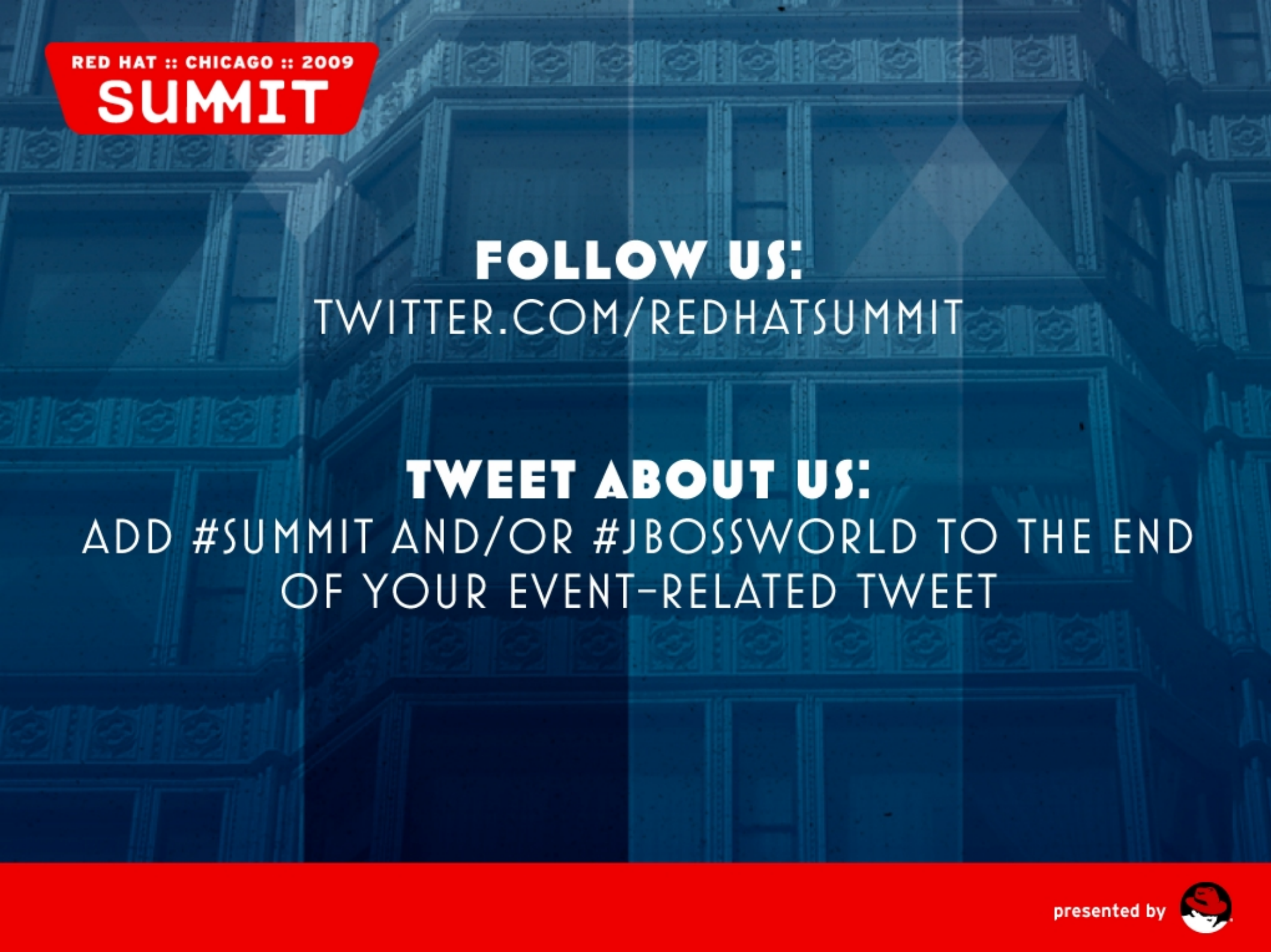

RED HAT :: CHICAGO :: 2009
SUMMIT
FOLLOW US:
TWITTER.COM/REDHATSUMMIT
TWEET ABOUT US:
ADD #SUMMIT AND/OR #JBOSSWORLD TO THE END
OF YOUR EVENT-RELATED TWEET
presented by

