Fall Semester 2013

cs 341 Distributed Information Systems

Chapter 10: Message-oriented Middleware

H. Schuldt

UNI BASEL

Overview

10.1 Asynchronous procedure and method calls / service invocations
- Lose coupling of applications

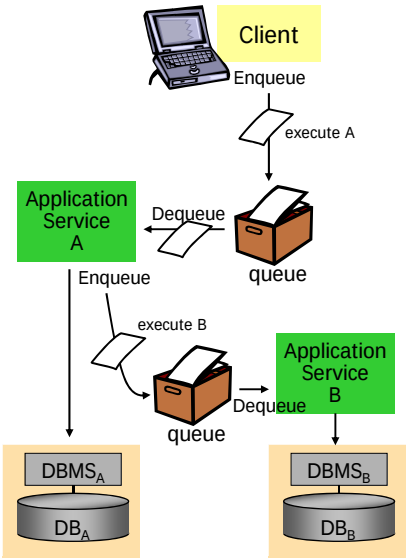
10.2 Queued Transactions
- Decoupling of client/server transactions

10.1 Asynchronous Procedure and Method Calls

- So far, we have only considered synchronous procedure, method and/or service calls:
  - Client starts a (remote) call
  - This is being processed; in the meanwhile, the client has to wait
  - After return of the result, the execution of the client program continues
- Well suited for the tight integration of different components into distributed applications / transactions
- Disadvantage of this approach
  - Requires tight coupling of all components
    - Client is blocked as long as a potentially long-running procedure or method call (which might even be propagated further along several layers) is being processed
  - When a component fails, this might have severe consequences on the overall system
    - Several clients then have to wait for the execution of a method/ procedure/service
  - Not suited at all for mobile components or for components which are temporarily not available

Message-Oriented Middleware (MOM) ...

- Asynchronous procedure, method and/or service calls are based on the exchange of messages between components
  - These contain both data as well as the specification of the procedure, method, or service that is to be called
- MOM: Message-Oriented Middleware
  - Infrastructure for sending/receiving messages
  - MOM provides APIs, clients do not have to care about the actual message transfer
  - Queue for temporarily storing messages
    - Supports asynchronous communication, even when senders or receivers are temporarily not available
    - No direct connection required



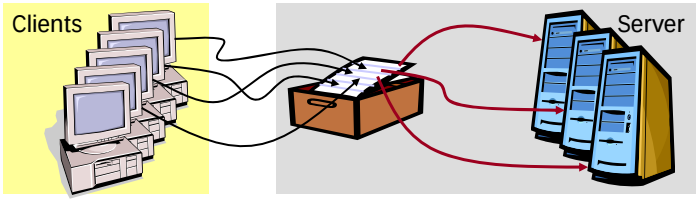
… Message-Oriented Middleware (MOM)

Different variants of MOM

- Transaction support
  - Without transaction support: MOM does not provide any guarantee that services that have been called asynchronously will actually by executed (e.g., in the case of non-persistent queues, messages can get lost when crashes occur)
  - Queued Transactions: secure exchange of messages between persistent queues of senders and receivers (see Section 10.2)
- Type of connection
  - Point-to-point: sender addresses recipient explicitly (1:1 communication)
  - Publish/Subscribe: sender does not know the recipient(s) of its messages (1:n communication possible)
- Further aspects
  - Load Balancing: choose a server (out of a server pool) to which a message is delivered in order to be processed. This uses load information of all servers
  - Priority: special treatment of „important“ messages by appropriate insertion in a queue based on their priority (priority queues)

MOM: Types of Connection …

- Point-to-Point connection (PTP)
  - Client sends message to a certain queue
  - Queue is part of the runtime environment; it distributes the message according to the address specification of the client (which means that the sender needs to be aware of the receiver)



