Lecture 11

MOM

Chapter 15

Message Oriented Middleware

Just what is Message Oriented Middleware, Anyway?

In Message Oriented Middleware (MOM):

- one or more Producers creates messages and sends them to a queue
- then one or more Consumers retrieve messages from the queue
- The queue, however, can actually be a message broker, that can do value added analyses on the messages, for example, can implement quality of service techniques based on priorities
- The use of this intermediary message broker (queue) allows the Producers and Consumers to work asynchronously with respect to each other.
- Also, the use of this intermediary message broker allows loose coupling, which can help when it is necessary to have heterogeneous systems communicate

Just what is Message Oriented Middleware, Anyway?

Curry (2004) said:

> "A client of a MOM system can send messages to, and receive messages from, other clients of the messaging system. Each client connects to one or more servers that act as an intermediary in the sending and receiving of messages. MOM uses a model with a peer-to-peer relationship between individual clients; in this model, each peer can send and receive messages to and from other client peers."

Curry (2004) further states that the message queue is a fundamental concept associated with Message Oriented Middleware

Point to Point vs. Publish/Subscribe

Liu and Gorton (2005) said:

"MOM typically supports two forms of messaging: point-to-point and publish/subscribe (Pub/Sub). In the PTP model, the message producer posts a *message* to a *queue* and the message consumer retrieves the message from the queue. In the Pub/Sub model, a message producer publishes a message to a *topic*, and all consumers subscribing to the same topic retrieve a copy of the message. MOMs also define a set of reliability attributes for messaging, including non-persistent or persistent and non-transactional or transaction queues."

Point to Point vs. Publish/Subscribe

Liu and Gorton (2005) said:

“MOM typically supports two forms of messaging: point-to-point and publish/subscribe (Pub/Sub). In the PTP model, the message producer posts a *message* to a *queue* and the message consumer retrieves the message from the queue. In the Pub/Sub model, a message producer publishes a message to a *topic*, and all consumers subscribing to the same topic retrieve a copy of the message. MOMs also define a set of reliability attributes for messaging, including non-persistent or persistent and non-transactional or transaction queues.”

Point to Point vs. Publish/Subscribe

Gomaa (2011) defines two communication patterns for group messaging:

- broadcast message communication pattern
  - with the broadcast message pattern, an unsolicited message is sent to all recipients.
- subscription/notification message communication pattern
  - The subscription/notification message pattern uses a selective form of group communication where the same message is sent to members of a group
  - A component can subscribe and unsubscribe from a group, and can belong to more than one group

Point to Point vs. Publish/Subscribe

In the point-to-point messaging paradigm, one or more producers sends a message through a message broker to a single consumer

- (in the simplest case, this would be a queue)
- The consumer receives this message only once

Point to Point vs. Publish/Subscribe

In the publish/subscribe message paradigm, messages can be distributed

- from one producer to many consumers
- or from many producers to many consumers
  - all consumers that subscribe to a particular topic (channel) will receive the message

When would you want to use Message Oriented Middleware?

Curry (2004) said:

“If the distributed systems will be geographically dispersed deployments with poor network connectivity and stringent demands in reliability, flexibility, and scalability, then MOM is the ideal solution.”

When would you want to use Message Oriented Middleware?

Message Oriented Middleware may also be used when a service it provides is useful.

Curry (2004) lists some common MOM services:

- message filtering—messages that are of interest are selected
  - some possible filters include:
    - channel based
      - events are categorized into predefined groups
      - consumers subscribe to groups of interest
    - subject-based
      - messages contain a tag field that says what their subject is
      - subscribers receive messages with selected subjects
    - content-based
      - querying languages are used to examine the data in the message
    - content-based with patterns (composite events)
      - querying languages are used to examine data in multiple messages

When would you want to use Message Oriented Middleware?

- transactions—tasks are grouped into a single unit of work
  - transactional messaging
    - used when several messages are sent, but the tasks from all messages must all succeed or else all must fail.
      - If any one message fails, then the results of all the other messages must be rolled back.
  - transaction roles—the roles of message producer, message consumer, and message broker are defined as follows:
    - message producer interacting with message broker:
      - send a message or a set of message to the message broker
      - the broker stores the messages and then sends them when it receives a commit from the producer
      - the broker disposes of all messages when it receives a Rollback from the producer
    - message consumer interacting with message broker:
      - the consumer receives a message or set of messages from the message broker
      - the broker disposes of all messages when it receives a commit from the consumer (this is because the messages have been correctly handled so they're no longer needed)
      - the broker re-sends all messages to the consumer when it receives a Rollback from the consumer (this is because the remaining messages that have taken effect must be known so they can be reversed)

When would you want to use Message Oriented Middleware?