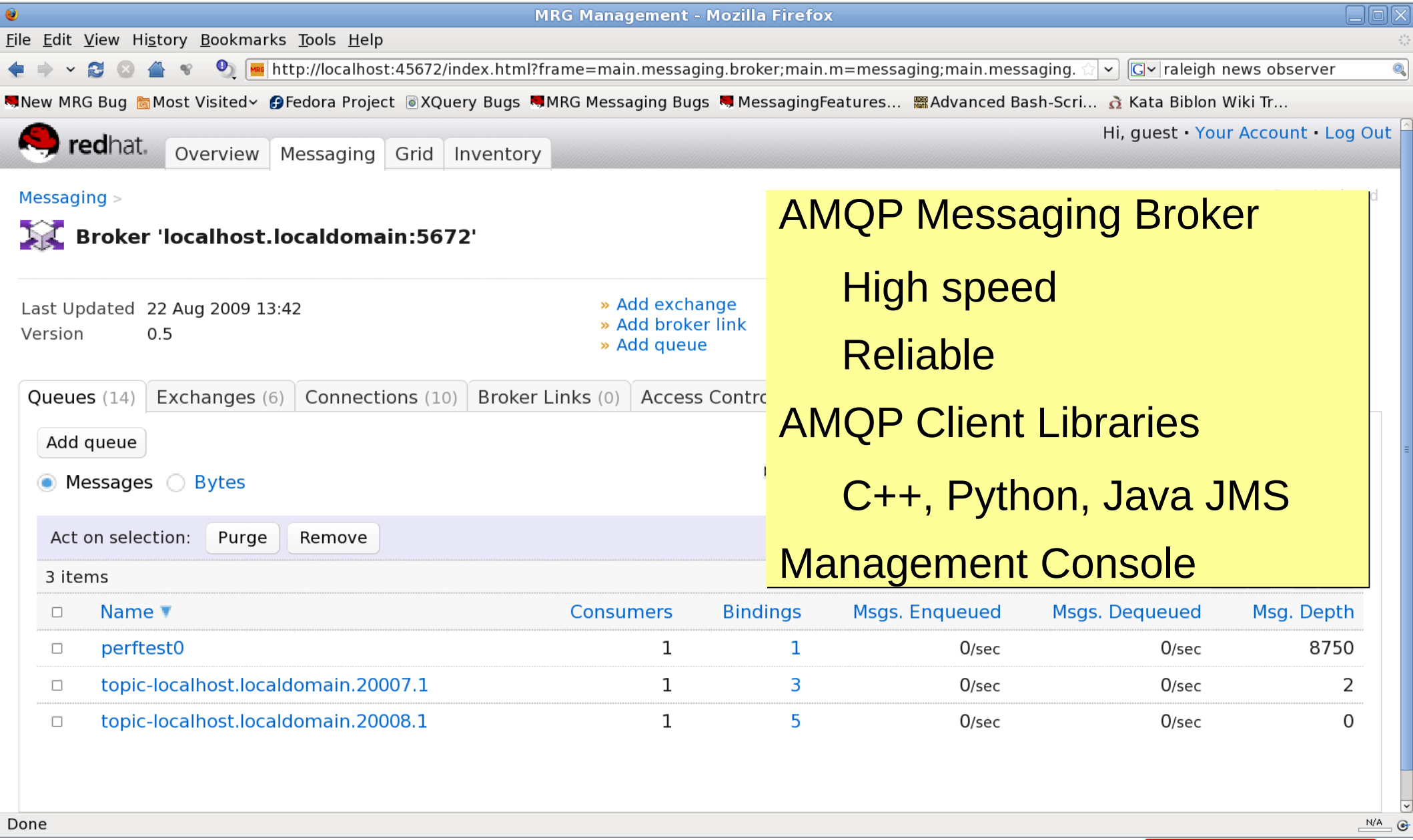
RED HAT :: CHICAGO :: 2009
SUMMIT

MRG Messaging:
A Programmer's Overview

Jonathan Robie
jonathan.robie@redhat.com
Software Engineer, Red Hat
2009-Sept-03

presented by

Red Hat MRG Messaging



AMQP Messaging Broker

- High speed
- Reliable

AMQP Client Libraries

- C++, Python, Java JMS

Management Console

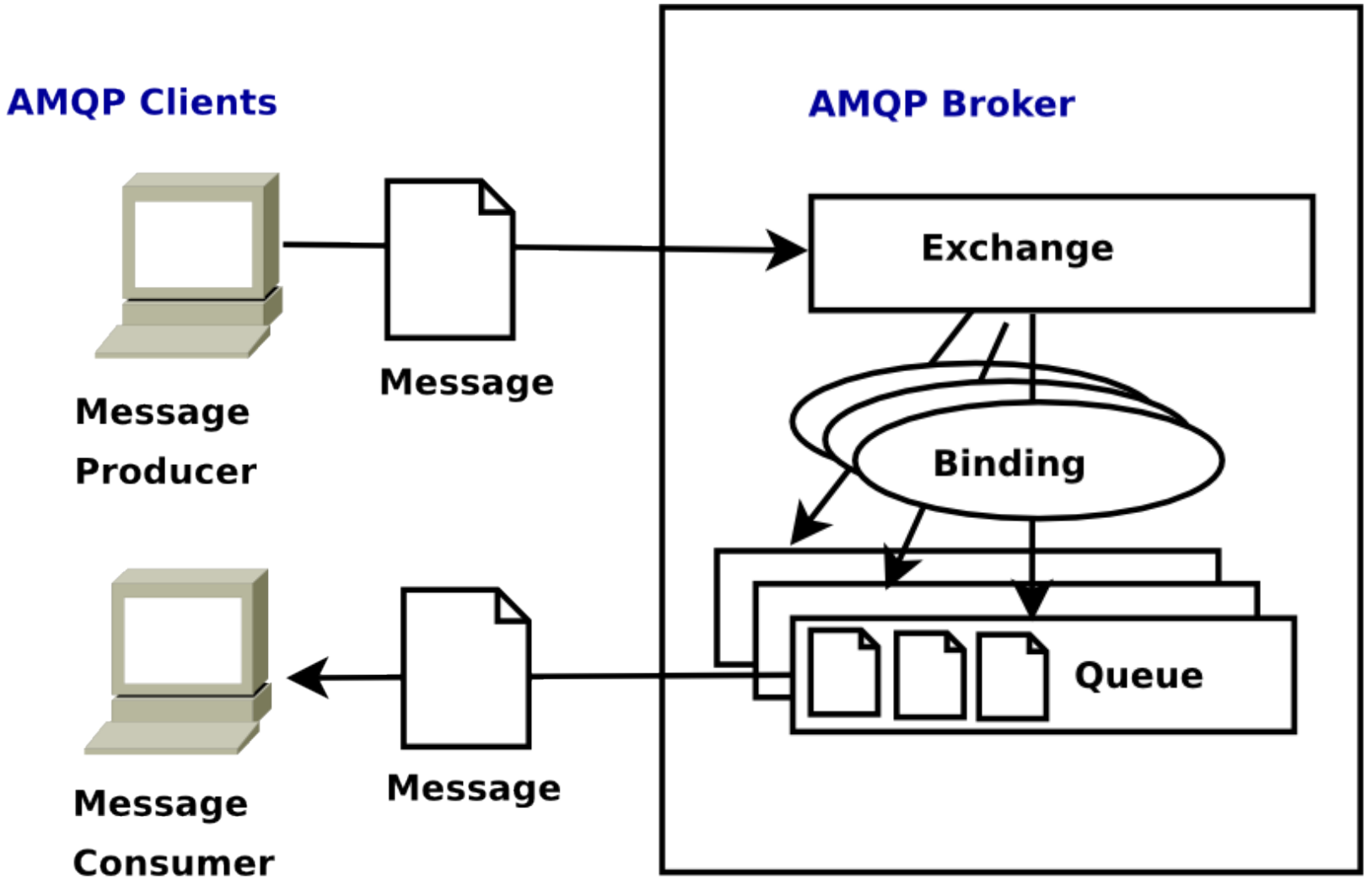
Why can't we all just talk to each other?

- “Programs communicate by sending messages”
  - But how can programs in different languages and on different platforms do this?
- Java JMS API ... is just a Java API
  - Only for Java
  - No interoperable protocol – even among JMS implementations
