Message oriented programming

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Limitations of direct remote invocation

- Remote invocation in client-server:
  - inspired by classic procedure invocation in sequential applications.
  - invocation typically synchronous, returning result;
  - client knows the identity of server;
  - client and server coexist in time;
- Strong coupling between client and server.

Uncoupling communication

- In many cases it can be useful to break the coupling.
- Examples:
  - work-load that can be processed in several stages;
  - work-load that can be processed in map-reduce pattern;
  - traders want to know when price of a stock reaches certain value;
  - sport journalists want to be informed about game results;
  - water control software wants to be informed about rain levels collected by many geographically spread sensors;
- In many cases, remote invocation to a central server is not the most appropriate solution.

Message based systems

- Alternative to client/server designs;
- System is composed by elements glued by messaging patterns;
- Supported by MOM – *message oriented middleware*;
- Set of rich standardized messaging semantics offered;
- Programmer writes mostly sequential code, with little or no concurrency control;
- MOM handles concurrency and distribution;

Some MOM roles

- Connection establishment, reconnection, polling;
- Message sending/receiving, including framing;
- Message queueing (and persistence in some cases);
- Load-balancing and fair queueing;
- Intermediation between parties, to glue and uncouple elements;
- Efficient message routing in large networks with broker nodes;

Messaging patterns

Some important messaging patterns:

- Request-reply;
- Pipeline with scatter-gather;
- Publish-subscriber;

Request-reply

- Client-server pattern;
- MOM can provide load-balancing to several servers;
- MOM can provide fair queueing of requests from several clients;
- Server-code can be programmed with no concurrency control;

Pipeline with scatter-gather

- Can be used to process workload in stages;
- Several stages can be pipelined;
- In each stage work can be spread to several workers (scatter);
- Work can be collected from several workers (gather) from the previous stage;
- MOM provides fair queueing and load-balancing;
- Server-code can be programmed with no concurrency control;
- The well-known map-reduce pattern is a special case;

Publish-subscriber

- Publisher produces messages (events);
- Subscribers register interest in receiving some kind of events;
- Based on the observer pattern;
- MOM routes events from publishers to relevant subscribers;
- These MOMs can be called *event notification systems*;
- We will look in some detail at this messaging pattern later;

Broker based vs brokerless designs

- We can broadly divided MOMs in either:
  - broker-based: provide a broker, to intermediate “clients”;
  - brokerless: each participating node can act in many roles, possibly connecting to others with no intermediation;
- Most classic MOM protocols assume a broker-based design:
  - the broker is an application, which is started in some node;
  - XMPP: clients interact through a broker, e.g., ejabberd;
  - AMQP: clients interact through a broker, e.g., RabbitMQ;
- Brokerless designs adopted by lower level *messaging fabric*:
  - libraries to be linked with the distributed application;
  - example: ZeroMQ middleware, used as library;
  - direct communication between nodes possible;
  - but a node can be designed to take a broker role, if appropriate;

Event notification services

- Event notification services are a kind of message oriented middleware, based on the publish-subscribe paradigm.
- Interaction between publishers and subscribers can be intermediated by a message broker.
- Examples of MOMs supporting event notification:
  - Event and Notification service in CORBA.
  - Commercial products, like MQSeries (IBM) e MSMQ (Microsoft).
  - XMPP: Extensible Messaging and Presence Protocol.
  - AMQP: Advanced Message Queuing Protocol.
  - ZeroMQ.

Publish-Subscribe model

- Appropriate for systems where some entities produce information that is of interest to some other entities;
- Two types of entities:

publisher	produces information (events) in which others are interested;
subscriber	has interest in being informed about published events;

- Weak coupling between publishers and subscribers.
- May be used as design pattern in non-distributed applications; a variant of the observer pattern.

