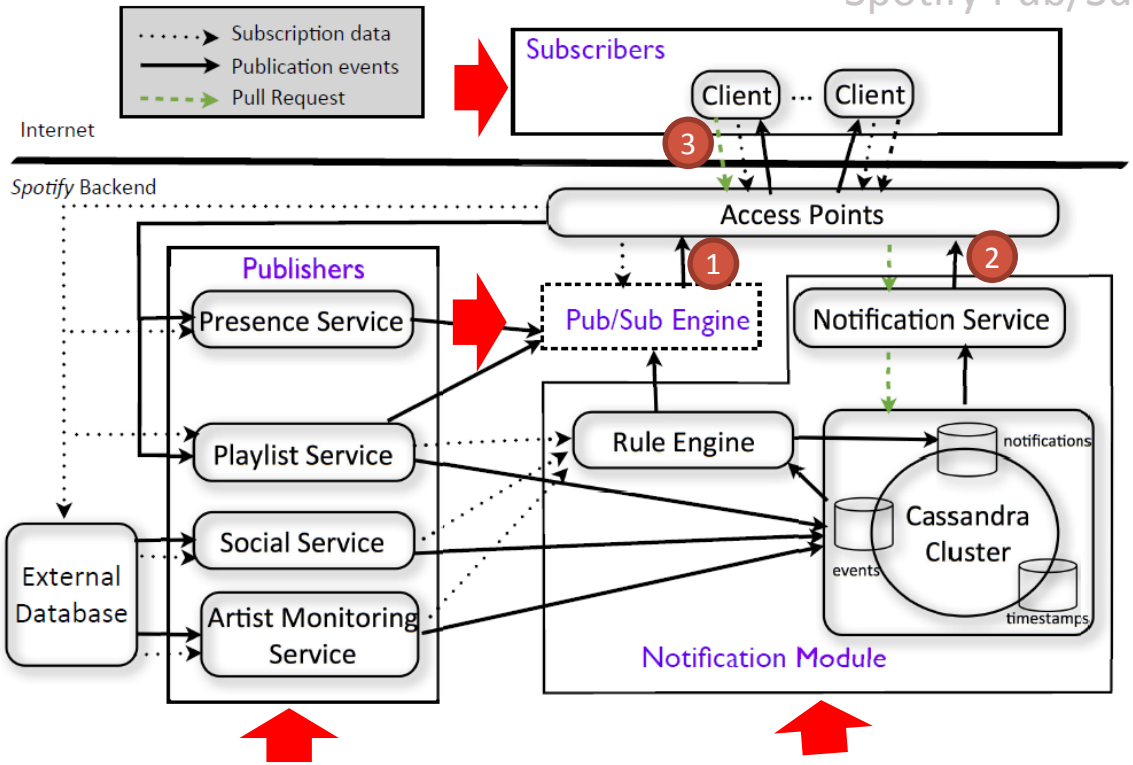
Architecture for supporting social interaction

Latency/Scalabilty tradeoff

⇩

3 Event flow paths :

1. Real time to online clients
   - No persistence, best effort, low latency
2. Persisted to online clients
   - Critical publications → Persistent, Reliable delivery, at least once across devices
3. Persistent to offline clients
   - Clients come only → pull notifications with a timestamp



Publishers and Subscribers

Access Points

- Interface to clients
- Subscription to pub/sub system
- Publications to clients

Subscribers

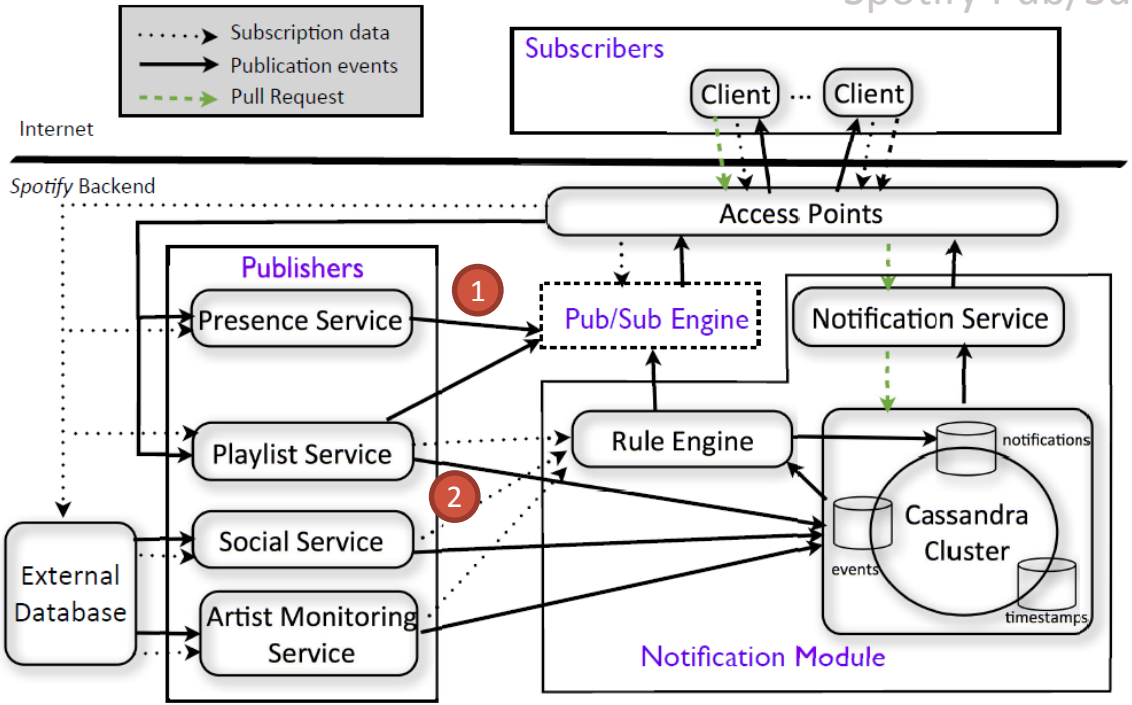
- Clients
- Subscription:

