Lecture 14

Robust messaging service

Analysis of messaging systems

- Messaging systems need to be tested under heavy load for a period of time (e.g., one week). It may happen that messages are lost during a short time interval. Why ?
  - Log files: systems detected an upgrade patch, did the upgrade and then automatically restarted.
- *Scenario:* normally, when processing a message over HTTP, a service would open a transaction against the database, attempt to write the data, and commit the transaction. Under regular circumstances this would succeed. If the message-processing server restarted mid-transaction, the database would detect the timeout of the transaction and rolls back its changes, maintaining its consistent state. However, when the server would start up again, the data of the original message would be available neither in memory nor in the network processing stack of the server—that message would be lost!

Reliability

- A reliable messaging service means:
  - no message is lost;
  - no duplicate arrives at destination.

- One of the best solutions is to send *persistent* messages within transactions, then receive them within a *transaction* from a non-temporary queue or from a durable subscription:
  - persistent: messages are not lost if the provider fails;
  - a transaction is a set of operations grouped together that are either all committed or all fail.

- It'll have an impact on performance, therefore... a lower level of reliability can reduce overhead and improve performance.

Basic reliability mechanisms

1. Using *message acknowledgment*.

2. Implement *message persistence*.

3. Set *message priority levels:* make messages order to depend on their priority- urgent messages are delivered first.

4. Allow *message to expire:* messages will not be delivered if they become obsolete – set Time-To-Live.

5. Create *temporary destinations:* last only for the duration of the connection in which they are created. They can be used to implement a simple request/reply mechanism.

1. Message acknowledgement

- A message is successful if it is acknowledged. The consumption of a message takes three stages:
    1. The client receives the message;
    2. The client processes the message;
    3. The message is ack. This operation is either carried out by the provider, or by the client depending on the session ack mode.
- In transacted sessions, ack happens automatically when the transaction is committed. If a transaction is rolled back, all consumed messages should be redelivered.
- In non-transacted JMS sessions, the ack depends on the value of the 2nd arg of the createQeueSession or createTopicSession method:
    - Session.AUTO_ACKNOWLEDGE;
    - Session.Client_ACKNOWLEDGE;
    - SessionDUPS_OK_ACKNOWLEDGE.
- If messages have been received but not ack when the QueueSession or TopicSesion (durable subscriber) terminates, the provider retains them and redelivers them when the consumer next accesses the queue or the topic.

2. Message persistence

- There are two modes:
  - The persistent delivery mode (by default at JMS) instructs the provider to ensure that the message is not lost if the provider fails (stable storage). This is an expensive solution that can be implemented for specific applications.
  - The non-persistent delivery mode. If the provider fails, messages on-transit are lost.

System consistency

- Since the transaction that includes processing a message may be rolled back as a result of a database deadlock, other information that will be untrue once the transaction rolls back can be sent.

- Consider the following *scenario: deadlocks* can occur in databases where multiple threads work with data from the same tables. One thread successfully locks table 1 and attempts to lock table 2. At the same time, a different thread successfully locks table 2 and attempts to lock table 1. *The database detects this deadlock, selects one thread as a victim, and aborts its transaction.*

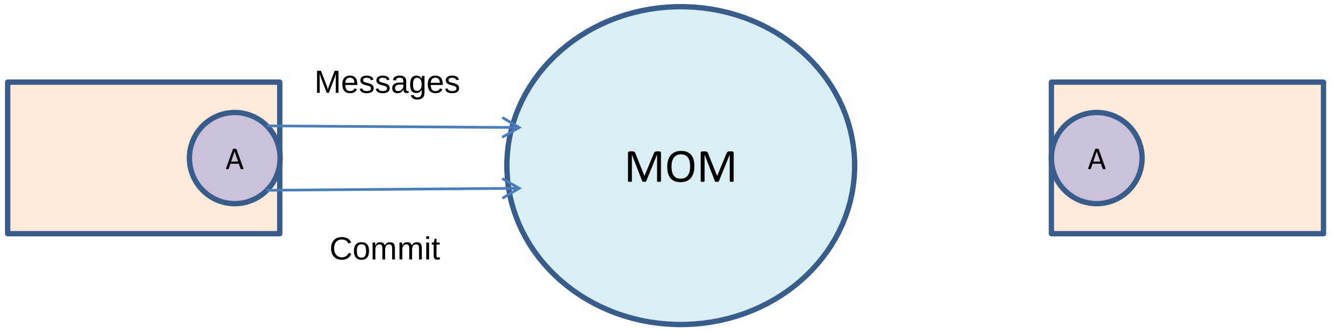
- The problem here is that one system was notified about a change to data in one table that has been rolled back. This may, in turn, *lead that system changing its internal data and further propagating misinformation, ultimately leading to global inconsistency.*

