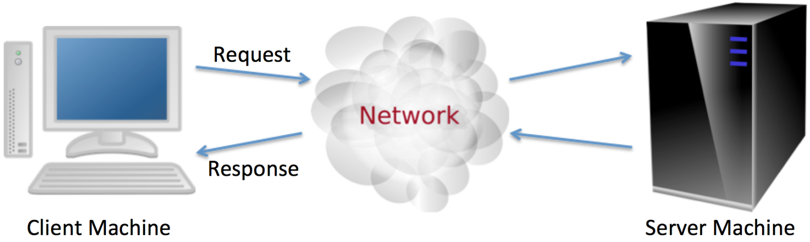
Publish-Subscribe Systems & Message-Oriented Middleware

Sistemas Distribuídos 2014/2015

So far we studied synchronous interactions...

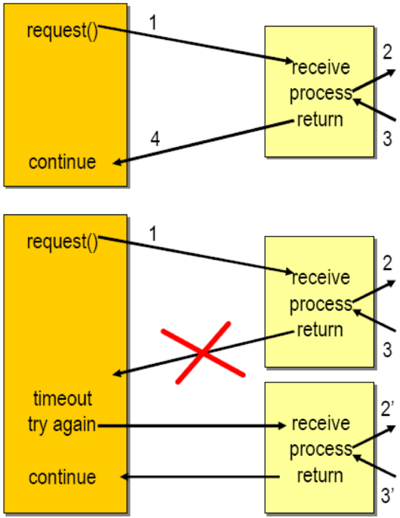


- Interprocess communication (Coulouris, Ch. 4) and remote invocation (Coulouris, Ch. 5).

Synchronous interactions

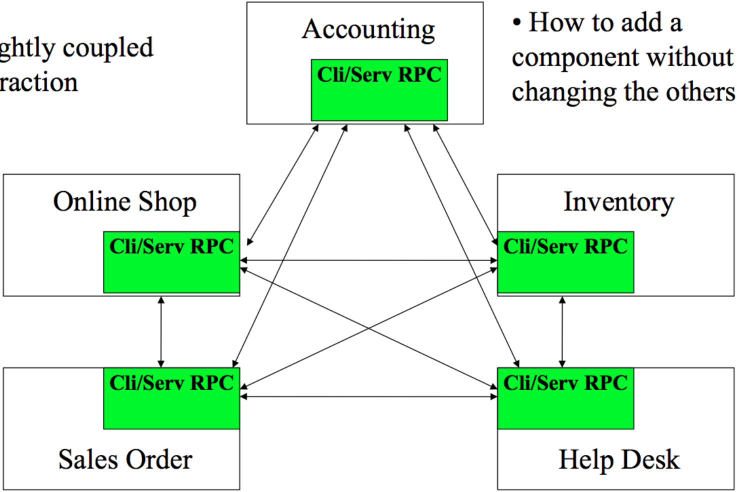
- There is a direct coupling between a sender and a receiver.
- This leads to a certain amount of rigidity in the system in terms of dealing with change.
- Because of the direct coupling, it is more difficult to replace a server.
- If the server fails, this directly affects the client, which must explicitly deal with the failure.

Failure scenarios in synchronous invocations



Limitations of synchronous invocations

- Tightly coupled interaction



- How to add a component without changing the others?

Indirect communication

- Communication between entities in a distributed system through an *intermediary*.
- No direct coupling between the sender and the receiver(s).
- Many indirect communication paradigms explicitly support one-to-many communication.

Interesting properties

- **Space uncoupling.** The sender does not know or need to know the identity of the receiver(s), and vice versa. Participants senders or receivers can be replaced, updated, replicated or migrated.
- **Time uncoupling.** The sender and receiver(s) do not need to exist at the same time to communicate. This has important benefits, for example, in more volatile environments where senders and receivers may come and go.

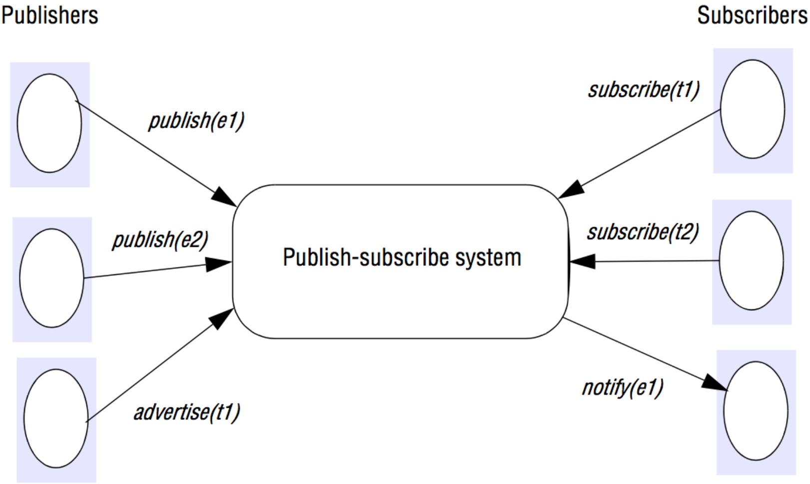
Models of interaction

	Time-coupled	Time-uncoupled
Space-coupled (sender knows receivers)	Communication directed toward a given receiver or receivers; receiver(s) must exist at that moment in time.	Communication directed toward a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes.
Space-uncoupled (sender does not know receivers)	Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time.	Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes.