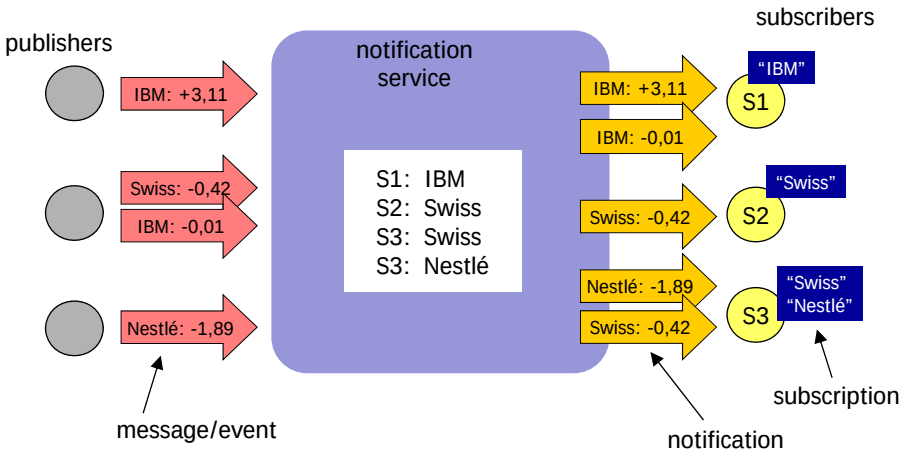
… MOM: Types of Connection

- Publish-and-Subscribe (Pub/Sub)
  - Even stronger decoupling of senders and receivers
  - Provides a powerful abstraction for building distributed applications
  - Participants are decoupled
    - in space: no need to be connected or even know each other (recipients are anonymous)
    - in flow: no need to be synchronized
    - in time: no need to be up at the same time
  - Good solution for highly dynamic, decentralized systems
    - Event-based communication between components
    - Example: coordinate complex workflows on top of distributed components
  - Applications: Stock information delivery, auctions, …
  - Pub/sub comes with a variety of names, for instance: notification service, data distribution service, continuous queries, event-based systems

Publish / Subscribe

- Basic architecture
  - Core component: Publish/Subscribe middleware (broker)
  - Receivers must register their interest beforehand with the broker (subscription)
  - The producer publishes messages or events. Those are automatically delivered to the recipients (which have declared their interest beforehand)
  - Thus, the sender does not take care of the addresses of all recipients, i.e., the sender just publishes events but does not have to wait for answers

Basic Interaction Model

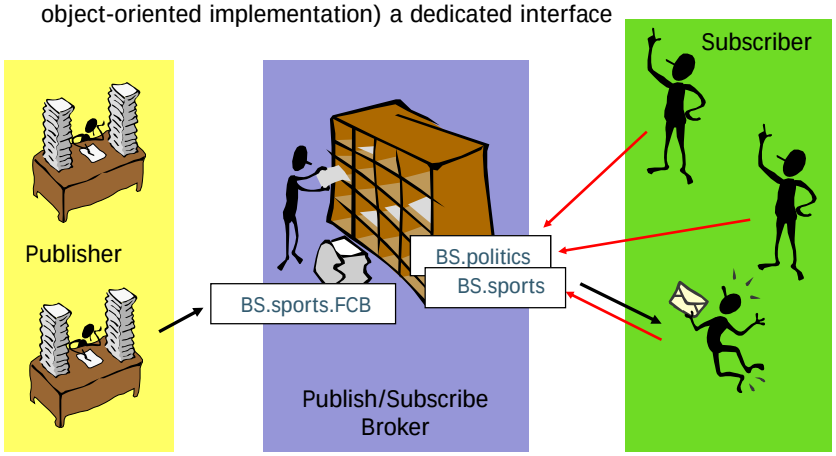


Variants of Publish/Subscribe Interactions (1)

- Pub/sub interactions differ in several ways
- The most obvious is in the way, messages or events are being described and handled
  - Topics-based
  - Type-based
  - Content-based

Topics-based Publish/Subscribe

- Basic requirement: fixed vocabulary for the description of messages/events
- Usually, this is done by using a (hierarchical) name space
  - But: this renders the approach to be quite inflexible (limited expressiveness)
  - Realization: per topic, there is either a dedicated channel, a queue or (in an object-oriented implementation) a dedicated interface

