Middleware and Distributed Systems

Message-Oriented Middleware

Martin v. Löwis
Message-Oriented Middleware

- Middleware for communication of messages between clients
- Focus on non-blocking communication style
  - Producer gives message to middleware
  - Consumer gets message from middleware
- Application responsibility for message structure, transparent to middleware
- Reliability through store-and-forward principle
- Loose coupling - decoupling of communication in space, in time, syntactically and semantically
Push / Pull Model

Diagram showing the push and pull model of message-oriented middleware with arrows indicating direction of data flow between consumers and producers.
Message Brokers

- MOM provider can transform / alter the format of the message content
- ‘Hub and spoke’ architecture
  - Completely connected graph of nodes - $n/2 \times (n-1)$ edges (even more with directed graph)
  - Decoupling of sender and receiver through hub (think of FedEx)
- Protocol translation and data translation
  - Common meta-format, otherwise same scalability problem for translators
Messing Models

- **Point-to-point**
  - Producing client sends message to consuming client through queue
  - Possibility of multiple producers
  - Each message delivered to only once to one of the consumers

- **Publish / Subscribe**
  - For one-to-many or many-to-many distribution of messages
  - Subscription to topic / channel, no restriction on client role
  - All subscribed clients receive the message, if available
Persistent Message Queues

- Message queue
  - Persists message until delivery
  - Referred to using a logical name, managed by queue manager
  - Many attributes: message length, max depth, persistence, ...
- Concept of persistent queues originates from TP monitors [Bernstein90]
  - Reliable management of transaction requests
  - Recoverable queue (‘stable memory of elements’) for client / server
  - Queue repository, queue manager
Queue Formats

- FIFO queues
- FIFO queues with priorities
- Public queue: Open access for all clients
- Private queue: Requires client authentication
- Temporary queue: Created for finite period of time or duration of a condition
- Journal queue: System keeps a copy of every message
- Connector / bridge queue: Acts as proxy between proprietary MOM systems
- Dead-Letter / Dead-Message queue: Final sink for undeliverable or expired messages
Message Delivery

• Configuration of message delivery options per queue
  • at-most-once, at-least-once, once-and-once-only delivery
  • Time-to-live (TTL) for messages
• Guaranteed message delivery
  • MOM saves pending messages on persistent storage
  • Consumer must confirm message receiving
• Certified message delivery
  • Consumption report sent to the producer
IBM MQSeries Example
Message Filtering

- Consumer / receiver demands selection of messages
- Typical filtering on properties of the message, sometimes on payload
- Boolean expression, SQL WHERE clause syntax
- Different filter types
  - Topic / Channel-based
  - Subject-based: Message subject as general property, string matching
  - Content-based: Parsing and check of payload information ("stock_symbol=SUN")
  - Pattern-based: Inspection of content over different messages ("stock_symbol=SUN AND ms_price < 50")
Transactional Messaging

- Group tasks into a single unit of work
  - All message delivery is completed, or all will fail together
  - Messages are not forwarded to the consumer until transaction commit by the sending producer
  - Messages are not removed from the MOM until transaction commit by the receiving consumer
  - Typically represented by transactional queue
- Local transactions within a single resource manager
- Global transactions with multiple (heterogeneous) distributed resource managers, and external transaction coordinator
- Queued Transaction Processing (QTP): Queues as transactional resources
Commercial Products

- TIBCO Rendezvous
- IBM Websphere MQ (MQSeries)
- Sun ONE Messaging Server
- Microsoft Message Queue Server (MSMQ)
- Sonic MQ -> ESB

Common focus on:
- Guaranteed message delivery features
- Scalability regarding message throughput
- Few standardization attempts (AMQP)
CORBA Event Service

- Alternative to RPC-based client-server communication
  - Extension with CORBA notification service
- Event: “something that happens”, event consumer and event supplier
- Notification: Message that informs about some kind of event
- Push (active producer) and pull (active consumer) model
- Single source, (possibly) multiple recipients
  - Source does not know all consumers
  - Non-blocking sending of message
  - Typed communication (untyped communication through any type)
Stock Exchange Example

interface StockExchange;
struct StockQuote
{
    string stock_id;
    StockExchange market_place;
    double current_quote;
    Time current_time;
};
interface Subscriber
{
    void receive (in ::StockQuote current_quote);
};
interface StockExchange
{
    void subscribe (in ::Subscriber customer);
};
CORBA Event Service Interfaces

```java
interface PushConsumer {
    void push (in any data) raises(Disconnected);
    void disconnect_push_consumer();
};
interface PushSupplier {
    void disconnect_push_supplier();
};

interface PullSupplier {
    any pull () raises(Disconnected);
    any try_pull (out boolean has_event)
        raises(Disconnected);
    void disconnect_pull_supplier();
};
interface PullConsumer {
    void disconnect_pull_consumer();
};
```
Stock Exchange Example - 2nd try

interface StockExchange2;
struct StockQuote
{
    ...
    StockExchange2 market_place;
    ...
};
interface Subscriber2 : ::CosEventComm::PushConsumer
{};

interface StockExchange2 : ::CosEventComm::PushSupplier
{
    void subscribe (in ::Subscriber2 customer);
};
Event Channel as Middleman

