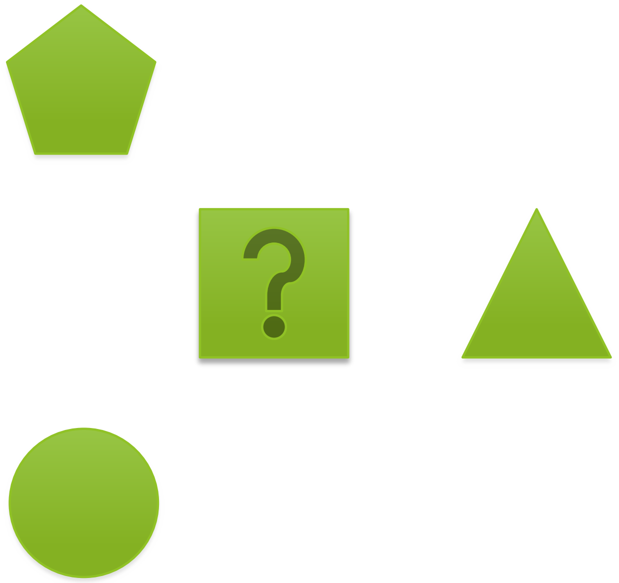
Message Oriented Middlewares

Diogo Gomes dgomes@ua.pt

Enterprise Application Integration (EAI)

- Enterprises are composed of multiple software components and products that most of the times do not comply with standard protocols.
- Enterprise Application Integration is the term used to describe the integration of the computer applications of an enterprise so as to maximize their utility throughout the enterprise.



EAI (2)

- EAI is about connecting system which were designed independently, but which are required to communicate in order to develop a business process.
- Types of EAI
  - Data Integration
    - Database consolidation
  - Process Integration
    - Interaction between applications on a control level
- Types of interaction in an EAI environment
  - synchronous client/server
  - asynchronous client/server

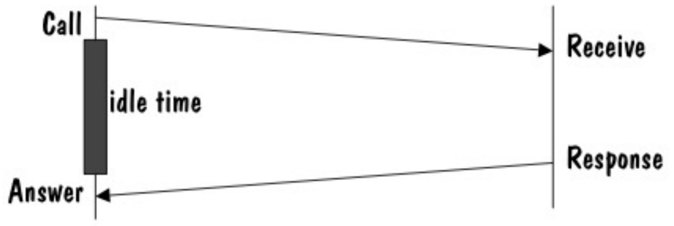
Synchronous Client/Server

- The most straightforward interaction between components is the request/response model in which the client sends a request and waits until the server provides a response:
  - closely resembles the way we program in procedural languages
  - the model is simple and intuitive
  - well supported by RPC
  - needs additional infrastructure when interactions becomes more complex (e.g., nested)

Client/Server disadvantages

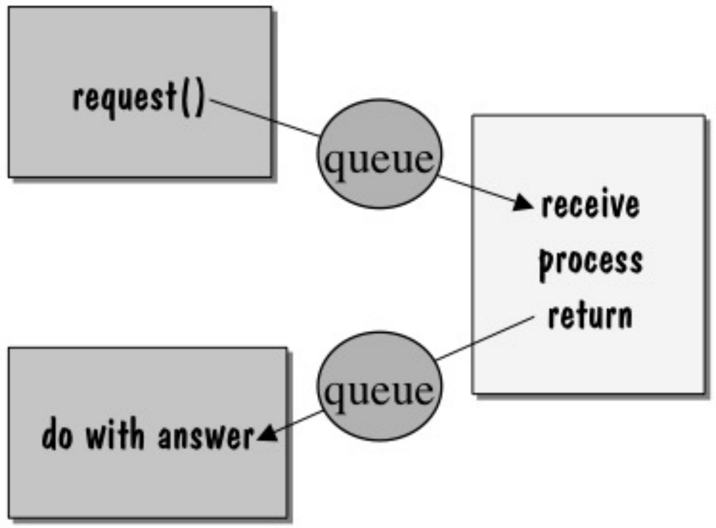
- Synchronous interaction requires both parties to be “on-line”: the caller makes a request, the receiver gets the request, processes the request, sends a response, the caller receives the response.
- Caller is blocked while the callee processes the request (can’t handle new processing batch’s)
- Requires the maintenance of a state on the caller (CPU + RAM resources) – scalability issues
- If systems are not 1 to 1 context must be passed along – communication overhead
- What if a call fails ? Worst case: cascading of calls! Re-issue call ?