  user_name, service_URI

  1. To Pub/Sub Engine
  2. To Notification Module

Publishers

- Presence Service
- Playlist Service
- Social Service
- Artist Monitoring Service



Notifications …

Notification module

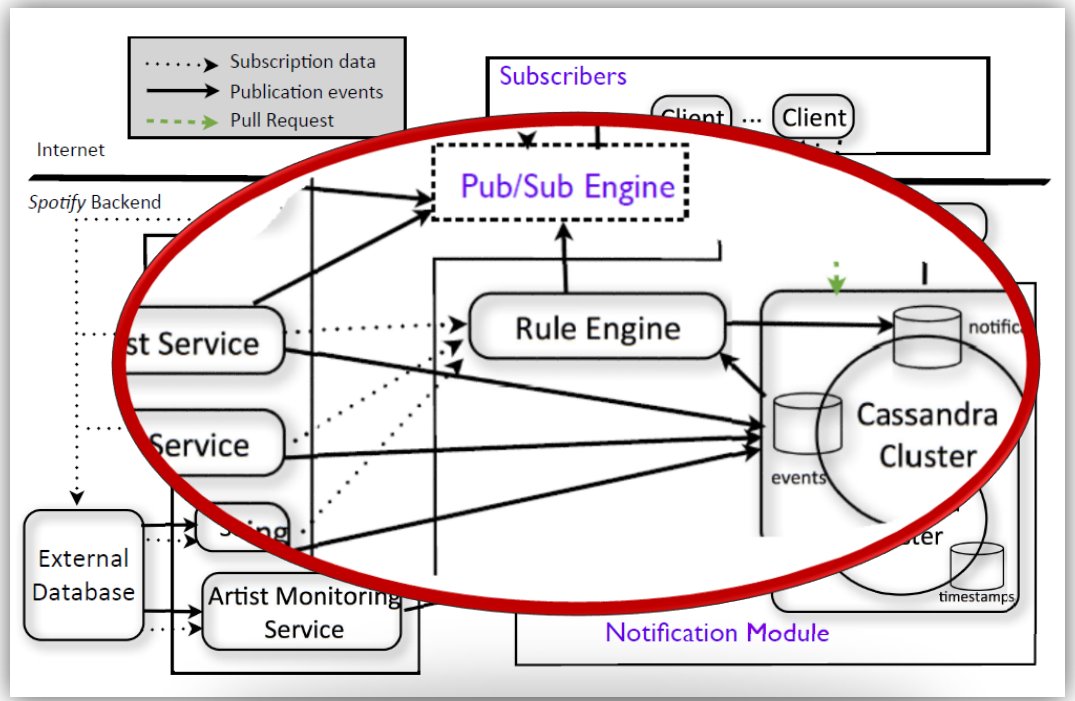
1. Receives publication events
2. Classifies events
3. Delivers events to clients
4. Pull requests (client connects)

Notification types

- In-client (guaranteed delivery)
- Push (mobile devices)
- Email

Even classification (Rule Engine)

- Online status of user
- Client device type
- User subscription preferences



Event Persistence

- Reliability
- Offline delivery
- Future retrieval
- Multiple clients delivery

How do the topics look like?

Table 1: List of topic types and corresponding services

Topic Type	URI	Service	Notification Type
User	hm://presence/user/<user-name>/	Presence	Friend-feed
Playlist	hm://playlist/user/<user-name>/playlist/<playlist-id>/	Playlist	Friend-feed, In-Client, Push and Email
Artist	hm://notifications/feed/artist-id%notification-type/	Artist Monitoring	In-Client, Push and Email
Social	hm://notifications/feed/username%notification-type/	Social	In-Client, Push and Email

Summary

- Indirect communication uses an intermediary and hence have no direct coupling between sender and receiver(s)
- Space uncoupling and time uncoupling
- Strategies:
  - Group communication
  - Message queues
  - Distributed shared-memory
  - Tuple spaces
  - Publish-subscribe systems

- Publish-subscribe systems
  - Programming model
  - Subscription models (filters)
  - Main concern
  - Tradeoffs
- Spotify publish-subscribe system
  - When it is used
  - How it is designed