Consumer -> Supplier
- push
- pull

Consumer -> Channel
- push
- pull

Channel -> Supplier
- push
- pull
interface Channel
   : ::CosEventComm::PushSupplier,
     ::CosEventComm::PushConsumer {
   void register_supplier (
       in ::CosEventComm::PushSupplier supplier);
   void register_consumer ( 
       in ::CosEventComm::PushConsumer consumer);
};

interface MyConsumer : ::CosEventComm::PushConsumer {}

interface MySupplier : ::CosEventComm::PushSupplier {}
CORBA Event Filters

- Subscriber might be only interested in specific notifications
- Demands interface for filter installation
CORBA Notification Service

• Wide usage of Event service, several proprietary extensions
• Extended event service in CORBA 3 -> Notification Service
• Filter objects
  • Interfaces to add, remove and modify constraints on message values
• Quality of service
  • Interfaces for control over notification delivery characteristics
  • Examples: discard policy, earliest delivery time, expiration time, max events per consumer, ordering policy, event priority, re-connection
• Demands general standardized event structure
Structured Notification Event

- Header with fixed and variable part
  - domain_name: vertical industry domain (telco, finance, health care)
  - type_name: type of event within domain (StockQuote, VitalSigns)
  - event_name

- Optional header fields, some standardized names
  - EventReliability / ConnectionReliability (short): 0 - best effort, 1 - persistent
  - Priority (short): -32.767 ... 0 ... 32.767
  - StartTime, StopTime or Timeout for delivery

- Body with filterable data and raw data
CORBA Notification QoS Levels

<table>
<thead>
<tr>
<th>Property</th>
<th>Per-Message</th>
<th>Per-Proxy</th>
<th>Per-Admin</th>
<th>Per-Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventReliability</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ConnectionReliability</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Priority</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>StartTime</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StopTime</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>StartTimeSupported</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StopTimeSupported</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxEventsPerConsumer¹</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OrderPolicy</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>DiscardPolicy¹</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaximumBatchSize²</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PacingInterval²</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

AnyOrder, FifoOrder, PriorityOrder, DeadlineOrder

¹: The MaxEventsPerConsumer property is deprecated in newer versions of CORBA.
²: The PacingInterval property is only relevant in specific scenarios like the PriorityPolicy or DeadlinePolicy.
Structured Event IDL

module CosNotification {
    ...
    struct FixedEventHeader {
        _EventType event_type;
        string event_name;
    };

    struct EventHeader {
        FixedEventHeader fixed_header;
        OptionalHeaderFields variable_header;
    };

    struct StructuredEvent {
        EventHeader header;
        FilterableEventBody filterable_data;
        any remainder_of_body;
    }
    ...
}
Notification Service Constraint Language

- Extended version of CORBA Trading service constraint language
  - Special token “$” for current event and any run-time variable
  - Reserved variable “$curtime”
  - Vendor extensions have “:” prefix

\[
$type\_name == 'CommunicationsAlarm' \text{ and not } \ (event\_name == 'lost\_packet')
\]

\[
$type\_name == 'CommunicationsAlarm' \text{ and } \ priority >= 1 \text{ and } \ priority <= 5
\]

\[
 origination\_timestamp\_high + 2 < curtime\_high
\]

\[
\text{exist } \ threshold\_.type\_id \text{ and } \ threshold\_.type\_id == 'short'
\]
Java Messaging Service

- Client API specification for messaging in Java (JSR 914, 2001)
- "...common set of enterprise messaging concepts..."
- No wire protocol, no security, no error notification
- JNDI typically used to get administered JMS objects
  - Administrative tasks outside of client code
  - ConnectionFactory - used to create JMS provider connection
  - Destination - used for specifying the destination of messages
**JMS Provider**

- „...entity that implements JMS for a messaging product...“ - JAR file
- Producer gives message to provider for delivery
- Consumer gets message synchronously / asynchronously from provider
JMS Relationships

Connection Factory
- Creates Connection

Connection
- Creates Session

Session
- Creates Message Producer
- Creates Message Consumer

Message Producer
- Sends To Destination

Message Consumer
- Receives From Destination

Message

Destination
## JMS Interface Comparison

<table>
<thead>
<tr>
<th>JMS Common Interfaces</th>
<th>PTP-specific Interfaces</th>
<th>Pub/Sub-specific interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
<tr>
<td>Destination</td>
<td>Queue</td>
<td>Topic</td>
</tr>
<tr>
<td>Session</td>
<td>QueueSession</td>
<td>TopicSession</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>QueueSender</td>
<td>TopicPublisher</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>QueueReceiver, QueueBrowser</td>
<td>TopicSubscriber</td>
</tr>
</tbody>
</table>
Developing a JMS Client