- Solutions:
  - have a pool of open connections
  - associate a thread with each connection
  - allocate connections as needed



Reliable Queuing

- A complement to synchronous interactions
  - Suitable in modular designs
    - the code for making a request can be in a different module (even a different machine) than the code for dealing with the response
  - It is easier to design sophisticated distribution modes (multicast, transfers, replication, coalescing messages)
    - helps to handle communication sessions in a more abstract way
  - More natural way to implement complex interactions



Queuing Systems

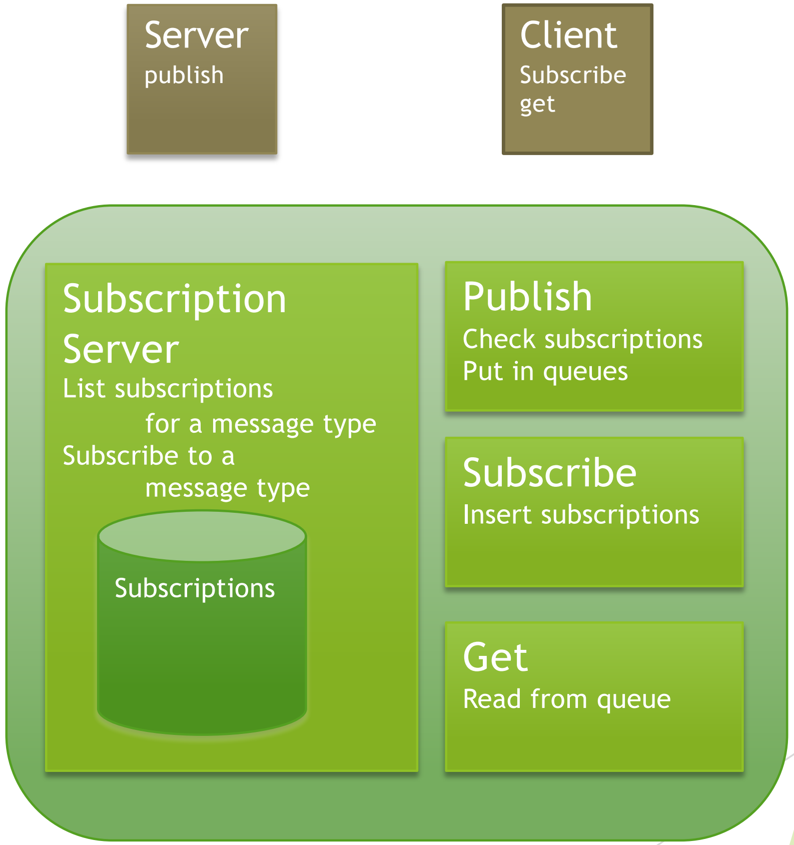
- Queuing systems implement asynchronous interactions.
- Each element in the system communicates with the rest via persistent queues. These queues store messages transactionally, guaranteeing that messages are there even after failures occur.
- Queuing systems offer significant advantages over traditional solutions in terms of fault tolerance and overall system flexibility: applications do not need to be there at the time a request is made!
- Queues provide a way to communicate across heterogeneous networks and systems while still being able to make some assumptions about the behavior of the messages.