- reliable message delivery. Message delivery can be configured to take place in different ways:
  - at most one time
  - at least one time
  - only one time
  - number of retries on delivery failure
- guaranteed message delivery
  - consumer must acknowledge message receipt
  - messages than have not been acknowledged are retransmitted after a time period (how this is done is settable according to reliable message delivery)
    - push—the message broker uses some algorithm to balance the load over multiple servers
    - pull—when a server has need of input, it pulls a message from the queue (the argument here is that the server itself knows best when it is not busy and needs more to do)

When would you want to use Message Oriented Middleware?

- message formats
  - one message format may be converted into another message format
  - possible plug ins to do format changes
- load balancing
  - the workload of the system is spread over several different servers. There are two main methods for balancing the load:
    - push—the message broker uses some algorithm to balance the load over multiple servers
    - pull—when a server has need of input, it pulls a message from the queue (the argument here is that the server itself knows best when it is not busy and needs more to do)
- clustering—replicating the state of a server across several other servers to help with fault tolerance.
  - In this way, if a server fails, the client can be sent to a different server in a seamless manner.

What Middleware Technologies Can Be Described as Message Oriented Middleware?

Curry (2004) said:

“To date, a number of MOM standardization have emerged such as the CORBA Event Service [25], CORBA Notification Service [26] and most notably the Java Message Service (JMS).”

Liu and Gorton (2005) said:

“Messaging services are implemented by message-oriented middleware (MOM) such as Microsoft MSMQ, IBM WebSphere MQ, CORBA Notification Services and Sun’s JMS (Java Messaging Service).”

What Middleware Technologies Can Be Described as Message Oriented Middleware?

Tarkoma (2012) said:

“Message oriented Middleware and Event Notification are becoming more popular in the industry with the advent of the CORBA Notification Service and DSS[sic], the Java Messaging Service, and other related specifications and products from many vendors.”

What Middleware Technologies Can Be Described as Message Oriented Middleware?

Another well-known Message Oriented Middleware is the Advanced Message Queueing Protocol (AMQP) from OASIS and the International Standards Organization

The Enterprise Service Bus can also be considered a Message Oriented Middleware

- although its features go beyond what is normally expected from an MOM
- Also, the Enterprise Service Bus could use some kind of web services instead of a message oriented middleware, so the kind of implementation might affect whether it truly meets the definition of MOM

CORBA Event Service and CORBA Notification Service

McHale (2007) says

- the CORBA Event Service provides a basic form of publish-subscribe
- whereas the CORBA Notification service greatly extends the Event Service to provide a richer version of publish-subscribe

McHale notes that the CORBA terminology is different from normal MOM technology.

- the term “supplier” is used instead of “publisher”
- the term “consumer” is used instead of “subscriber”
- the term “topic” becomes “event channel”

CORBA Event Service and CORBA Notification Service

The CORBA Event Service provides a PushConsumer IDL that (in part) is defined as follows:

```
interface PushConsumer {
   void push(in any data);
   void disconnect_push_consumer();
 };
```

Then you provide servant code on the Consumer side that would handle this IDL interface

- (the IDL skeleton code)

CORBA Event Service and CORBA Notification Service

When the Event service receives a message from the supplier that is for you, the Event Channel that has been defined on the Event Service would call “Push” and your CORBA code would handle it:

- Supplier→ Push message to Event Service, which puts message in a queue
- When message moves at front of queue, Event service → Push message to CORBA Server

Note that there can be several suppliers and several consumers.