- Proprietary messaging software ... is proprietary
- Success requires good, interoperable implementations
  - Open Standards (AMQP)
  - Open Source (Apache Qpid, QpidComponents.org)
  - Vendor buy-in (Red Hat, Microsoft, JBoss...)

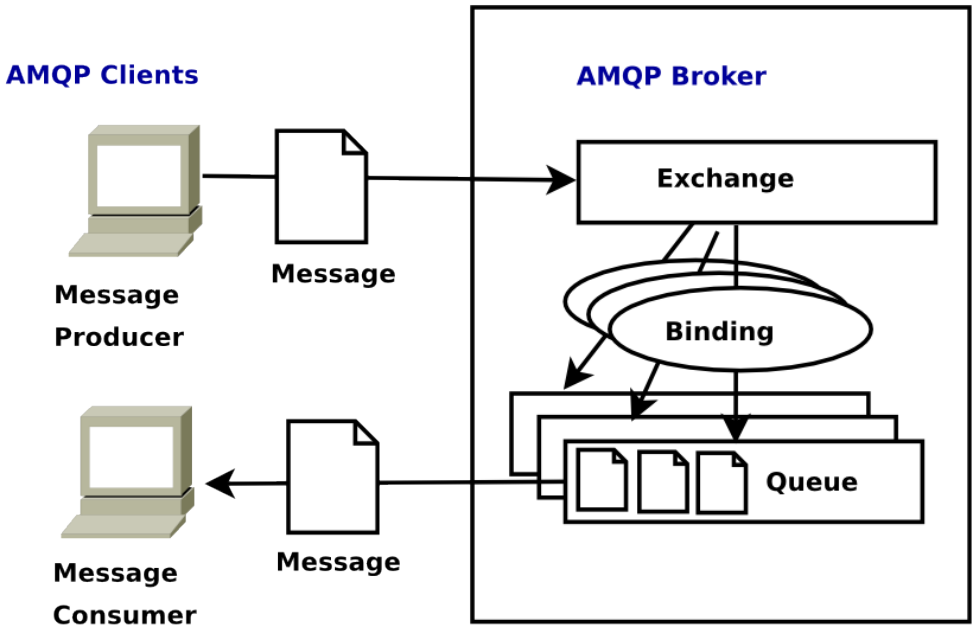
Agenda

- Infrastructure for Messaging
  - AMQP: Advanced Message Queuing Protocol
  - Apache Qpid APIs
  - MRG Messaging
- Programming with MRG Messaging
  - C++, Python, Java JMS
  - Point-to-Point, Publish-Subscribe, Broadcast, Request-Response, XML Content Routing
- Advanced Features
  - Guaranteed Delivery, Transactions, High Availability Messaging Clusters, Federation

