Java is a trademark of Sun Microsystems, Inc.

JavaOne℠

Balancing JMS™ Performance with Reliability

Rob Davies
Progress Inc
Gordon Sivewright
Sun Microsystems Inc

Balancing JMS Performance with Reliability

- Key factors affecting performance and reliability in a JMS messaging application.
- How can I design/tune my application to balance performance and reliability?
- How can I configure my JMS provider to improve performance and reliability?

Speaker info

Rob Davies

- Director of Engineering for Progress FUSE
- Developer Apache ActiveMQ
  - http://activemq.apache.org/

Gordon Sivewright

- Staff Engineer, Sun Microsystems
- Developer Glassfish™ Open Message Queue
  - https://mq.dev.java.net/

Content

- Sending and consuming messages with JMS
  - JMS reliability model
  - Where are the major performance bottlenecks?
- JMS options for controlling reliability
  - Impact on performance
- JMS provider tuning options
- Deployment options for JMS providers
  - Options for boosting performance and reliability

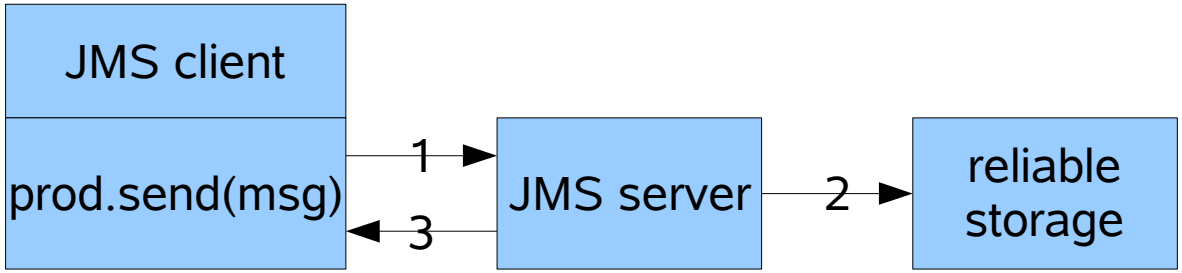
Life cycle of a (reliable) JMS message

- Producer sends message
  - Client producer creates message
  - Producer sends message to server
  - Server persists message
- Server delivers message
  - Server determines which consumers should receive message
  - Server delivers message to client consumer
- Consumer processes and acknowledges message
  - Consumer processes message
  - Consumer sends acknowledgement to server
  - Server processes acknowledgement and updates/removes message

Message send (reliable)

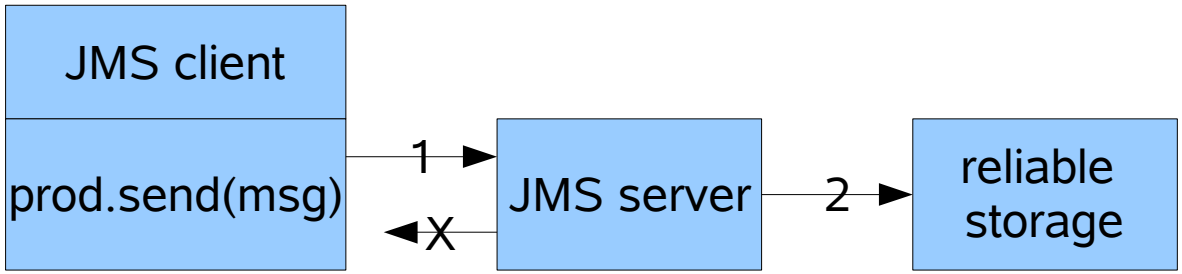
```
MessageProducer producer = session.createProducer(dest);
producer.setDeliveryMode(DeliveryMode.PERSISTENT);
producer.send(msg);
```

1) client sends message
2) server stores message
3) server sends reply to client



Message send (reliable)