CORBA Event Service and CORBA Notification Service

The CORBA Event Service provides a PullSupplier IDL that (in part) is defined as follows:

```
interface PullSupplier {
    any pull();
    any try_pull(out boolean has_event);
    void disconnect_pull_supplier();
  };
```

The difference between pull and try_pull is

- pull is blocking
- try_pull is non-blocking.
  - If there is an event when try_pull is called, then the boolean has_event is set true

CORBA Event Service and CORBA Notification Service

This works as follows:

- Supplier→ Push message to Event Channel
- Consumer→ Pull from Event Channel

What we have just seen are the Canonical Push Model and the Hybrid Push/Pull model of Event Service operation

CORBA Event Service and CORBA Notification Service

There are actually four models of operation:

- The Canonical Push Model:
  - Active Suppliers call Push on the Event Channel to push a message to the Event Service
  - Passive Consumers receive a push call from the Event Channel
- The Canonical Pull Model:
  - Active consumers call Pull on the Event Channel to ask the Event Service for a message
  - The Event Channel then calls Pull on the Passive Supplier

CORBA Event Service and CORBA Notification Service

- The Hybrid Push/Pull Model:
  - An active supplier calls Push on the Event Channel, which then queues the message
  - The active consumer calls Pull on the Event Channel to receive the message

- The Hybrid Pull/Push Model:
  - The Event Channel calls Pull on a Passive Supplier
  - The Event Channel calls Push on a Passive Consumer
  - (This model requires the Event Service to be somewhat intelligent)

CORBA Event Service and CORBA Notification Service

The CORBA Notification Service is a super set of the Event Service. It provides Quality of Service as well as other additional functionality.

Some of the functionality provided:

- Filter objects can be attached to event channels
    - These filter objects provide constraints as to the messages that can be passed
        - This is intended to:
            - reduce work at the server, throwing away inappropriate messages
            - reduce network traffic due to inappropriate messages
    - A constraint language is provided to write these constraints.
        - This constraint language is an extension of the constraint language used by the old CORBA Trading Service
-

CORBA Event Service and CORBA Notification Service

- Additional control over when to deliver messages is provided. This includes (among many other items):
  - Expiration times for event messages
  - Maximum number of events accepted per consumer
  - Priority of events
- Structured Events
  - Has headers that include fixed and variable portions
  - Can specify specific event types
  - Can specify event domains (telecommunications, health care, etc.)

Java Message Service and Message Beans

The Java Message Service provides queue and topic based messaging. It provides two messaging models:

- Point to point
- Publish/subscribe

With point to point, a queue is used, with publish/subscribe a topic is used.

Java Message Service and Message Beans

The JMS components are:

- JMS client—produces or consumes messages
- Non-JMS clients—use a messaging system’s native API instead of JMS
- Messages—defined by an application, used to send information between clients
- Service provider—implements the JMS interfaces. Examples of providers include (among many others):
  - OpenJMS
  - Apache ActiveMQ
-

Java Message Service and Message Beans

- Administered objects—clients connect to these using JNDI, use them to establish connections:
  - Destination— the object a client uses to specify the target of messages produced or consumer. When using point to point, destinations are queues, in publish/subscribe, destinations are topics.
  - Connection factories—used to establish a Connection between a JMS client and a service provider
-

Java Message Service and Message Beans

- After a Connection is established, a Session is then created.
  - A Session is single threaded
  - A message producer associated with a session sends messages to a destination
  - a message consumer associated with a session receives messages from a destination.
  - a message delector associated with a session is used to filter messages

Java Message Service and Message Beans

A JMS message consists of (only the header is required):

- Header—used for routing
- Additional properties
- Body, message types include:
    - Text (string data)
    - Byte (byte data)
    - Map (key/value pairs)
    - Stream (stream of primitive values)
    - Serializable Object

Java Message Service and Message Beans

There are two ways a Consumer can handle messages with JMS:

- Synchronously
  - subscriber calls receive method, to fetch the message from the associated destination.
  - receive blocks until message arrives or timeout
- Asynchronously
  - a message listener is registered with a consumer.
  - when a message arrives, the JMS provider calls “onMessage” on the listener

Java Message Service and Message Beans

JMS provides some quality of service mechanisms, these include (among others):

- Message expiration time
- Priority
- Persistence

Message Beans Accessed by Java Message Service

There are two major kinds of Enterprise Java Beans (EJBs) normally discussed:

- Session Beans
- Message Beans

- Both qualify as EJBs because they both are sever-side components that run in an EJB container in an application server
- However, Message Beans operate as a Message Oriented Middleware.