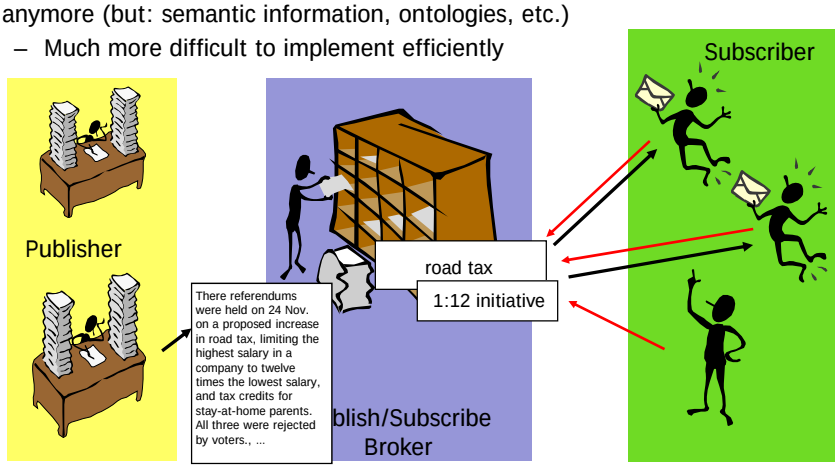


Type-based Publish/Subscribe

- Very similar to topics-based pub/sub
  - Notifications (messages or events) are objects
  - Type is the discriminating attribute, i.e., subscribers register for a particular message or event type

Content-based Publish/Subscribe ...

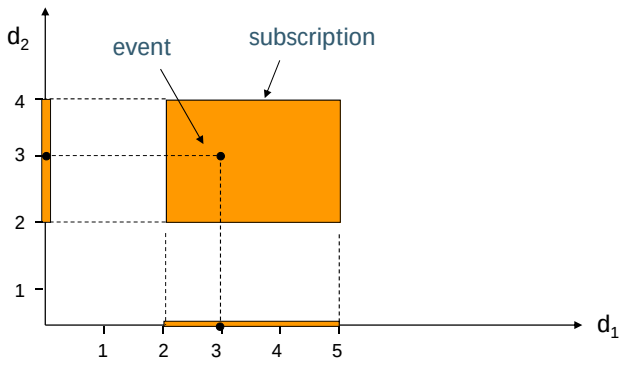
- Messages (events) are self-contained
- Subscriptions are generic queries SQL-like on the event schema ("continuous queries")
- More flexible and general approach since no common vocabulary is needed anymore (but: semantic information, ontologies, etc.)
  - Much more difficult to implement efficiently



... Content-based Publish/Subscribe

General model:

- Subscriptions and events defined over an n-dimensional event space
  - Example: StockName = "Swiss" ∧ change < -3.5
- A subscription is a conjunction of constraints
- Depending on the range of an event dimension, subscriptions can even include range constraints

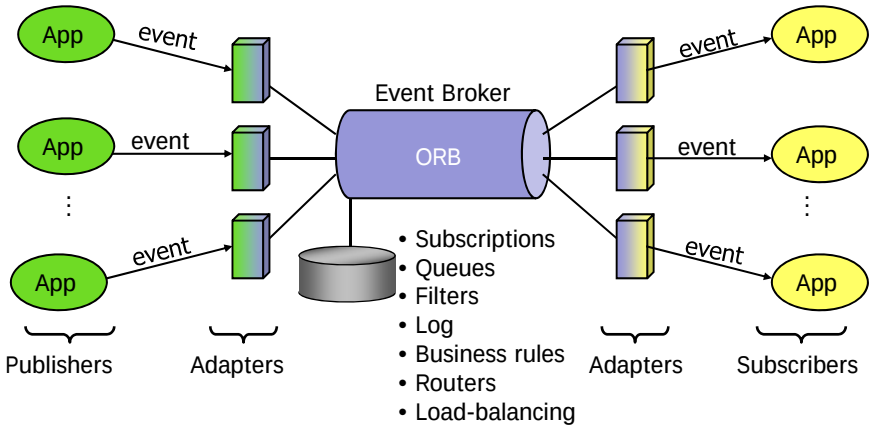


Variants of Publish/Subscribe Interactions (2)