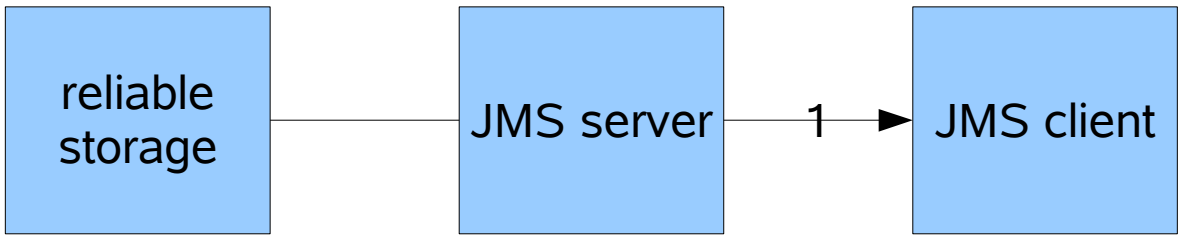
- If send call returns successfully, then message delivery is “guaranteed”.
- If a failure of client or server occurs during the send, the message is in doubt.
  - JMS Spec: “It is up to a JMS application to deal with this ambiguity. In some cases, this may cause a client to produce functionally duplicate messages.”



Message delivery to consumer

```
Connection connection = connectionFactory.createConnection();

Session session =
  connection.createSession(false,Session.CLIENT_ACKNOWLEDGE);

MessageConsumer consumer = session.createConsumer(dest);

connection.start();
```

1) server (asynchronously) delivers message to consumer



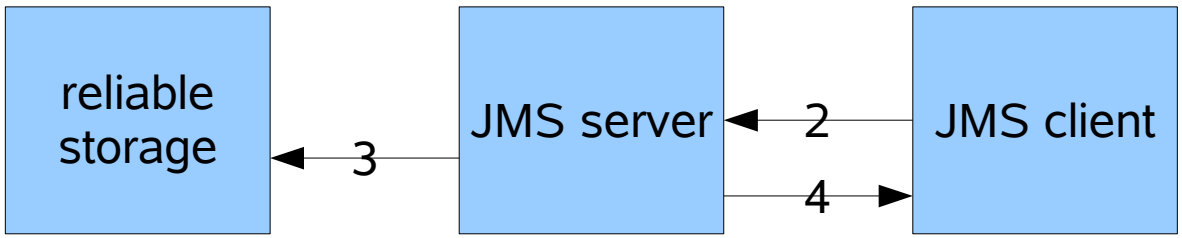
Message acknowledgement (reliable)

```
Message message = consumer.receive();
// process message
message.acknowledge();
```

2) consumer sends back a message acknowledgment to the server

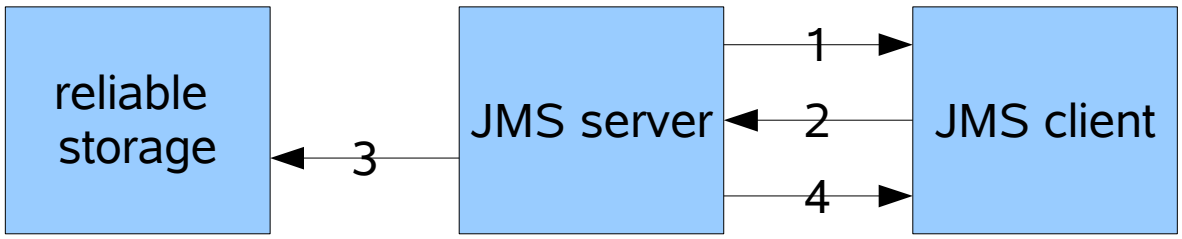
3) server removes message from store

4) server sends reply to client



Message delivery failure

- The message will not be removed from storage until acknowledged by consumer
- Any system failure before message is acknowledged will result in unacknowledged messages being re-delivered to a consumer.



So where are some of the performance bottlenecks?

- Creating JMS components
- Producer waiting for a response from server on send()
- Server persisting message
- Application processing the message
- Consumer waiting for a response from server on acknowledge()

So how can we improve performance?

