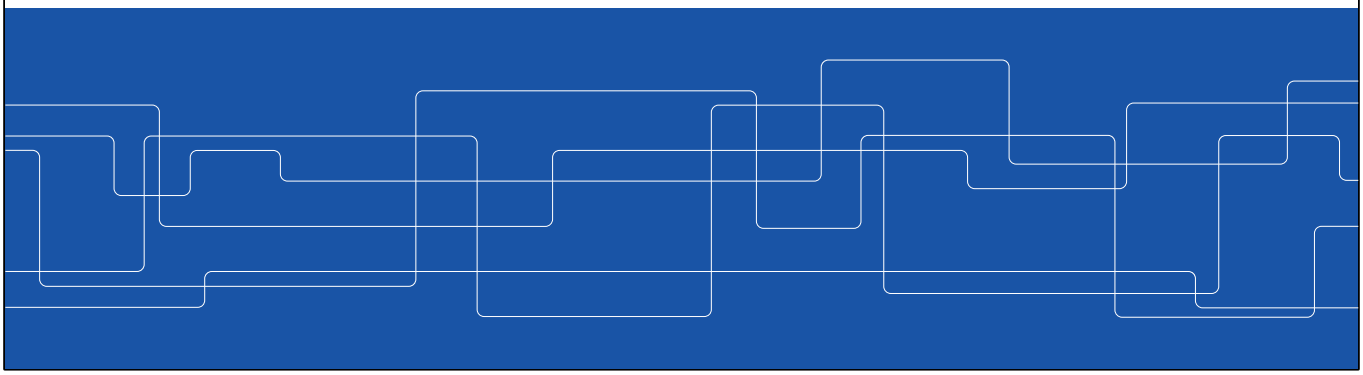
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Indirect Communication

Vladimir Vlassov and Johan Montelius



Time and Space

In ***direct communication, the sender and the receiver exist simultaneously and know*** each other.

In ***indirect communication,*** we relax these requirements: ***the sender and the receiver are uncoupled*** (or decoupled)

Time and space uncoupling

Time uncoupling: a sender can send a message even if the receiver is still not available.

The message is stored and picked up at a later moment.

Space uncoupling: a sender can send a message but does not know to whom it is sending nor if more than one, if anyone, will receive the message.

	time coupled	time uncoupled
space coupled	direct communication	message storing systems; email
space uncoupled	Broadcast; IP multicast	group communication; pub/sub

Space coupling - Time-coupled: Properties: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment. Examples: Message passing, remote invocation

Space coupling - Time-uncoupled: Properties: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes. Examples: email

Space uncoupling - Time-coupled: Properties: The sender does not need to know the identity of the receiver(s); the receiver(s) must exist at that moment. Example: IP multicast

Space uncoupling - Time-uncoupled: Properties: The sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes Examples: Most indirect communication paradigms covered in this chapter

Indirect Communication

- ***group communication***
- publish-subscribe
- message queues
- shared memory

Group communication

More than simple multicast:

- the group is well defined and managed
- ordered delivery of messages
- fault-tolerant, delivery guarantees
- handles multiple senders

Broadcast vs Multicast

No one keeps track of who listens in a ***broadcast*** service, cf., radio broadcast, IP broadcast 192.168.1.255, etc.

In a ***multicast*** service, the sender sends a message to a specific group; the system keeps track of who should receive it, cf. IP-multicast 239.1.1.1

IP-multicast is unreliable and does not keep track of members or the order of messages when we have several senders.

"**Broadcast** domain" refers to a subnet network, such as 192.168.1.0/24, with a **broadcast** address of **192.168.1.255**. "**Broadcast** domain" refers to all within the **broadcast** range or all **IP** addresses that will receive a **broadcast message** within the **Internet Protocol** subnet.
A **broadcast domain** is a logical division of a computer network in which all nodes can reach each other by broadcasting at the data link layer. A broadcast domain can be within the same LAN segment or bridged to different LAN segments.
Space uncoupling - Time-coupled: Properties: The sender does not need to know the identity of the receiver(s); the receiver(s) must exist at that moment. Example: **IP multicast**

Ordering of messages

- ***FIFO order***: All messages are received in the order sent.
- ***Causal order***: If a message m_2 is sent as a consequence of a message m_1 (i.e., a process has seen m_1 and then sends m_2), then all members should see m_1 before m_2.
- ***Total order***: All members will see messages in precisely the same order.

Causal ordering does not strictly imply FIFO; a process can send m_1 and then m_2 but has not yet seen its message m_1.

We can observe events; what do we know about causality?