AMQP Model: Message, Exchange, Binding, Queue



AMQP Model



Exchanges

Message Producers write Messages to Exchanges

Queues

Message Consumers read Messages from Queues

Queues store Messages until they are read

Bindings

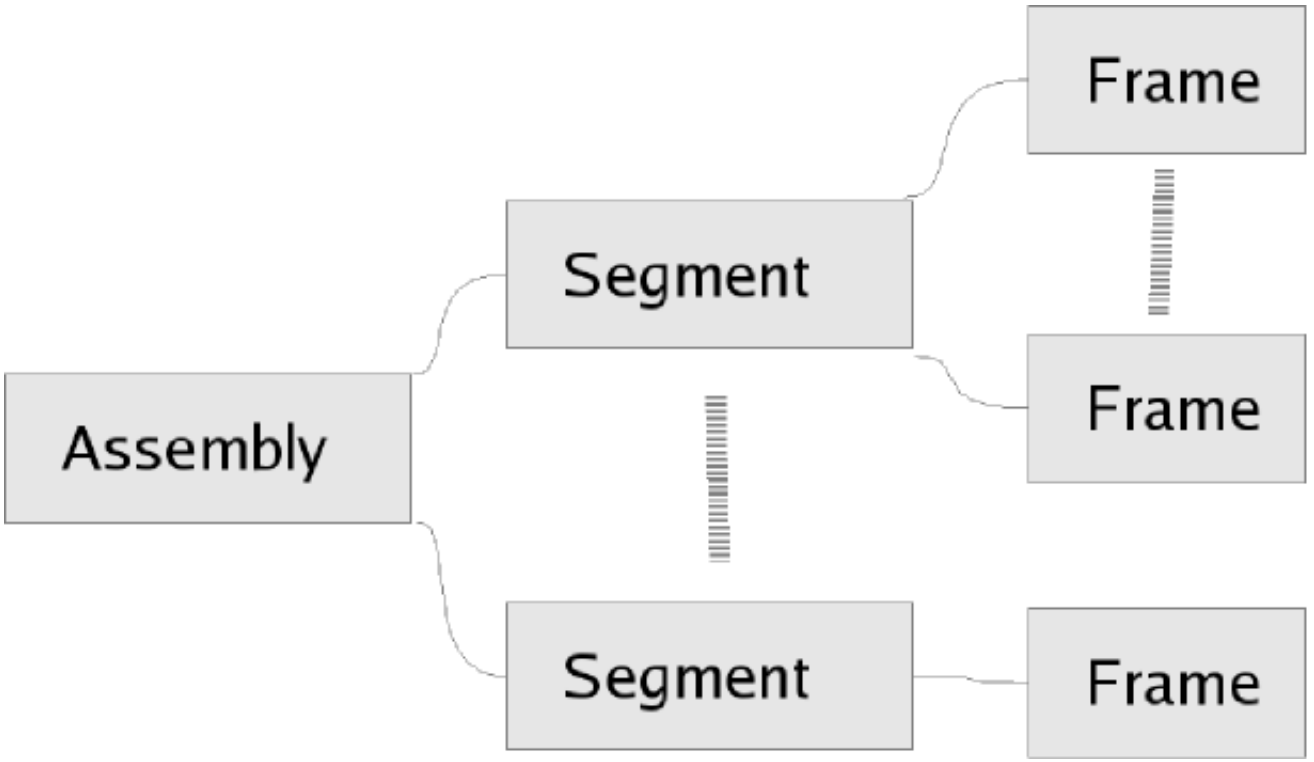
A Binding connects a Queue to an Exchange, specifies Message routing

AMQP Message (Model View)