- Be aware of the performance costs of different messaging options
- Application should use appropriate levels of JMS reliability
- Understand configuration options in your JMS provider that affect performance and reliability

General techniques for improving performance in distributed systems

- Reuse JMS components (caching)
- Reduce network traffic
- Reduce size of messages
- Use non-blocking calls
- Use non-persistent messaging
- Batch work (bundle multiple messages together)
- Process message data serially
- Control resources (sockets, threads, memory)

Managing life cycle of JMS objects

Generally a high cost to create and close JMS objects – reuse them where possible

- Connection (most expensive)
- Session
- Destination
- Consumer
- Producer

Connection life-cycle costs

```
Connection connection =
  ConnectionFactory.createConnection();
```

Creating a connection may involve:

- Creating a socket
- Creating a thread on both client and server
- Sending “hello” message to server
- Sending authentication data
- Exchanging protocol and configuration messages

OpenMQ clients often connect first to portMapper and then reconnect to assigned port.

If you need to create many connections consider connecting directly to JMSService

Connection life-cycle costs

```
connection.close();
```

- Closing a connection may involve:
  - Sending “goodbye” message to server
  - Closing all sessions
  - Closing any temporary destinations owned by connection
  - Stopping connection thread
  - Closing socket

Session life-cycle costs

```
Session session = connection.createSession(...);
```

- Creating a session may involve:
  - Creating a client session thread
  - Request to server

```
session.close();
```

- Closing a session may involve
  - Request to server
  - Stopping client session thread
  - Waiting for any message handling methods to complete
  - Resolving any “in-process” work
    - (open transactions, unacknowledged messages)
  - Closing consumers
  - Closing producers

Consumer life-cycle costs

```
MessageConsumer consumer =
  session.createConsumer(destination);
```

- Creating a consumer may involve:
  - Request to server
  - Pre-sending of messages from server to client

```
consumer.close();
```

- Closing a consumer may involve
  - Request to server:
  - Re-delivery by server of any unacknowledged messages that were pre-sent to consumer

Benefits of caching

Example of object creation and closing costs for OpenMQ:

Relative message production rates for non-persistent queues.

Create or cache connection, session and producer per message send.

- cache connection: ~ 4 times faster
- cache connection and session: ~ 10 times faster
- cache connection, session and producer: ~80 times faster

Managing life-cycle of JMS components

- Avoid repeatedly creating and destroying components. Cache if possible.
- Check if JMS components are being recreated by your messaging frameworks
  - (e.g. Spring JMSTemplate).
- Close connections when no longer needed to save resources.
- Make use of connection pooling solutions

Connection pooling – application server environment

- JMS connection management in an application server is specified by JCA (J2EE Connector Architecture)
  - JCA gives you connection pooling e.g. for outbound connections.
  - JMS connections are wrapped in managed connections.
  - Closing a connection returns it to the pool.
  - Creating a new connection fetches an existing connection from the pool.
  - configuration of connection pools is application server specific .
    - e.g. minPoolSize, maxPoolSize, idleTimeout
- As well as connection pooling, some resource adapters have options for session and producer pooling:
  - e.g. JMSJCA RA: https://jmsjca.dev.java.net

Pooling – outside app server

- Manage your own pooling – reuse connections
- Use a standalone connection manager that provides connection pooling (e.g. JMSJCA)
- Use a Application Client Container (ACC)
  - A light weight container which manages the execution of JEE application client components in their own JVM, in conjunction with an JEE application server
- ActiveMQ provides a PooledConnectionFactory

Managing message data

- Message size has large impact on performance
  - Message cached in memory in server
  - Maybe cached in consumer
  - More work to read and write
  - More network packets
- Message passed between client and server will consist of JMS Message content plus additional provider overhead
- Message body – generally not unmarshalled by server
- Message headers – may be unmarshalled by server (selectors)

Managing message data

- Minimise any redundant application message data
- Avoid sending a series of separate messages where you could consolidate application data into one message.
- Very large messages may have to be specially handled by provider.
  - Consider storing very large messages outside of JMS.
  - JMS could be used to notify consumer of location of data

Provider tuning: reducing message data size

- Consider using provider compression:
  - OpenMQ:

```
MyMsg.setBooleanProperty("JMS_SUN_COMPRESS",true);
```

  - ActiveMQ:

```
ActiveMQConnectionFactory.setMessageCompression(true);
```

- May be able to tune provider specific wire protocol to reduce size of data
  - ActiveMQ: Open wire protocol
    - has choice of tight or loose encoding of packets

Message Delivery Mode

Choose appropriate delivery mode

- Persistent (default) messages must be stored “reliably” and should survive a failure and recovery of JMS server.