Publish-subscribe systems

- A *publish-subscribe system* is a system where publishers publish structured events to an event service.
- Subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.
- The task of a publish-subscribe system is to match subscriptions against published events and ensure the correct delivery of event notifications.
- Publish-subscribe is fundamentally a one-to-many communications paradigm.

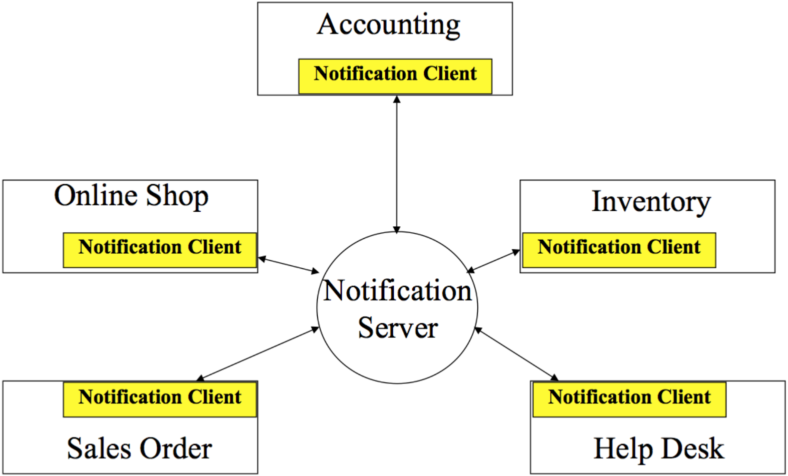
Specifying event interests and receiving notifications



Publish-subscribe systems

- Some applications require:
  - Asynchronous notifications (time-uncoupled).
  - Information based on contents rather than source.
- Examples: financial information systems, RSS feeds (live feeds of real-time data), Google's ad-clicks dissemination to interested parties, monitoring applications...
- For some applications it is not enough for subscriptions to express queries over individual events $\leadsto$ complex event processing.
- Pub/sub systems can be seen as one way of managing multicast groups.

Same shop with event-based communication



Examples of publish-subscribe implementations

- CORBA Event Service
- TIB Rendezvouz
- Scribe
- TERA
- Siena
- Gryphon
- Hermes
- MEDYM
- Meghdoot
- Structure-less CBR

There are some drawbacks

- The structure of data/events is inflexible (hard to modify publishers).
- Message delivery guarantees are not always assured by the implementation (may lead to unconventional solutions, such as the subscriber publishing acknowledgements).
- The publisher may be unaware that critical events are being missed by a faulty subscriber.
- Difficult to guarantee scalability during load surges, as well as to guarantee security.

Message-oriented middleware

- Message queues provide a point-to-point service using the concept of a message queue as an indirection.
- Point-to-point: the sender places the message into a queue, and it is then removed by a single process.
- Distributed message queues are also referred to as message-oriented middleware.

Three typical ways to receive messages

- **Blocking receive**, which will block until an appropriate message is available;
- **Non-blocking receive** (a polling operation), which will check the status of the queue and return a message if available, or a not available indication otherwise;
- **Notify operation**, which will issue an event notification when a message is available in the associated queue.

Support for transactions

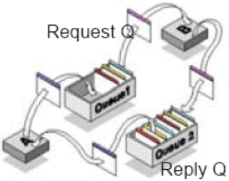
- Most commercially available systems provide support for the sending or receiving of a message to be contained within a transaction.
- The goal is to ensure that all the steps in the transaction are completed, or the transaction has no effect at all (the “all or nothing” property).

Queues

TX 1:
Start
  get input
  construct request
  enqueue request
Commit

TX 3:
Start
  dequeue reply
  decode reply
  process output
Commit

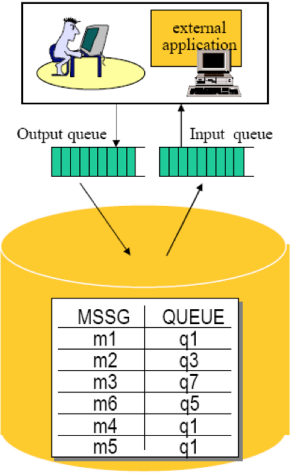
Message queues are good for asynchronous point to point (1:1) messaging



Server B

TX 2:
Start
  dequeue request
  process request
  enqueue reply
Commit

Queues


external application
Output queue
Input queue
MSSG QUEUE
m1 q1
m2 q3
m3 q7
m6 q5
m4 q1
m5 q1

Message-oriented middleware

- Ensures that messages are properly distributed among applications.
- Provides fault tolerance, load balancing, scalability, and transactional support for reliable exchange of messages in large amounts.
- Vendors implement their own mechanism, but provide the same API for message manipulation (e.g., JMS, Java Message Service).