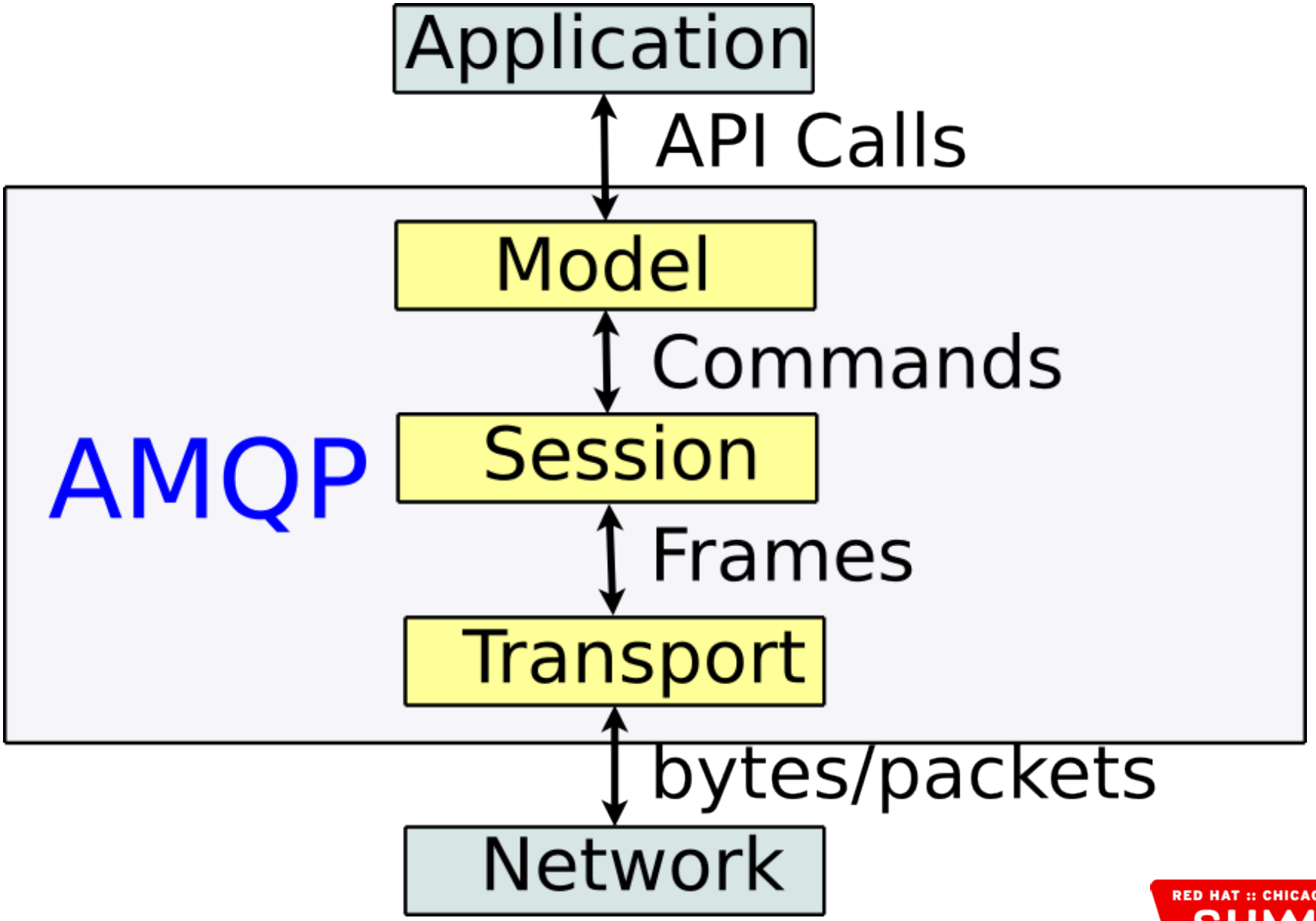
Message properties
- Message-Id
- Reply-to
- Content-type
- Content-encoding
- etc.

Delivery properties
- Routing Key
- Time to live
- Persistence (delivery mode)
- priority
- expiration
- etc.

AMQP Message (Session View)