- Performance cost:
  - persistent messages normally written to disk.
  - producer.send(message) should block until server has stored message.

Non-persistent delivery mode

Sending non-persistent messages is faster:

- Server does not need to store message reliably
- Message may be lost if server fails
- Typically server will not write message to disk
- Client producer does not need to block on producer.send(msg) waiting for reply from server

Acknowledging non-persistent messages

- Server can process non-persistent message acknowledgements faster than persistent
  - No need to remove message from reliable storage
- Acknowledging non-persistent topic messages may be faster than queue messages
  - JMS Spec allows non-persistent messages to be lost in the event of provider failure, but does not allow duplicates.
  - This implies Queue receivers should block on non-persistent message acknowledgement, to avoid duplicate delivery in the event of client failure.
  - Blocking behavior is not consistent across all JMS providers
  - Topic subscribers may not need to block on non-persistent message acknowledgment.
  - May be able to configure behavior.

Producer flow control

- JMS servers may force producers to slow down if the server is running out of memory or disk storage.
- Message store resources will be used up over time if your application is producing data faster than it is consuming data.
- Ensure server has sufficient resources to avoid producer throttling
- May want to configure destination limits to control message store growth.

Pre-sending messages

- Many providers have option to pre-send or push messages to consumers
- Messages are sent in batches from server to client
- Messages are cached in consumer waiting for the application to process them
- Makes more effective use of network and thread resources.

Pre-sending messages

- Can have big impact on performance
  - particularly for non-persistent messages
- For queues with multiple receivers, pre-sending may lead to some loss of message order.
  - If a queue receiver closes, unconsumed messages may be re-dispatched to another receiver. These messages may appear out of order.
  - JMS spec does not define behavior for multiple queue receivers.

Provider tuning: controlling message pre-sending

- ActiveMQ connectionFactory property:
  - prefetchPolicy.xxxPrefetch (where xxx is one of queue, queueBrowser,topic, durableTopic)
    - Specifies the maximum number of messages that will be present to a consumer.
- OpenMQ connectionfactory property
  - imqConsumerFlowLimit
    - Specifies the maximum number of messages that will be present to a consumer.
  - ImqConsumerFlowThreshold
    - Controls percentage of present messages that need to be consumed before more messages will be sent by the server.

Asynchronous message consumption

- Consuming messages asynchronously may be faster than consuming synchronously with consumer.receive()
- For asynchronous delivery:
  - 1) messages are read from the socket by a connection thread and placed on a session queue.
  - 2) a session thread pulls messages off the session queue and calls MessageListeners.onMessage().
- For synchronous delivery:
  - 1) messages are read from the socket by a connection thread and placed on a session queue.
  - 2) a session thread pulls messages off the session queue and places them on a consumer queue.
  - 3) The application thread pulls messages of the consumer queue by calling consumer.receive()

Concurrent message consumption

- Processing messages concurrently can improve overall throughput
- Will affect order in which messages are processed and acknowledged
- Can achieve concurrency by:
  - Explicitly creating multiple sessions and consumers
  - Using JMS ConnectionConsumer facility (application needs to supply a ServerSessionPool).
  - Can use containers; e.g. Spring `DefaultMessageListenerContainer` or `ServerSessionMessageListenerContainer`
- Concurrent message consumption is provided by MessageDrivenBeans in a JEE application server.