Terminology

Event	happening of interest that can be observed;
Notification	data describing an event;
Message	carries notifications;
Publisher	component that produces notifications;
Subscriber	reacts to notifications delivered by the MOM;
Subscription	describes sets of notifications of interest to some subscriber;
Advertisement	describes sets of notifications a publisher can publish; many times optional; aids routing;
Notification service	mediator that routes notifications from publishers and delivers them to relevant subscribers;

Features of notification services

Anonymity between publishers and subscribers:

- a publisher does not know what subscribers are interested in its events;
- a subscriber does not know which publishers publish some event.

Many-too-many the relation publishers and subscribers allows many of each publishing and consuming events.

Features of notification services

Message queues some systems allow queueing events, as opposed to delivering them only to connected subscribers.

- allows future delivery, possibly by defining event life-time criteria.
- allows subscribers not running when an event is generated; decoupling publisher and subscriber life-times.

Persistency it is useful for systems that support queues to save events persistently.

- allows event life times beyond a server crash / reboot.

Features of notification services

Real time some systems may concern with event delivery with very low latency; these can define aspects like priority levels for events.

Subscription model different notification services have different ways to allow subscribers to express criteria to specify in which events they are interested.

Subscription model

Some possible ways for subscribers to specify in which events they are interested, supported by notification services are:

channel based each kind of event is represented by a channel: an entity that publishers and subscribers use; e.g., the CORBA event service;

subject/topic based each notification has a special field that can be used to distinguish different kinds of events; comparisons and sometimes even regular expressions can be used on it;

content based a notification may have several fields (or be a key-value map); predicates over values of several fields in the notification can be used;

content + patterns predicates can be used relating different notifications correlated over time; this gives rise to *complex event processing*.

Architecture of notification services

- Different architectures may be used:

  centralized a single entity where publishers and subscribers connect; does not scale;

  tree system is composed by a set of nodes forming a tree (connected undirected graph with no cycles); facilitates routing;

  generic peer-to-peer graph, possibly with cycles, where each node knows a set of neighbors through which notifications are routed; system must detect duplicates and avoid infinite loops;

Notification routing

- In centralized systems what matters most is the efficiency of the *subscription matching algorithm*.
- For a distributed notification service to be scalable, it is important to have appropriate *notification routing algorithms*.
- These should minimize the ammount of communication; the system should aim to route a notification only to nodes with interested subscribers.
- Trade-offs between propagating subscriptions and notifications; usual assumption: many more notifications than subscriptions;
- May be important to explore forms of *multicast* to send notifications to subscribers.

Notification routing

- *Flooding*
    - simple: brokers disseminate all notifications between themselves; no subscriptions are propagated;
    - brokers with connected subscribers test subscriptions and decide which subscribers to notify;
    - many unnecessary messages sent;
- *Filter-Based Routing*
    - brokers maintain routing tables;
    - these consist of (filter, destination) pairs;
    - entries are updated according to subscriptions, which are propagated between brokers;

Specification of notification service

- Client-broker interface:
  - being C a client, N a notification, and F a filter (predicate over notifications);
  - operations: sub(C, F), unsub(C, F), pub(C, N), notify(C, N)
- Safety: a notification N is delivered to a client C:
  - only if pub(X, N) occurred;
  - sub(C, F) without a subsequent unsub(C, F), with F matching N;
  - notify(C, N) occurs at most once;
- Liveness:
  - if filter F is kept active by a client C (not unsubscribed by C after some time), there exist a time T such that all notifications published after T that match F will be delivered to C.

Order guarantees

- FIFO order:
  - if pub(C1, N1) $<$ pub(C1, N2) then for any client C that delivers N1 and N2: notify(C, N1) $<$ notify(C, N2);
  - useful and easy to enforce at each broker;
  - beware concurrent filter matching and routing at each broker;
- Causal order:
  - if pub(C1, N1) $<$ pub(C2, N2), then for any client C that delivers N1 and N2: notify(C, N1) $<$ notify(C, N2);
  - with the usual happens-before partial order defined by order in each client and by pub(C1, N) $<$ notify(C2, N);
- Total order:
  - if, for some $C1$, notify(C1, N1) $<$ notify(C1, N2), then for any C that delivers N1 and N2, notify(C, N1) $<$ notify(C, N2);
  - unreasonable to enforce in a distributed notification service.