Implementations

- **JGroup**: Java based
- **Akka**: Scala based
- **Spread**: C++ based
- **pg**: a not so advanced library in Erlang

Indirect Communication

- group communication
- ***publish-subscribe***
- message queues
- shared memory

Publish-subscribe

Processes *publish events*, not knowing if anyone is interested.

A process can *subscribe to events* of a given class.

Limited guarantees on ordering or reliability - scales well.
Used when the flow of events is very high:
Trading platforms, news feeds, etc.

A publish-subscribe system is a system where ***publishers*** publish structured events to an event service and ***subscribers*** express interest in particular events through ***subscriptions*** which can be arbitrary patterns over the structured events. Publish-subscribe systems have two main characteristics: Heterogeneity; Asynchronicity: Notifications are sent asynchronously by event-generating publishers to all the subscribers that have expressed an interest in them to prevent publishers from needing to synchronize with subscribers – publishers and subscribers need to be decoupled.
Operations: **publish(event); subscribe(filter); unsubscribe(f**); When events arrive at a subscriber, the events are delivered using a **notify(e)** operation.
Some systems complement the above set of operations by introducing the concept of advertisements. With advertisements, publishers can declare the nature of future events through an **advertise(f)** operation. Also, unadvertise(f)

Subscriptions

IOT Systems, Ex: Forest: Channel About Humidity, Channel Rain, Temp Channel and etc..

Sensor : Fire Detect

- ***Channel***: events are published to a channel that processes can subscribe to.
- ***Topic (Subject)***: an event is published given one or more topics (#foo). If topics are structured in a hierarchy, processes can choose to subscribe to a topic or a sub-topic.
- ***Content***: subscribers specify properties of the content, more general - harder to implement
- ***Type***: used by object-oriented languages; subscribe on an event of a particular class

MetaData Provide more subscriptions

Channel-based: In this approach, publishers publish events to named channels, and subscribers then subscribe to one of these named channels to receive all events sent to that channel. Example: CORBA Event Service
Topic-based (also referred to as subject-based): In this approach, we assume that each notification is expressed in several fields, with one field denoting the topic.
Content-based: Content-based approaches are a generalization of topic-based approaches allowing the expression of subscriptions over a range of fields in an event notification. More specifically, a content-based filter is a query defined in terms of constraints over the values of event attributes.

Implementation

How do you implement a pub/sub system?

It's simple - one central server that keeps track of all subscribers.

Availability? use two servers

Scalability? use a distributed network of event brokers

The task of a publish-subscribe system is clear: to ensure that events are delivered efficiently to all subscribers that have filters defined that match the event.
First option: Centralized versus distributed implementations. In distributed implementations of publish-subscribe systems, the centralized broker is replaced by a network of brokers that cooperate…

Broker networks

In between the group needs information

A network of ***brokers*** that distribute events; clients connect to the brokers. Near

The ***network of brokers*** forms an ***overlay network*** that can route events.

Given a broker network, how do we distribute events from publishers to subscribers?

The implementations of channel-based or topic-based schemes are relatively straightforward. For example, a distributed implementation can be achieved by mapping channels or topics onto associated groups (as defined in Section 6.2) and then **using the underlying multicast communication facilities to deliver events** to interested parties (using reliable and ordered variants, as appropriate). **The distributed implementation of content-based (or by extrapolation, type-based) approaches is more complex**…

Event routing

The ***event routing*** depends on our subscription model and performance, fault tolerance, availability, and consistency requirements.

- Flooding
- Filtering
- Advertisement
- rendezvous

The more advanced the subscription mechanism, the more complex the routing mechanism.

The heart of the architecture is provided by **the event routing layer supported by a network overlay infrastructure.** Event routing ensures that event notifications are routed as efficiently as possible to appropriate subscribers. The overlay infrastructure supports this by setting up appropriate networks of brokers or P2P structures.

Flooding

- send all published events to all nodes in the network
- each node does matching
- can be implemented using underlying network multicast

Simple but inefficient - events are distributed even if no one is subscribing.

Alternative - let the subscriptions flood the network, and publishers keep track of subscribers.

The most straightforward approach is based on flooding, sending an event notification to all nodes in the network, and then carrying out the appropriate matching at the subscriber end.
As an alternative, flooding can be used to send subscriptions back to all possible publishers. The matching is done at the publishing end, and matched events are sent directly to the relevant subscribers using point-to-point communication. Flooding can be implemented using an underlying broadcast or multicast facility.

Filtering

Let the brokers take a more active part in the publishing of events.

- a subscription is sent to the closest broker
- brokers share information about subscriptions
- a broker knows which neighboring brokers should be sent published events

Requires a more stable broker network

How do we implement content-based subscriptions?