Message Beans Accessed by Java Message Service

- A Message Bean is typically called using the Java Messaging Service (JMS)
- Message Beans can support any messaging protocol (queue-based)
  - by default they support the Java Message Service (JMS)

Message Beans Accessed by Java Message Service

- Unlike a Session Bean, a Message Bean does not have an interface that a client looks up in order to call a method on the interface
  - rather, a Message Bean is not coupled directly to a client

- A Session Bean would normally have an interface class and a bean class, whereas a Message Bean only has a Bean class

Message Beans Accessed by Java Message Service

- In the case of a Message Bean, the client sends messages to a queue.
  - the Message Bean listens for messages on the queue
- A Message Bean is similar in some ways to a Stateless Session Bean.
- A Message Bean is stateless.
- The container can call any Message Bean instance to handle a message
  - so there can be pools of Message Beans in a container
- A Message Bean might more accurately be called a “message driven” bean

Message Beans Accessed by Java Message Service

- Message beans are annotated with "@MessageDriven"
- Message beans receive messages from:

javax.jms.MessageListener, which has one method:

void onMessage(Message inMessage)

Outline of Java Message Bean

```
@MessageDriven(
            mappedName = "theQueue",    activationConfig = {
                        @ActivationConfigProperty( propertyName = "destinationType",
                        propertyValue = "javax.jms.Queue")
                            }
)
public class theMessageBean implements MessageListener {


  public void onMessage(Message message) {

            … put very important stuff here…
  }
}
```

Message Beans Accessed by Java Message Service

- At the beginning, the term “mappedName” is the JNDI name of the location from which the message bean will consume messages
- The destinationType says what kind of destination we’re working with: the choices are javax.jms.Queue or javax.jms.Topic
  - here we’re using a queue so we select that one
  - note that for this code to work, some code somewhere must have previously created a Queue
    - annotated as @Resource
    - with a mappedName of “theQueue”

Message Beans Accessed by Java Message Service

- when the queue receives a message,
  - the EJB container invokes the onMessage method of the MessageListener interface
    - which then calls onMessage
      - which then does the appropriate work

- Note that with Java Messaging Service, the API is specified, but the format of messages is not
  - so every JMS broker can implement messages in a different format

OMG Data Distribution Service (DDS)

- The OMG Data Distribution Service is intended to provide real-time, scalable, dependable publish/subscribe.

- Corsaro (2010) compares DDS to Java Message Service and AMQP:

  "One way of comparing DDS to JMS and AMQP is to measure what they standardize. JMS standardizes an API while AMQP standardizes a wire-protocol -- DDS standardizes both." DDS is fully distributed and does not require the presence of any broker mediating communication between producers and consumers. On the other hand, both JMS and AMQP have a broker-based architecture where one or more brokers mediate the distribution of information from sources to destinations."

OMG Data Distribution Service (DDS)

- A wire protocol is defined by PCMagazine Encyclopedia (2016) as follows:

  “In a network, a wire protocol is the mechanism for transmitting data from point a to point b. The term is a bit confusing, because it sounds like layer 1 of the network, which physically places the bits "onto the wire." In some cases, it may refer to layer 1; however, it generally refers to higher layers, including Ethernet and ATM (layer 2) and even higher layer distributed object protocols such as SOAP, CORBA or RMI. See OSI model, communications protocol, data link protocol and distributed objects.”

OMG Data Distribution Service (DDS)

- Corsaro (2010) also says DDS:
  - provides more extensive Quality of Service control than either JMS or AMQP
  - provides more extensive querying of messages while providing low latencies.
- DDS is language, operating system, and hardware independent.
- In DDS, a topic is a unit of information that a Publisher and a Subscriber exchange.
  - A topic has a name, a type, and a quality of service setting.

OMG Data Distribution Service (DDS)

A DDS topic type is described in a struct in IDL, and may contain primitive types, sequences, arrays, structs, etc.

It is similar to a CORBA IDL interface, except it focuses on data and not method calls:

```
struct myDataType {
        string thename;
        float thevalue;
        long anothervalue;
       short yetanothervalue;
};
```

OMG Data Distribution Service (DDS)

- A topic type is associated with a list of keys.
  - A key identifies a particular piece of data within a topic (an instance of a topic type)
  - Topics are registered with a DDS implementation.
  - Then data is written to a topic using a data writer and read from a topic using a data reader.
  - A listener on a topic uses a data reader to read the data from the topic.

