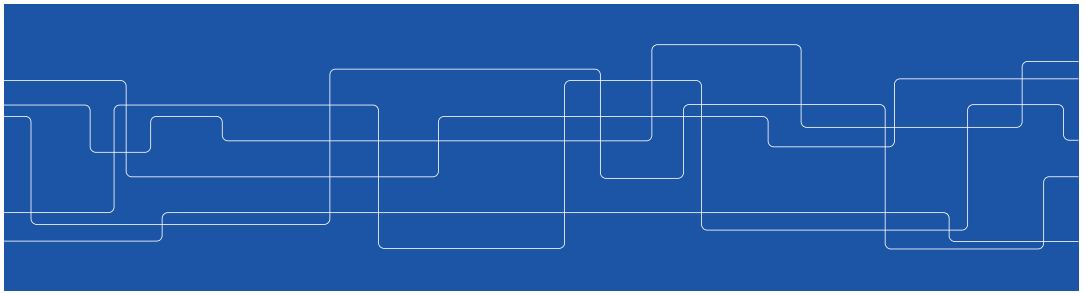
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Indirect Communication

Vladimir Vlassov and Johan Montelius



Time and Space

In ***direct communication*** sender and receivers exist in the same time and know of each other.

In ***indirect communication*** we relax these requirements: sender and 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
space uncoupled	broadcast	group communication

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

In a ***broadcast*** service, no one keeps track of who is listening, cf. radio broadcast, IP broadcast 192.168.1.255 etc.

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

IP-multicast is unreliable, does not keep track of members nor order of messages when we have several senders.

Ordering of messages

- ***FIFO order***: All messages are received in the order sent.
- ***Causal order***: If a message m2 is sent as a consequence of a message m1 (i.e. a process has seen m1 and then sends m2), then all members should see m1 before m2.
- ***Total order***: All members will see messages in exactly the same order.

Causal ordering does not strictly imply FIFO, a process can send m1 and then m2 but has not yet seen its own message m1.

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 on 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.

Subscriptions

- ***Channel***: events are published to channel that processes can subscribe to.
- ***Topic (Subject)***: a event is published given one or more topics (#foo), if topics are structured in a hierarchy processes can be choose to subscribe on 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 event of a particular class

Implementation

How do 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

Broker networks

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

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

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

Event routing

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

- Flooding
- Filtering
- Advertisement
- rendezvous

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

Flooding

- send all published event to all nodes in the network
- matching is done by each node
- 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.

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?

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

Can be combined with filtering

Rendezvous

An advertisement approach can overload a frequent publisher, all subscribers needs 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?

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 that is 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, similar to in RPC can be the responsible for keeping track of queues.

- WebSphereMQ 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 a shared memory why would it not be possible for distributed process 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

Made popular by the Linda coordination language from 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

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, uncoupled in space and time.

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