Message acknowledgement modes

- **`AUTO_ACKNOWLEDGE`**
  - Each message is automatically acknowledged when onMessage() completes or receive() returns.
- **`CLIENT_ACKNOWLEDGE`**
  - Allows client to control when messages are acknowledged
  - Acknowledging multiple messages should improve consumer performance.
- **`DUPS_OK_ACKNOWLEDGE`**
  - Provider may optimize message acknowledgement
    - Multiple message acknowledgments grouped together
    - Consumer does not need to block leading to faster performance
  - Duplicate messages may be received in the event of failure

Provider tuning: no-ack acknowledgement modes

- Some JMS providers offer a no-ack mode,
  - messages are removed from server as soon as they are sent to a consumer.
  - This can improve performance at the risk of message loss

e.g. OpenMQ:

```
Session noAckSession =
   ((com.sun.messaging.jms.Connection)connection)
   .createSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE)
   ;
```

Transactions

- Used to group a series of messaging operations into an atomic unit of work
- Local transactions:
  - Transacted sessions
  - Transaction boundaries controlled by calling session.commit();
- Distributed XA transactions
  - Used in conjunction with a transaction manager
  - Provides atomic commit over multiple transactional resources

Transaction performance

- There is additional overhead for JMS provider in processing transactions, e.g.
  - Storing transaction id, transaction state
  - Ensuring atomicity
- Work done in transaction can be consolidated by provider
  - e.g. blocking call to server not required on message sends and acknowledgements
  - Only need to block on session.commit()

Transaction Log

Many JMS providers use transaction logs to implement reliable transactions

- Transactional work is written first to a transaction log, and then to main persistent storage.
- In the event of failure, on recovery, transaction log is replayed to main storage.
- Periodic checkpoints allow main storage to be synchronised and transaction log reset or swapped.

Transaction log performance

- writes to transaction log are typically forced to disk.
- writes to main storage are typically not forced to disk until checkpoint.
- transaction log writes to end of file minimise disk head seeks and allow maximum throughput.
- if possible, consider placing transaction log on separate disk to avoid contention with other IO.
- can group commit concurrent transaction from multiple clients.

XA Transactions

-
- For XA 2-phase commit, transaction has two stages:
  - prepare(): all work is persisted
  - commit(): all work is committed
- A failure after prepare must be recoverable

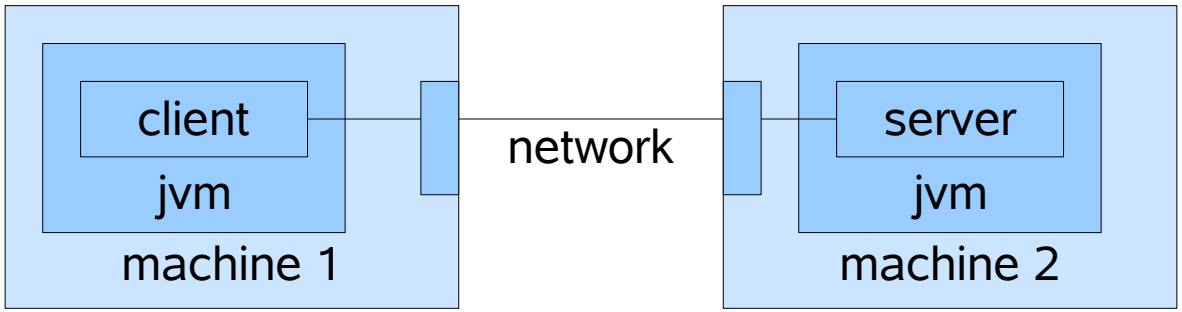
XA transaction optimizations

- Single-phase commit
  - TransactionManager (TM) should use single phase commit if only 1 resource involved in transaction.
  - Single phase commit is much more efficient
    - TM does not need to log transaction progress
    - Single write to JMS transaction log
- IsSameRM optimization
  - If two or more resources involved in a transaction represent the same resource manager, TM can join resources and use single phase commit
  - Check if your JMS provider/RA supports this option

Deployment options for JMS providers

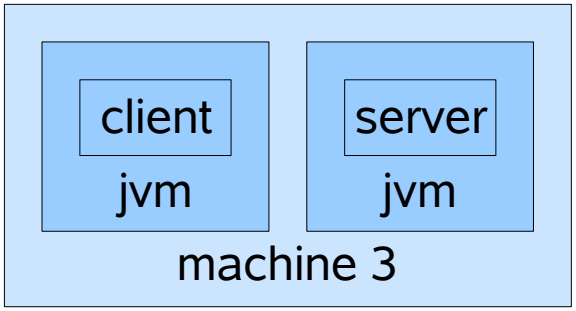
- For improved performance
  - Co-locating client and server
  - Partitioning data over a cluster
  - Server-less messaging
  - System tuning
- For improved reliability
  - Protecting message data against failure
  - Replication strategies