MOM properties

- Allows messages to be prioritized;
- Delivers messages either synchronously or asynchronously;
- Guarantees messages are delivered once and only once;
- Supports message delivery notification;
- Supports message time-to-live;
- Supports transactions.

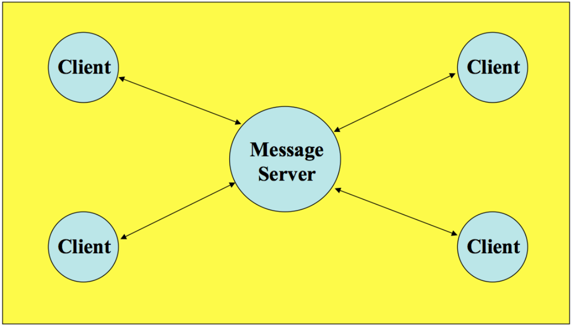
MOM architectures

- Centralized architecture
- Decentralized architecture
- Hybrid architecture

Centralized architecture

- Relies on a message server (also called message router/broker).
- Message server delivers messages from one messaging client to others.
- Decouples a sending client from other receiving clients.

Centralized architecture (event broker)



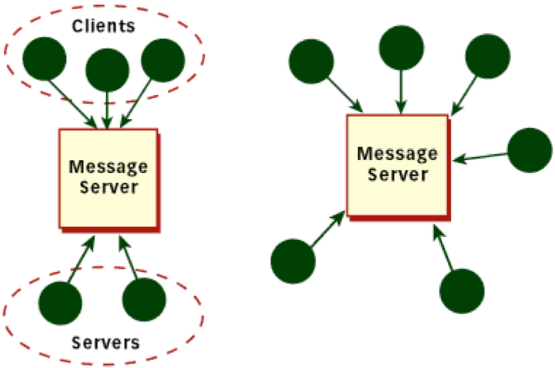
Decentralized architecture

- Uses IP multicast at the network level.
- Server functionality is embedded as a local part of the client.
  - Persistence
  - Transactions
  - Security

Hybrid architecture

- Clients may connect to a daemon process, which in turn communicate with other daemon processes using IP multicast groups.

Client-server and point-to-point using MOM



Main advantages

- **Messaging decouples resources.** It allows business components to be combined into a flexible system with extremely loose coupling between components. Makes the application more reliable (a failure in one part of the application is less likely to affect an unrelated part of the application).
- **Messaging provides scalability.** An application built around a messaging architecture scales well as both clients and servers are added to the system. (Can we take this for granted?)
- **Messaging masks both heterogeneity and change.** The common element in a messaging application is the message. As long as components can read and understand the message, the platform on which they reside and the languages in which they're written are unimportant.

Commercial products

- IBM WebSphere MQ (MQ Series)
- Oracle Advanced Queuing (AQ)
- Tuxedo
- MessageQ
- TIBCO Rendezvous

Applications of MOM

- Suited to applications with time- independent responses
- Most appropriate for loosely-coupled, event-driven applications.
- Particularly good for Internet-based software solutions which require a fast, reliable and scalable link between broad variety of applications and data sources
- Used for enterprise application integration

MOM versus RPC

	MOM	RPC
metaphor	Post office	Telephone
Client/server communication	Asynchronous	Synchronous
Client/server sequence	None	Server must be up before client
Paradigm	Queued	Call-Return
Partner needs to be available	No	Yes
Load balancing	Single queue can be used to implement FIFO or priority-based policy	Requires a separate TP Monitor
Transactional support	Yes, in some products.	No. Requires transactional RPC
Message Filtering	Yes.	No.
Performance	Slow, because of queue	Fast

Properties of message queues

Property	Explanation
Real enough time	In most applications the processing does not need to occur in real time. Thus real enough time is acceptable.
Priorities	Messages can be stored in the servers queue according to their priorities. Thus the server processes the highest priority messages.
Reliability	Messages can be stored on disk . Thus messaging systems are less prone to errors as a result of application or system failure.
Scalability	Queues buffer requests thus making it possible to tune the number of processes servicing requests during peak-load periods.

Bibliography

- George Coulouris *et al.*, Chapter 6, Distributed Systems: Concepts and Design, 5th edition, 2011.