OMG Data Distribution Service (DDS)

- When a data reader is created, it can be associated with a Content Filtered Topic.
  - A Content Filtered Topic is
    - associated with a particular topic
    - filters the data in that topic.
    - Thus, a Data Reader that is associated with a Content Filtered Topic will only be able to read data that the filter allows through

- A pre-existing Data Reader can be called with a Query Condition (created specifically for that particular Data Reader) as an argument
- Then the Data Reader will only be able to read data that matches the Query Condition

OMG Data Distribution Service (DDS)

- Various quality of service properties may be specified: reliability, availability, priority, etc.
  - When a publisher is matched to a subscriber, the quality of service provided by the publisher must match the quality of service needed by the subscriber
  - Specific kinds of quality of service are specified in DDS, this includes (among others):
    - Reliability
      - All data guaranteed delivered, or
      - Best effort
    - Deadline
      - Maximum inter-arrival time between data items may be specified
    - Latency Budget
      - Maximum delay from when data is written until data given to the receiver

Advanced Message Queueing Protocol (AMQP)

Advanced Message Queueing Protocol (AMQP) is

- an ISO/IEC Standard (ISO/IEC 19464)
- an OASIS standard

Advanced Message Queueing Protocol (AMQP)

According to an OASIS press release at the time AMQP became a standard, see Geyer (2012), AMQP can be described as:

> “AMQP is a wire-level messaging protocol that offers organizations an efficient, reliable approach to passing real-time data and business transactions with confidence. AMQP provides a platform-agnostic method for ensuring information is safely transported between applications, among organizations, within mobile infrastructures, and across the Cloud.”

Advanced Message Queueing Protocol (AMQP)

Geyer (2012) says further:

> “AMQP supports common interaction patterns: one way, request/response, publish/subscribe, transactions, and store-and-forward. It does this with flow-control, multiplexing, security, recovery and a portable data representation that enables message filtering. AMQP is capable of being used in both point-to-point and hub-and-spoke (broker-based) topologies.”

Advanced Message Queueing Protocol (AMQP)

- AMQP is used for messaging between individual components of OpenStack
- The Microsoft Azure Service Bus cloud service also supports AMQP. According to Microsoft Azure:
  - “The Azure Service Bus is a generic, cloud-based messaging system for connecting just about anything—applications, services, and devices—wherever they are.”

Advanced Message Queueing Protocol (AMQP)

- AMQP is designed to work on any language or operating system
- Before publishing can take place, an exchange, a queue, and bindings must first be defined

Advanced Message Queueing Protocol (AMQP)

- AMQP has exchanges, routes, queues, and bindings.
  - An Exchange receives messages and then routes them to appropriate queues.
  - Bindings are rules for how messages are sent from exchanges to queues.
- A producer publishes a message to an exchange
  - Routing rules then determine which queues will receive a message
  - Then one or more consumers that subscribe to that queue will receive a message.
  - Most commonly one subscriber (consumer) is associated with a queue.
  - If there is more than one subscriber then the messages are sent round robin to the consumers

Advanced Message Queueing Protocol (AMQP)

The different kinds of exchanges are as follows (a routing key is a string that contains routing information):

- Fanout exchange
  - any message sent to the exchange is sent to all queues bound to that exchange, no routing keys involved
- Direct exchange
  - a queue is bound to the exchange to request messages that exactly match a routing key
- Topic exchange
  - a queue is bound to the exchange to request messages that match a routing key pattern
- Headers exchange
  - uses additional headers for routing rather than using routing key strings
  - this enables different types of data (other than strings) to be used for routing.
  - However, otherwise the operation is similar to direct exchange

Advanced Message Queueing Protocol (AMQP)

Spring (2016) provided a comparison of AMQP and Java Messaging Service

- AMQP has only queues whereas JMS supports queues and topics
- With AMQP, a message from a producer is published to an exchange, then various bindings (routing rules) select which queues receive the message
  - however, only one consumer receives messages from an AMQP queue
- With JMS, multiple consumers may consume a message on a topic
  - but only one consumer can receive a message on a queue
- Thus although AMQP and JMS work quite differently, they are very similar in actual results for sending messages from producers to consumers

Advanced Message Queueing Protocol (AMQP)

- JMS and AMQP both have message headers that allow message routing
- with bothJMS and AMQP, brokers are responsible for message routing