Pub/sub systems also differ in terms of their topologies (architectures)

- Decentralized approach: Peer-to-peer (see Chapter 9)
  - Message/event is published via broadcast mechanisms (channel = group; exploited for group communication)
  - Each potential receiver has to filter incoming messages and pick the ones (s)he is interested in
  - Large communication overhead
- Centralized broker
  - Bus architecture, e.g., the CORBA Event Service
  - Hub-and-Spoke: Publish/Subscribe broker & (persistent) queues (e.g., IBM's MQSeries)

Example: Bus Architecture



Variants of Publish/Subscribe Interactions (3)

Pub/sub systems differ in the way they filter information

- Globally, by the broker
  - Allows to specify certain filter predicates when subscribing (e.g., specify ranges)
  - Broker checks these predicates before a message is forwarded to the recipient
- Locally, by the recipients
  - Messages are distributed (according to topic or content). Additional filters have to be applied by the receiver
  - This is automatically the case in a decentralized environment

Pub/sub systems differ in the way they distribute messages

- Push: automatic distribution
- Pull: recipient has to explicitly check whether there are entries in the queue (some kind of inbox) whether entries are there

Variants of Publish/Subscribe Interactions (4)

Pub/sub systems differ in the way they handle persistence of messages

- Non-persistent queues
  - Events ware published and immediately forwarded to their subscribers. If some subscribers are not online then, these messages will be lost (for them)
  - Only messages are sent which have been published after the subscription
- Persistent Queues
  - Publish/Subscribe broker persistently stores events/messages
  - Subscribers can even make use of archived information at the time they submit their subscription
  - In many products, persistence/non-persistence is a property of the message (i.e., the publisher can determine, whether or not a message is to be stored persistently)

Pub/sub systems differ in terms of transaction support

- Non-transactional transfer vs. queued transactions

Variants of Publish/Subscribe Interactions (5)

Pub/sub systems differ in terms of the numbers of recipients of messages

- Events/messages are made available to all subscribers (according to topic or content)
  - This is the general case in pub/sub interactions
- Events are sent only to one subscriber (although there could be much more)
  - For instance when choosing a recipient based on its load
    goal: load balancing

MOM: Meet the Players ...

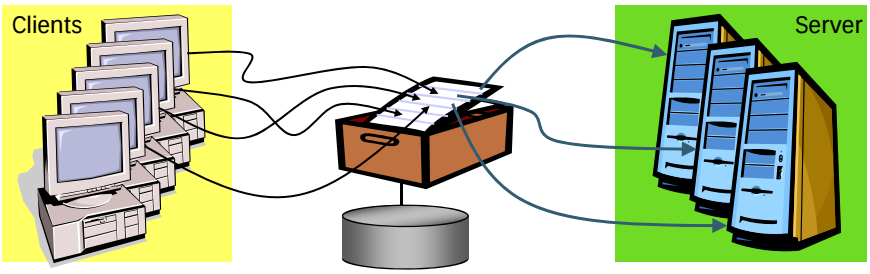
- MOM is not completely orthogonal to the infrastructure components we have discussed before ...
  ... but is –more or less well hidden– available in all systems

- CORBA: Event Service and Notification Service (both are part of the Common Object Services defined in the CORBA standard)
  - Support Publish/Subscribe interactions
  - Asynchronous method calls, usually without persistent queues