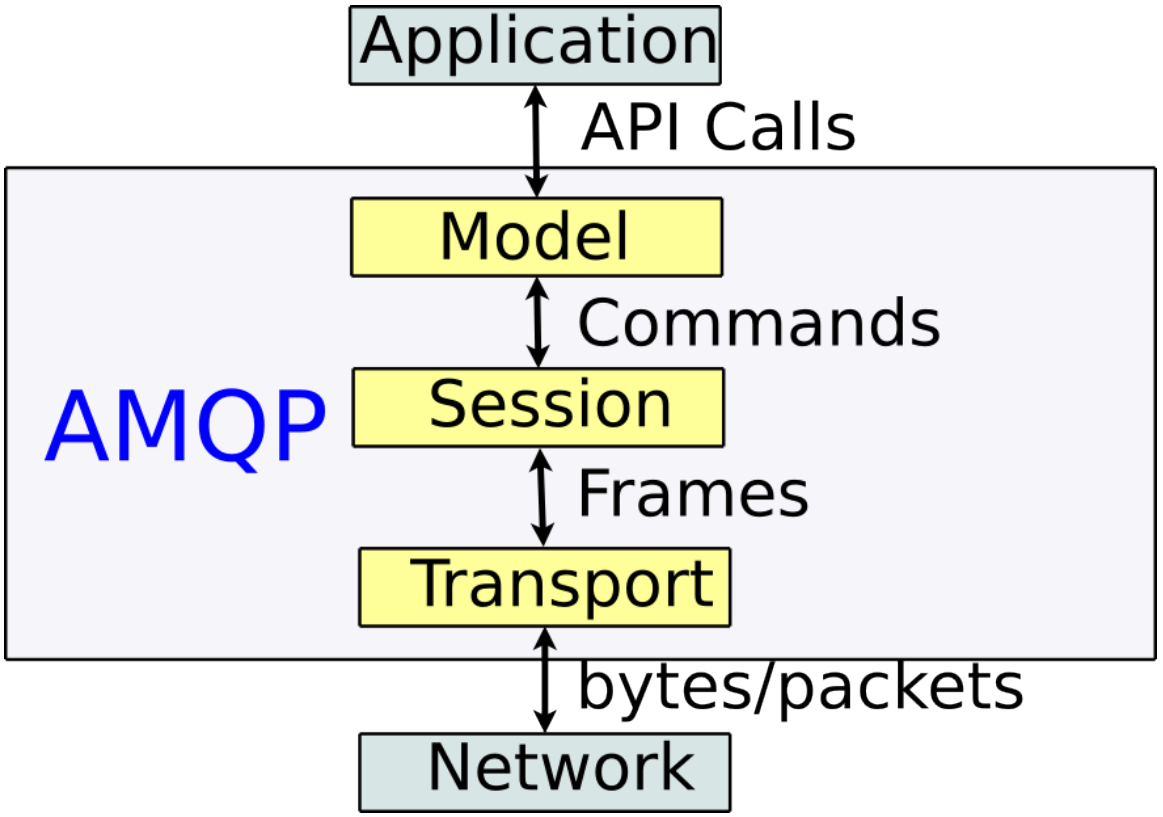
AMQP Layers


Application
API Calls
Model
Commands
AMQP
Session
Frames
Transport
bytes/packets
Network

AMQP Layers: Model

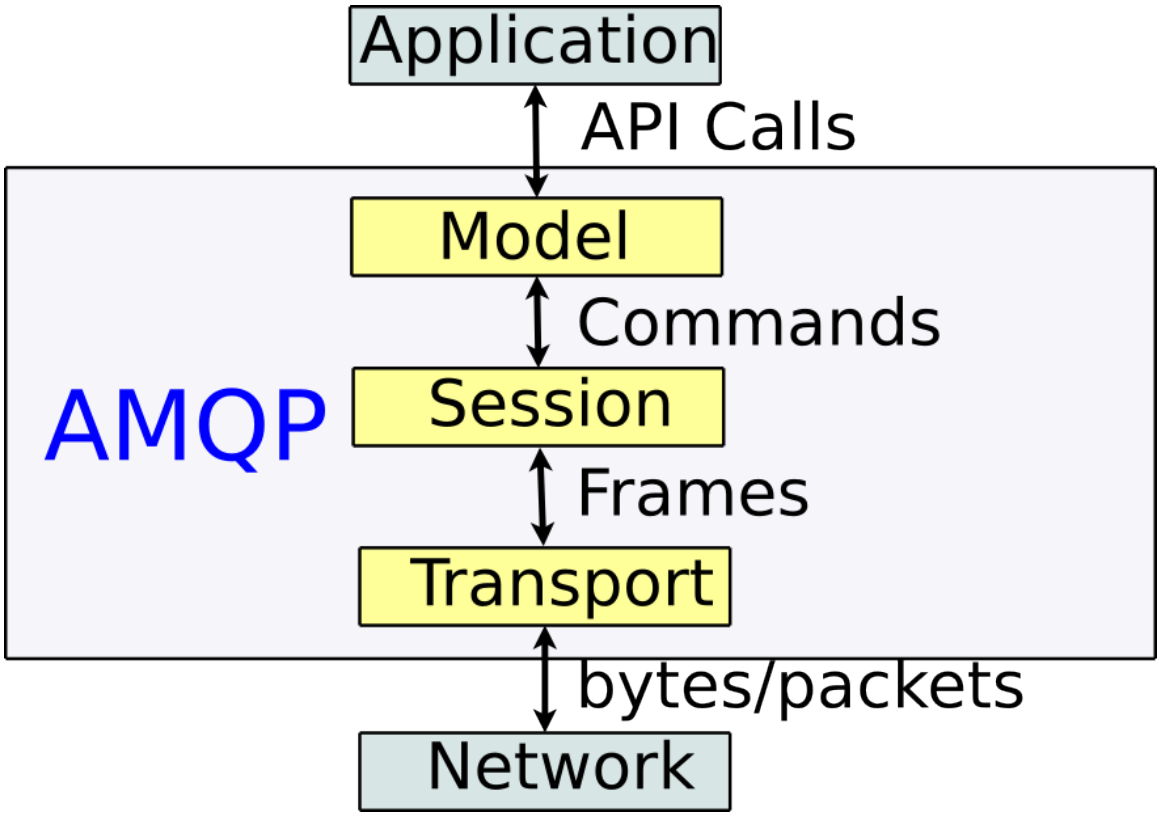
- “Abstract API”
  - Language independent
  - C++, Python, Java JMS APIs map to the AMQP Model
  - Java JMS mapping preserves JMS semantics
- Components
  - Message, Exchange, Binding, Queue, etc...
- Commands
  - Create Queue, Bind Queue to Exchange, Send Message, etc ...
- Datatypes
  - Language and platform-independent types

AMQP Layers: Session



- Session defines interaction between client and server
- Supports replay, synchronization, error handling
- Translates AMQP commands to frames for network

AMQP Layers: Transport



- Transport transfers bytes/packets across the network
- MRG Messaging network protocols
  - TCP/IP
  - Infiniband