One principle that underpins many approaches is applying filtering in the brokers' network. This is referred to as filtering-based routing. Brokers forward notifications through the network only where there is a path to a valid subscriber. This is achieved by propagating subscription information through the network toward potential publishers and storing associated states at each broker. Specifically, each node must maintain a neighbors list containing a list of all connected neighbors in the network of brokers, a subscription list containing a list of all directly connected subscribers serviced by this node, and a routing table. Crucially, this routing table maintains a list of neighbors and valid subscriptions for that pathway.

Advertisement

Let the publishers advertise that they will publish events of a particular class.

- Publishers advertise event classes
- advertisements are propagated in the network
- subscribers contact publishers if they are interested.

It can be combined with filtering.

The pure filtering-based approach described above can generate a lot of traffic due to the propagation of subscriptions, with subscriptions essentially using a flooding approach towards all possible publishers. In systems with advertisements, this burden can be reduced by propagating the advertisements toward subscribers similarly (actually, symmetrical) to the propagation of subscriptions. There are interesting trade-offs between the two approaches, and some systems adopt both approaches in tandem.

Rendezvous

An advertisement approach can overload a frequent publisher; all subscribers need to talk to the publisher.

Distribute the load by delegating the subscription handling to another node.

How do we select the node that should be responsible for a particular class?

Rendezvous nodes are broker nodes **responsible** for a given subset of the event space. Function SN(s) takes a given subscription, s, and returns one or more rendezvous nodes that take responsibility for that subscription. Each rendezvous node maintains a subscription list as in the filtering approach and forwards all matching events to the set of subscribing nodes. **When an event *e* is published**, the function EN(*e*) also returns one or more rendezvous nodes, this time responsible for matching *e* against subscriptions in the system. **DHT** can be used for this…

Pub/Sub Systems

Often part of a messaging platform:

- Java Messaging Service (JMS)
- ZeroMQ
- Redisi69
- Kafka

Or a separate service:

- Google Cloud Pub/Sub

Several standards:

- OMG Data Distribution Service (DDS)
- Atom - web feeds (RSS), clients poll for updates

Indirect Communication

- group communication
- publish-subscribe
- ***message queues***
- shared memory

Message queues

A ***queue*** (normally FIFO) is an object independent of processes.

Processes can:

- send messages to a queue
- receive messages from a queue
- poll a queue
- be notified by a queue

More structured and reliable compared to pub/sub systems.

Implementations

Queues could be running on either node in the system, but we need a mechanism to find the queue when sending or receiving.

A central server is a simple solution but does not scale.

A binder can be responsible for keeping track of queues.

- WebSphere MQ by IBM
- Java Messaging Service
- RabbitMQ
- ZeroMQ
- Apache Qpid

Standard: AMQP - Advanced Message Queuing Protocol.

Erlang message queues

In Erlang, message queues are similar but different:

- a queue is attached to a process: one queue - one receiver
- the queue is not persistent: if the process dies, the queue dies
- there is only a blocking receive (but you can use a timeout)
- only intended for Erlang process communication

Indirect Communication

- group communication
- publish-subscribe
- message queues
- ***shared memory***

Shared memory

Why not make it simple - if concurrent threads in a program can communicate using shared memory, why would it not be possible for distributed processes to do the same?

A distributed shared memory - DSM.

Parallel computing

Shared memory is mostly used in shared-memory multiprocessors, multi-core processors and computing clusters where all nodes are equal and run the same operating system.

Shared-memory architectures:

- UMA: uniform memory access
- NUMA: non-uniform memory access
- NUCA: non-uniform cache access (in multi-core processor)
- COMA: cache-only memory access

High-performance computing systems also use message passing rather than shared memory to scale better.

Tuple spaces

A shared memory on a higher level - a ***shared tuple space***.

- write: add a tuple to the store
- read: find a matching tuple in the store
- take: remove a matching tuple from the store

It was made famous by the Linda coordination language in 1986.

Implementing tuple spaces

A centralized solution is simple ... but does not scale.

Distributed implementation is much harder:

- write: replicate the tuple, make sure that all replicas see the tuple
- read: read from any replica
- take: more problematic, how does it conflict with a concurrent write operation

The distributed implementation uses several spaces to reduce conflicts.

Object spaces

A more general form replaces tuples with objects.
Example: JavaSpaces included in Jini.

Summary

Communication is uncoupled in space and time.

- group communication
- publish-subscribe
- message queues
- shared memory

ID2201 Distributed Systems / Indirect Communication

Lecture starts 15:15