… MOM: Meet the Players

- Object Transaction Monitors (OTMs): Messages are, together with „Transactions“ and „Objects“, central parts of all Object Transaction Monitors
  - COM+: in addition to COM/DCOM and the MS Transaction Server (MTS), also MS Message Queues (MSMQ) is part of the COM+ package
  - EJB: support by Java Message Services (JMS) – Message-driven Beans
    - They are linked with a queue or a topic
    - Message-driven Beans have to have a method onMessage. There, the reaction to incoming messages has to be coded (i.e., they wait for JMS messages which activate the onMessage)
    - Both point-to-point and publish/subscribe interactions between client and server objects possible. The specification which of them is to be used and the specification of a concrete queue or topic, respectively, will be done during deployment
- Additionally: MOM functionality is implemented as separate software packages, e.g., IBM MQSeries (together with workflow management support)

10.2 Queued Transactions



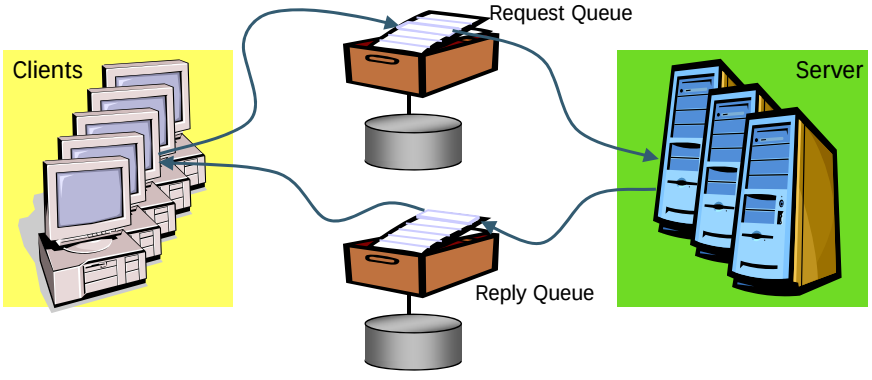
- Goal: messages or requests are persistently stored (persistent Queues) and therefore survive system crashes
- Several clients can concurrently access the queue
  - Enqueue: insertion into queue
- Analogously, several servers can concurrently access the queue
  - Dequeue: take messages out of the queue

Guaranteed Execution ...

- The persistent storage of transactions solely prohibits that messages will go lost in case of crashes of the queue
- What happens when server failures occur while asynchronous requests are being processed?
  - In the synchronous case, this is not a problem since it leads to an abort of the overall transaction
  - In the synchronous case, the enqueue or dequeue, respectively, of requests has to be atomic. This means it has to be implemented by short but nevertheless distributed transactions between client and queue and between server and queue

- Guaranteed execution (exactly once) by implementing asynchronous client/server calls as sequence of three independent transactions (Queued Transactions)
  1. Client generated message, inserts it into the queue, and commits
  2. Server dequeues messages, processes the request, enqueues the reply to the reply queue, and commits
  3. Client dequeues the reply from the reply queue and commits

... Guaranteed Execution



- Transaction 1: Insert into Queue
- Transaction 2: Dequeue Request
- Transaction 3: Dequeue Response

Queued Transactions: Recovery

With queued transactions, client and server failures can be solved

- Server failures
  - Abort of transaction 2 leaves request in the request queue
  - Request can be processed by another server or by the same server after it is recovered

- Client failures
  - Start of the request processing can be checked any time via the contents of the request queue and the reply queue
  - Prerequisite: each request has a unique ID, the request queue maintains the highest ID of accepted requests per client, $ID_{max}(C)$
  - After recovery: client checks, whether the request is in one of both queues (if this is the case, then: wait –in request queue– or get reply –out of reply queue). Otherwise:
    - If ID > $ID_{max}(C)$: re-send request
    - If ID ≤ $ID_{max}(C)$: wait, since request is currently being processed

Further Reading

[Bal 05] R. Baldoni: *The Publish/Subscribe Communication Paradigm and its Application to Mobile Systems*. MINEMA Summer School, Klagenfurt, Austria. July 2005

[BN 09] P. Bernstein, E. Newcomer: *Principles of Transaction Processing*, 2nd edition Morgan Kaufman Publishers, 2009