AMQP Exchange Types

Direct Exchange

- Routes to queue if message's routing key matches binding key
- Point-to-Point
- Request/Response

Default Exchange

- Routes to queue if message's routing key matches queue name
- Same use cases as Direct Exchange

Fanout Exchange

- Routes to every bound queue
- Broadcast

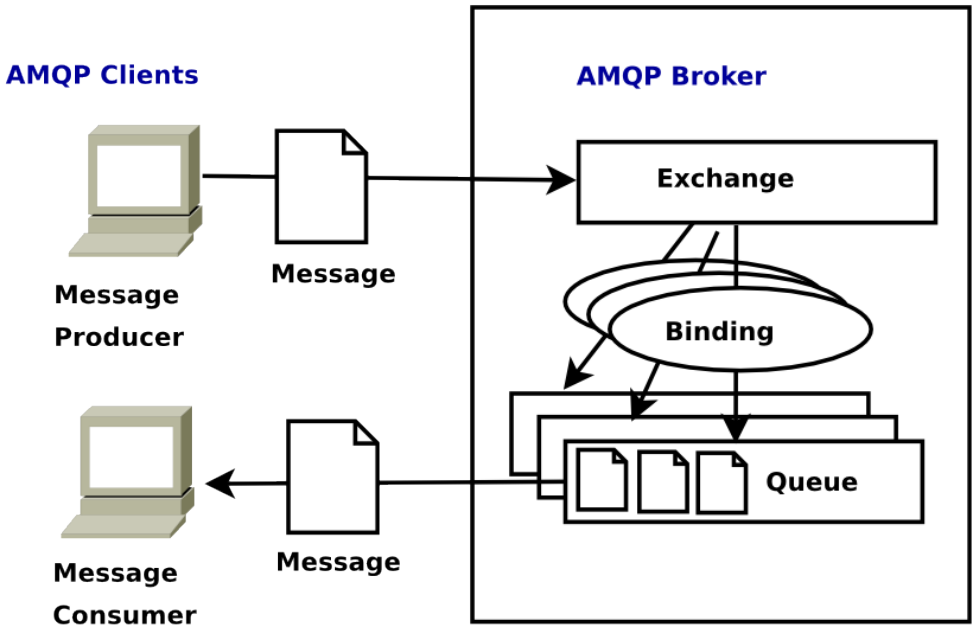
Topic Exchange

- Matches message's routing key against binding key, using wildcards
- Publish/Subscribe

Header Exchange

Custom Exchanges

Apache Qpid APIs



- Programming APIs for AMQP Clients
- C++, Python APIs closely based on AMQP Model
- Java already has a messaging API – Java JMS
  - Java JMS client uses URLs to address AMQP components
  - Interoperable with C++, Python

C++ and Python APIs

Apache Qpid C++ API

Summary
http://www.ibiblio.org/jwrobie/blog/?p=18

Reference (Doxygen)
http://qpid.apache.org/docs/api/cpp/html/index.html

Apache Qpid Python API

Summary
http://www.ibiblio.org/jwrobie/blog/?p=29

Reference (Epydoc)
http://qpid.apache.org/docs/api/python/html/index.html

A Simple C++ Messaging Program

```
#include <qpid/client/Connection.h>
  #include <qpid/client/Session.h>
  #include <qpid/client/Message.h>

using namespace qpid::client;
  using namespace qpid::framing;

int main() {
      const char* host = "127.0.0.1";
      int port = 5672;

    Connection connection;
      try {
          connection.open(host, port);
          Session session = connection.newSession();

        Message message;
          message.getDeliveryProperties().setRoutingKey("routing_key");
          message.setData("Hi, Mom!");

        session.messageTransfer(arg::content=message,
                                arg::destination="amq.direct");
          connection.close();
      } catch(const std::exception& error) {
          std::cout << error.what() << std::endl;
      }
  }
```

Using Java JMS with MRG Messaging

- Configure Queues, Exchanges – several possible ways
  - Using JNDI properties (see following slide)
  - Using the MRG Management Console
  - Using Low Level Java API
  - Using programs in C++ or Python
- Use vanilla Java JMS for messaging once configured

Configuring Java JMS via JNDI Properties

connectionfactory.<jndiname>

The Connection URL used by the connection factory to create connections.

queue.<jndiname>

A JMS queue, implemented as an amq.direct exchange.

topic.<jndiname>

A JMS topic, which is implemented as an amq.topic exchange.

destination.<jndiname>

Can be used to define any amq destination, using a “Binding URL”.