Remote client and server



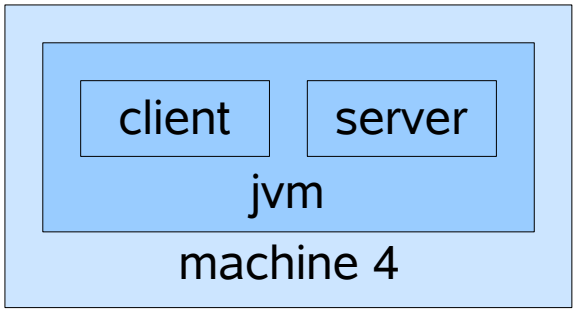
- Typical deployment
- Client and server on separate machines
- Transport over network

Co-located client and server



- Client and server located on same machine
- Running in separate JVMs
- Avoids network hop

Embedded client and server



- Client and server reside in same JVM
- Client can communicate with server using java method calls rather than passing messages over tcp stack
- Avoids marshalling of message data
- Avoids many thread context switches
- Fewer message exchanges (may be no need for a reply message)

Configuring embedded server: OpenMQ

- creating embedded server

```
BrokerInstance brokerInstance =
  ClientRuntime.getRuntime().createBrokerInstance();

Properties props = brokerInstance.parseArgs(args);

BrokerEventListener listener = new
  ExampleBrokerEventListener();

brokerInstance.init(props, listener);

brokerInstance.start();
```

- connecting to embedded server

```
com.sun.messaging.ConnectionFactory cf = new
  com.sun.messaging.ConnectionFactory();

cf.setProperty(ConnectionConfiguration.imqAddressList,
  "mq://localhost/direct");

Connection connection = cf.createConnection();
```

Configuring embedded server: ActiveMQ

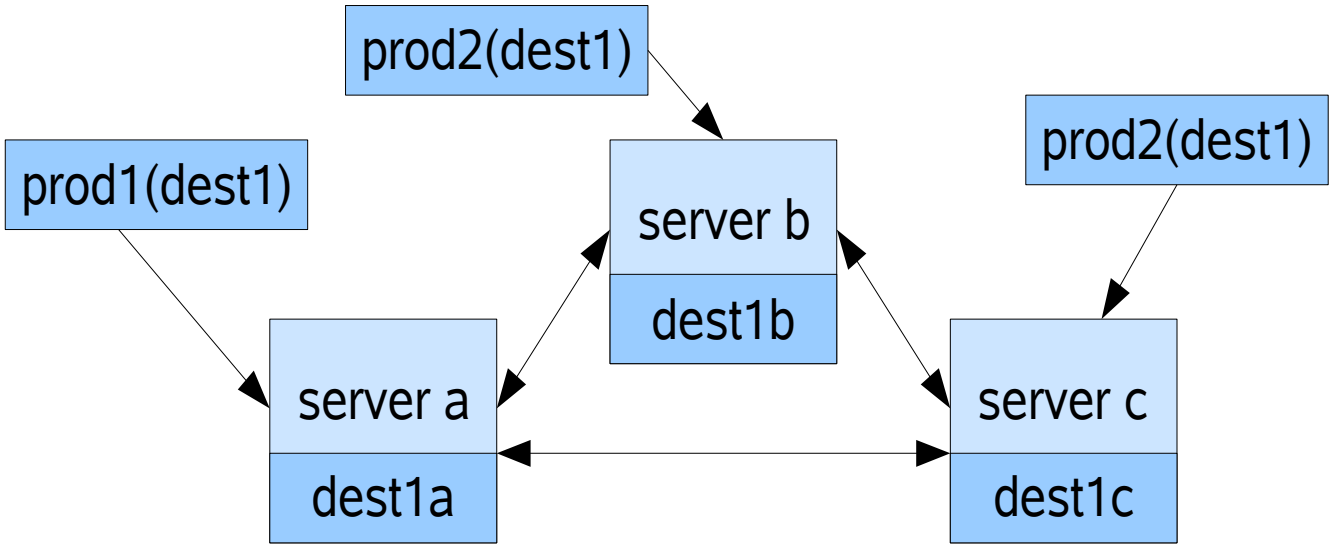
- creating embedded server

```
BrokerService broker = new BrokerService();

broker.setBrokerName("service");

broker.addConnector("tcp://localhost:61616");

broker.start();
```