Queues Advantages

- Queues allow to implement complex interaction patterns between modules:
  - 1-to-1 interaction with failure resilience
  - 1-to-many (multicast: put in a queue and then send from this queue to many other queues) this is very helpful for “subscriptions”. The fact that the queues are implemented in the database even helps with performance since the logic for distribution can be embedded in the database itself
  - many-to-1 many modules send their request to a single module that can then assign priorities, reorder, compare, etc.
  - many-to-many as in replicated services for large amount of clients
- Queues flexibility at the cost of performance
- Queues promote distribution (e.g. Clustering)

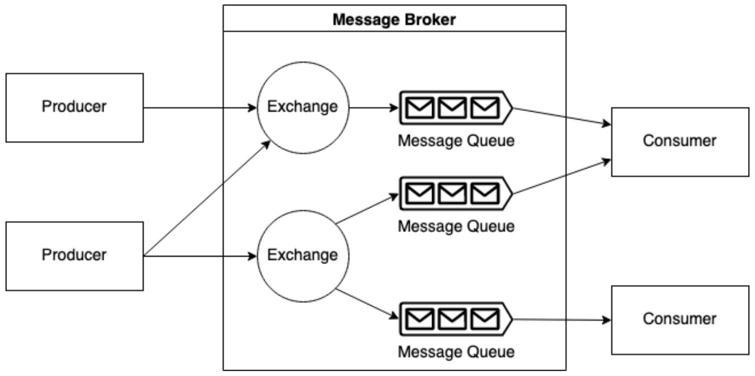
Publish/Subscribe

- Standard client/server architectures and queuing systems assume the client and the server know each other (through an interface or a queue)
- In many situations, it is more useful to implement systems where the interaction is based on announcing given events:
  - a service publishes messages/events of given type
  - clients subscribe to different types of messages/events
  - when a service publishes an event, the system looks at a table of subscriptions and forwards the event to the interested clients; this is usually done in the form of a message put into a queue for that client



Message Brokers

- Message brokers add logic to the queues and at the level of the messaging infrastructure.
- Messaging processing is no longer just moving messages between locations but designers can associate rules and processing steps to be executed when given messages are moved around
- The downside of this approach is that the logic associated with the queues and the messaging middleware might be very difficult to understand since it is distributed and there is no coherent view of the whole



RabbitMQ APACHE ACTIVEMQ

Message Broker

- Publish and Subscribe Messaging
  - In a publish and subscribe message system, producers send messages on a topic. In this model, the producer is known as a publisher and the consumer is known as a subscriber. One or many publishers can publish on the same topic, and a message from one or many publishers can be received by many subscribers. Subscribers subscribe to topics, and all messages published to the topic are received by all subscribers on the topic. This model provides for simple interest driven delivery based on what topics applications subscribe to.
- Point-to-point communication
  - Point-to-point communications in its simplest form has one producer and one consumer. This style of messaging typically uses a queue to store messages sent by the producer until they are received by the consumer. The message producer sends the message to the queue; the message consumer retrieves messages from the queue and sends an acknowledgement that the message was received. More than one producer can send messages to the same queue, and more than one consumer can retrieve messages from the same queue. When multiple consumers are used each consumer typically receives a portion of the message stream to allow for concurrent processing.
- Server-based Message Brokers
  - Server-based message brokers have many advantages, including centralized processing, centralized message distribution, and centralized data persistence. Examples include many proprietary message brokers, Java Message Service (JMS) implementations, and open source products like Apache Kafka and Eclipse Mosquitto.
  - In some cases a decentralized communications model is needed. This peer-to-peer approach typically provides a mechanism to support direct communication between applications for use cases where very low latency is required.

Message brokers vs. APIs

- REST APIs use Hypertext Transfer Protocol (HTTP) to communicate.
  - HTTP is a request/response protocol, it is best used in situations that call for a synchronous request/reply.
  - Services making requests via REST APIs expect an immediate response. If the client receiving the response is down, the sending service will be blocked while it awaits the reply. Failover and error handling logic should be built into both services.
- Message brokers enable asynchronous communications between services so that the sending service need not wait for the receiving service's reply.
  - Improves fault tolerance and resiliency in the systems in which they're employed.
  - Message brokers makes it easier to scale systems since a pub/sub messaging pattern can readily support changing numbers of services.
  - Message brokers keep track of consumers' states.