12: Message-Oriented Middleware

Intro

[...]

Remote Execution Issues

Remote execution is the main service provided by Middleware. There are some issues:

- Synchronous communication
  - The caller code must block and wait until the called code completes execution and returns control to it
  - On the other hand, this does mean that the client and server are synchronised
- Tight coupling of the client and server
  - The server interface explicitly indicates what can be invoked for execution
  - Lack of flexibility
- Reliability
  - Any failure outside of the application (e.g. code, network, hardware) can affect the reliable transport of data between the client and the server
- Scalability (or maybe just Performance)
  - The whole system slows to the maximum speed of its slowest component (client or server)
- Availability
  - All components must be simultaneously available – if any component fails, the whole system fails

The Case for MOM

Distributed components of an application can asynchronously exchange messages in a P2P fashion – components don't all have to be simultaneously available.

The sender and receiver only need to know the message format and the destination to use – loose coupling.

Message loss through network or system failure is prevented using a store and forward mechanism. It can be guaranteed that every message will be delivered to every intended recipient exactly once. There can also be different levels of robustness to provide different levels of service.

Performance characteristics of components are decoupled from each other – subsystems can be scaled independently with little or no disruption to other components.

Concepts

Each component connects to a messaging agent that provides means for creating, sending, receiving, and reading messages.

The provider can implement either point-to-point communication, or publish-subscribe communication.

A consuming client has two methods for receiving messages:

- pull: the consumer polls the provider for messages
- push: the consumer asks the provider to send relevant messages as soon as they are available

Point-to-Point

Each message is addressed to a specific queue. Queues retain all messages until they are consumed or expire.

Features:

- Every message has only one consumer.
- There are no timing dependencies among peers.
- The receiver acks the successful processing of a message.

Queue attributes:

- queue has to have name (unique namespace to prevent confusion with queues from other middleware systems?)
- size

- threshold
- [...]

Publish/Subscribe

Messages are addressed to a topic (analogous to an event type in event notification). Topics retain messages as long as it takes them to distribute them to current subscribers.

Clients subscribe before receiving messages (i.e. don't receive messages that were published before they subscribed), and must continue to be active to consume messages.

Comparing Models

Publisher/Subscriber is normally used in a broadcast (more like multicast) scenario – the publisher has no control over the number of clients who receive message, nor a guarantee any will receive it. Topics can be useful to organise messages even in a one-to-one messaging scenario. P/S is more powerful and flexible, but is complex.

A common application of the point-to-point model is load balancing. With multiple consumers receiving from a queue, the workload for processing messages is distributed among them.

MOM Services 1

- Message Filtering
  - on the attributes of the message
  - on the message payload
- Transactions
  - when transactional messaging is used, either all messages in a transaction happen, or none happen (e.g. if an error happened, all changes are rolled back)
  - messages delivered to the provider in a transaction are not forwarded to the receiving client until the sending client commits the transaction
- Load Balancing
  - consumers can pull messages from a queue at their own pace – only when ready to process it

Benefits

- Easy to add new topics / queues
- Asynchronous execution between components

Example: Oracle JMS

Java message service is a specification for a general Java API that allows distributed applications to use messages.

It also allows Java applications to access other messaging systems, e.g. IBM MQSeries.

Messages can be consumed either synchronously or asynchronously.

Synchronously: The receiver explicitly fetches messages from the destination by calling the `receive()` method, which can block until a message arrives or can time out.

Asynchronously: The receiver registers a message listener. When a message arrives, the JMS provider delivers the messages by calling the listener's `onMessage()` method.

`Session` objects provide a transactional context – either all messages in a transaction are sent, or none of them are.

Advantages

[...]

Robustness

We say the system is robust if:

1. No messages are lost.
2. No messages are received (by the same receiver) more than once.

We can test these properties with a high message load.

Note that if you have a queue or topic, even though we're thinking of it as a single element, it may be split across several computers/cache spaces.

Stage 1: Producer to Service Provider

There are a couple of things we can do to guarantee that messages are received, and are received only once:

1. Use ACKs to indicate that messages have been received and stored in the queue
2. Use transactions to guarantee that either everything is received or nothing is

Stage 2: Service Provider to Consumer

Using ACKs on this side presents a problem:

When a message is received, it goes through several steps (needs to be received, processed, and then results need to be returned). At what point should the ACK be sent – as soon as the message is received, or along with the results?

3. The provider's message storage should be persistent – if errors happen, the messages are still available.

Protocol

1. Message received and acknowledged by the queue/topic
2. Queue/topic is persistent (doesn't lose messages)
3. Message is received, processed, and then acknowledged by the consumer

Transactions

If there are n messages in a transaction, the producer must store all messages until the whole transaction has been completed, as if the transaction fails, it'll have to start again from the beginning.

The client must also store these messages, and not process them and make their changes until the transaction is complete. If they begin making changes and then the transaction fails, they will be in an inconsistent state.

TTL

Using a time-to-live value for messages preserves storage – messages are deleted if they become obsolete. Keeps system load at a manageable level.

Temporary Destinations

Temporary destinations can be used to balance the load by distributing messages. Messages are directed to the temporary destinations for some time before being directed onwards to their regular destinations.

Use-Case for MOM

- persistent p2p means messages don't get lost
- durable pub/sub means the subscriber can disconnect from time to time and still receive messages (when it reconnects)
  - use these for critical parts of the data flow (e.g. it's not a big deal if some statistics go missing)
    * is it also that there's much more data involved in the statistics, so persistence and durability are a lot more expensive?