- connecting to embedded server

```
ActiveMQConnectionFactory cf = new
  ActiveMQConnectionFactory("vm://service");

Connection connection = cf.createConnection();
```

Distributed destinations in OpenMQ



- Fully connected cluster
- Messages stored on the server to which a producer connects.
- Messages delivered to consumers wherever they connect

Distributed destinations in OpenMQ

- Distributed destinations provide scalability
- More servers can be added as number of clients or message load increases
- Each server only needs to store and process a fraction of the destination
- Performance of producers is the same as for a single server
- Performance of consumers receiving messages stored on a remote server is reduced
  - (message and message acknowledgment are routed through local server to remote server)

Server-less messaging with ActiveMQ

- Clients communicate directly with each other
- Use a combination of multicast and broadcast to make efficient use of the network
- Targeted at real time applications
  - eg market data distribution

Other system factors

- Storage tuning
  - Fragmentation of data reduces performance of disk reads and writes
    - Over time message storage may get fragmented and performance may decline
    - Consider periodically consolidating message store, e.g. For openMQ: “imqcmd compact dst”
- Network tuning
  - tcpNoDelay:
  - socket buffer sizes
- JVM, GC settings
  - Consider monitoring GC and checking if provider defaults are appropriate for your application

Improving reliability

- Protecting data against failure
- Data replication strategies

Improving reliability: protecting against JMS server failure

- Use persistent messaging
  - Significantly slower than non-persistent messaging
- Cache non-persistent messages on producer
  - ActiveMQ provides option for caching sent messages on the producer
  - Benefits: In the event of some failures, these can be replayed to avoid message loss.
  - Costs: extra memory overhead. Does not protect against failure of producer.

Improving reliability: protecting against OS/system failure

- How safe are disk writes?
- Depends on write cache settings of JMS provider, OS and disk drives.
- To avoid possible loss of data from buffers, use sync-to-disk
  - Significantly slower than using buffered writes
  - JMS provider can group together concurrent writes to transaction log to reduce the number of syncs
- Alternative: Use data replication strategies

Improving reliability with replication

- Replication protects against localized failures
- Can control number and location of replicas
- Different levels of replication:
  - Persistent data files
  - Databases
  - JMS servers

File system replication

- Replicate persistent data files
- Use a reliable file system (disk arrays, SAN etc) resilient to individual disk failure
- Performance usually acceptable
  - In the event of a server failure, where persistent data is still available, a replacement server can be manually restarted.
  - Can script to automatically restart a failed server.
  - Can use more sophisticated management software to restart server on a different system (e.g. SunCluster).

Database replication

- Store data in a reliable clustered database
  - (e.g. Oracle RAC or MySQL Cluster)
  - OpenMQ and ActiveMQ both offer storage on databases through JDBC
  - OpenMQ allows active servers in a cluster to monitor and automatically takeover the data of failed servers.
  - Storing messages in databases may give poorer performance than using JMS provider message stores

JMS server replication

- Replication of message data between multiple JMS servers
- ActiveMQ supports a master-slave configuration
  - Slave acts as a hot-standby to take over if master fails.

Conclusions

- There are many options available in JMS.
- We have shown how some of these options affect messaging performance and reliability.
- Improving performance often reduces reliability and vice versa.
- Check you are using JMS appropriately for each of your message flows in your application.
- Understand the options and configurations available in your JMS provider and choose those that give you the right balance of performance and reliability.

JavaOne℠

Thank You

Gordon Sivewright
gordon.sivewright@sun.com
Rob Davies
rodavies@progress.com

Sun microsystems