Solution: transactional messaging

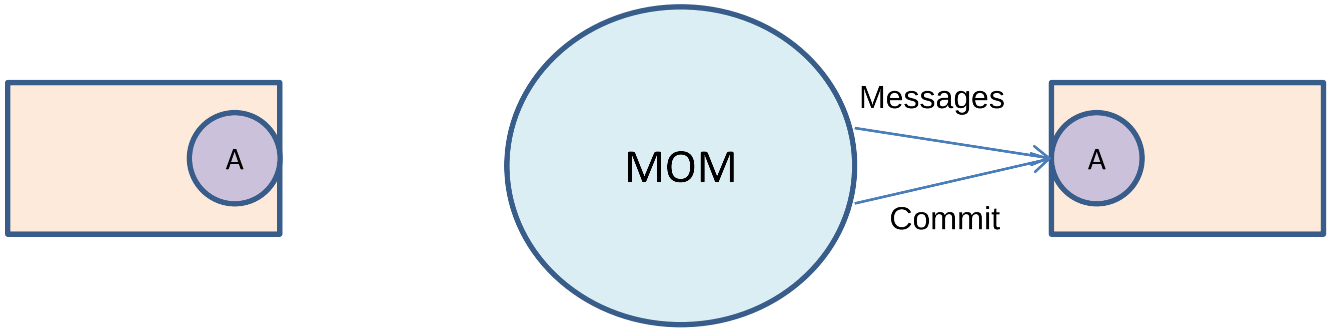
- When sending a message to a queue, the message is not sent when the method call returns. This is very different from the behaviour exhibited by HTTP and other connected technologies. Before a message is sent, it is stored locally on the same machine in *an outgoing queue*. When the message is sent within the context of a transaction, only after the transaction commits is the message released so that the system can actually consider it sent.

- Note that transactional messaging does not imply that both the sender of a message and the receiver processing that message share a transaction. The fact that a message arrives at a receiver implies that a separate transaction at the sender committed successfully. However, the fact that that transaction was committed says nothing about what may happen when the receiving service handles that message.

MOM transaction model



Sender – MOM transaction



MOM – Receiver transaction

Once a consumer has received the message, a consumption report is generated and sent to the message sender to confirm the consumption of the message.

Message processing exception

- At times, a message may arrive at a client where the processing of that message results in *an exception. There are many possible causes for* this exception.

- It's important to understand the differences between them and how to handle them. Sometimes an exception may occur only once or twice in the processing of a message; other times it appears that no matter how many times the client will try to process that message, it will always fail.

Loss of messages can be good

- Let's consider an over Internet business-to-business scenario for e-commerce: the ordering system is communicating with an external shipping company to take care of the shipment of orders to customers. For every accepted order, a message needs to be sent to the shipping system. In order to prevent messages from getting lost if either system goes down, durable messaging is used along with transactional handling of messages to transfer the order information.
- If we take an average message size of 1MB for each order sent, and the order system is processing 10 orders per second, and the shipping servers become unavailable, these messages will be durably stored in an outgoing queue. In terms of I/O usage, that's 10MB per second, or 600MB/min !

- Servers tend to lose stability when they aren't able to write to their hard drives anymore — not only that, it's almost impossible to bring them back up.

- Conclusion: it's better to throw away messages that had not yet reached their destination successfully.

Advanced reliability mechanisms

- Creating durable subscriptions: the subscriber receives messages published when it was not active.
- Using local transactions

Performance metrics

- System throughput: total number of messages sent/received per second across the provider.
- Success rate: total number of messages received/total number of messages sent.
- Scalability can be defined as the ability of the system to manage:
  - an increased number of destinations while keeping the traffic per location constant (horizontal topology), or
  - an increased number of messages through a fixed number of destinations (vertical topology)

- SPECjms2007 benchmark can be used to evaluate the above metrics.

References

- Udi Dahan: "Build Scalable Systems That Handle Failure Without Losing Data"
- http://msdn.microsoft.com/en-us/magazine/ cc663023.aspx
- Performance Evaluation of Message-oriented Middleware using the SPECjms2007 Benchmark

  https://www.cl.cam.ac.uk/research/srg/opera/publications/papers/SamKai08-PerfEvalJ-SPECjms2007.pdf