- Use JNDI to find a `ConnectionFactory`
- Use JNDI to find one or more `Destination` objects
- Use `ConnectionFactory` to establish connection with demanded delivery feature
- Use `Connection` to create a `Session`
- Create `Producer / Consumer` object through the `Session` to a particular `Destination`
  - Specific class depends on messaging model
JMS Messages

- Standardized message interfaces
- Message implementation by JMS provider
- Message header information
  - Common set of mandatory header fields for receiver
  - Facility for adding properties to header (standard, application-specific, provider-specific)
- Some predefined message body structures
  - Still no promise for interoperability!
JMS Message Header

- **JMSDestination**, **JMSDeliveryMode**
- **JMSMessageID**: Assigned by provider
  - Scope of uniqueness is implementation-specific
- **JMSTimestamp**: Time of delivery to provider
- **JMSCorrelationID**: might be used for relation of response message to request message
- **JMSRedelivered**: Indication for consumer
- **JMSType**: Reference to provider message repository, no defaults specified by JMS
JMS Message Header

- **JMSExpiration**: „...clients SHOULD not receive messages that have expired...“
- **JMSPriority**: 0-4 (normal), 5-9 (expedited)
- Administrator can override **JMSDeliveryMode**, **JMSExpiration** and **JMSPriority**

<table>
<thead>
<tr>
<th>Header Fields</th>
<th>Set By</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>Send Method</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>Client</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>Client</td>
</tr>
<tr>
<td>JMSType</td>
<td>Client</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>Provider</td>
</tr>
</tbody>
</table>
JMS Message Properties

- Additional set of optional header elements with “JMSX” prefix
- Set by provider on send
  - JMSXUserId, JMSXAppId, JMSXProducerTXID
- Set by provider on receive
  - JMSXDeliveryCount, JMSXConsumerTXID, JMSXRecvTimestamp
- Set by client
  - JMSXGroupId, JMSXGroupSeq
JMS Acknowledgment Modes

- Controlled at session level

- **AUTO_ACKNOWLEDGE**
  - Asynchronous mode: Handler acknowledges successful return (???)
  - Synchronous mode: Client has successfully returned from receive()

- **CLIENT_ACKNOWLEDGE**
  - Client calls acknowledge() by itself

- **DUPS_OK_ACKNOWLEDGE**
  - Lazy acknowledge which might lead to duplicates
  - Consumer must be able to handle duplicates
    -> reduces messaging overhead
JMS Message Types

- TextMessage: java.lang.String object
- MapMessage: Set of name/value pairs
- BytesMessage: Stream of bytes
- StreamMessage: Stream of Java primitive values, read sequentially
- ObjectMessage: Serializable Java object

<table>
<thead>
<tr>
<th></th>
<th>boolean</th>
<th>byte</th>
<th>short</th>
<th>char</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>String</th>
<th>byte[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>short</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>String</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>byte[]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
import javax.naming.*;
import javax.jms.*;

Context c = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) c.lookup("ConnectionFactory");
Queue stockQueue = (Queue)c.lookup("StockSource");
Connection connection =ConnectionFactory.createConnection();
Session session; // no TA, automated ack of messages
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer sender = session.createProducer(stockQueue);
String selector = new String("(StockSector = 'Technology')");
MessageConsumer receiver = session.createConsumer(stockQueue,selector);
connection.start();

MapMessage message = session.createMapMessage();
message.setString("Name", "SUNW");
message.setDouble("Value", stockValue);
message.setStringProperty("StockSector", "Technology");
sender.send(message);

MapMessage message = (MapMessage)receiver.receive();
## Messaging In The Internet

<table>
<thead>
<tr>
<th>MOM</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOM message</td>
<td>SMTP message</td>
</tr>
<tr>
<td>Message queue</td>
<td>Mailbox</td>
</tr>
<tr>
<td>Consumer</td>
<td>POP3 client</td>
</tr>
<tr>
<td>Producer</td>
<td>SMTP client</td>
</tr>
<tr>
<td>Queue manager</td>
<td>MTA</td>
</tr>
<tr>
<td>Routing key</td>
<td>To: / Cc: address</td>
</tr>
<tr>
<td>Publish / Subscribe</td>
<td>Mailing list</td>
</tr>
<tr>
<td>Message filter</td>
<td>Server-side spam check</td>
</tr>
<tr>
<td>Message acknowledge</td>
<td></td>
</tr>
<tr>
<td>Transactional messaging</td>
<td></td>
</tr>
<tr>
<td>Time to live</td>
<td></td>
</tr>
</tbody>
</table>