Example JNDI properties file for Java JMS

```
# JNDI properties file

java.naming.factory.initial =
  org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# register some connection factories
  # connectionfactory.[jndiname] = [ConnectionURL]
  # See MRG Messaging Tutorial for ConnectionURL format

connectionfactory.qpidConnectionfactory =
  amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'

# Register an AMQP destination in JNDI
  # destination.[jndiName] = [BindingURL]
  # See MRG Messaging Tutorial for BindingURL format

destination.directQueue =
  direct://amq.direct//message_queue?routingkey='routing_key'
```

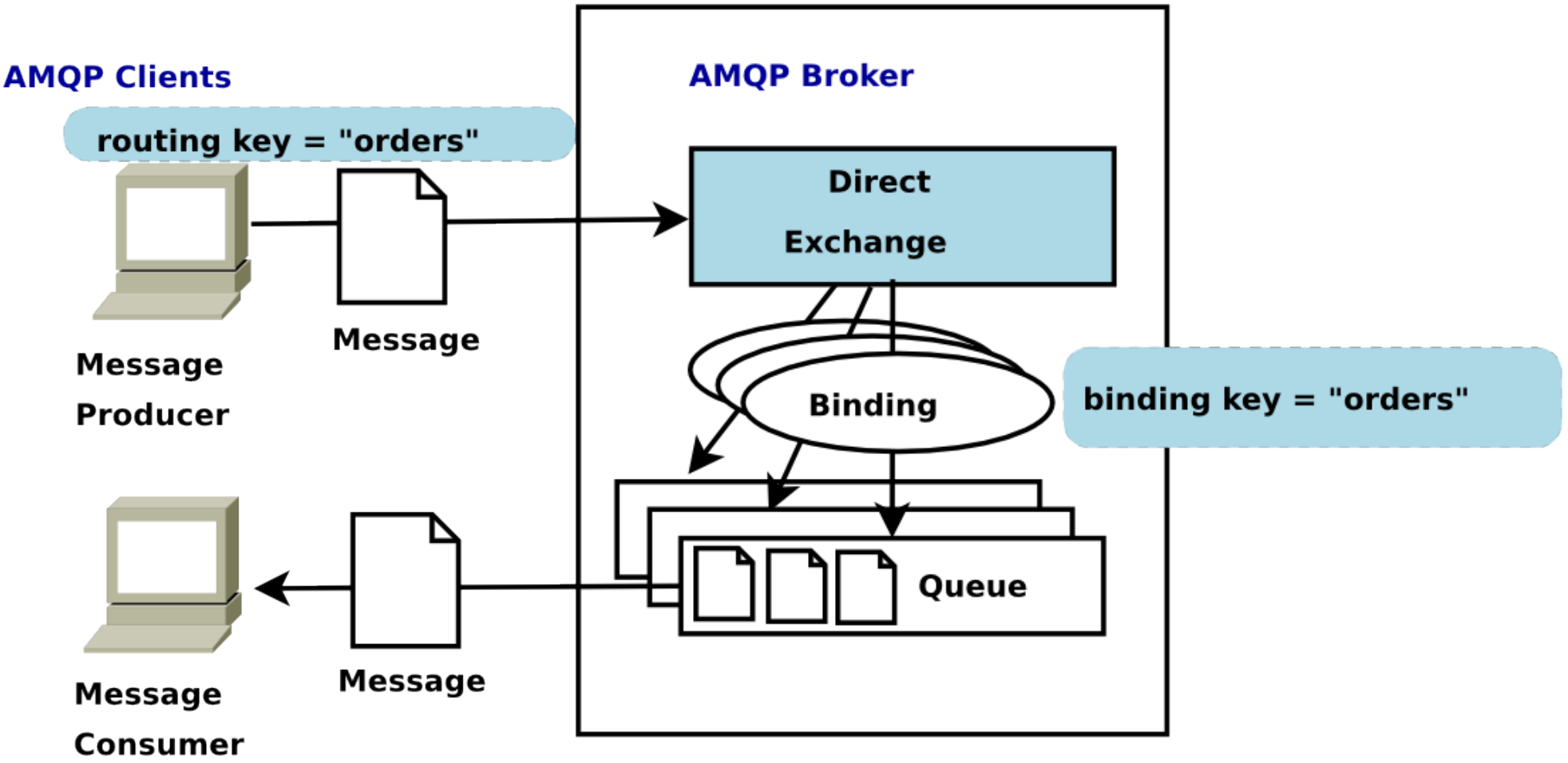
Using JNDI to create Java JMS Session, Connection, Destination

```
// Load JNDI properties

Properties properties = new Properties();
  properties.load(this.getClass().getResourceAsStream("direct.properties"
  ));

// Create the JNDI initial context using JNDI properties

Context ctx = new InitialContext(properties);

// Look up Java JMS destination and connection factory

Destination destination = (Destination)ctx.lookup("directQueue");

ConnectionFactory connFact =
  (ConnectionFactory)ctx.lookup("qpidConnectionfactory");

// Create Java JMS connection and the session using this
  // connection factory

Connection connection = connFact.createConnection();

Session session = connection.createSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

Once configured, use the Java JMS API

```
// Using standard Java JMS to send a message

MessageProducer messageProducer = session.createProducer(destination);
  TextMessage message;

message = session.createTextMessage("This is a text, this is only a
  text ...");

messageProducer.send(message, getDeliveryMode(),
         Message.DEFAULT_PRIORITY,
         Message.DEFAULT_TIME_TO_LIVE);
```

Direct Exchange: Point-to-Point



Point-to-Point: Declaring a Queue and Binding

```
// arg::queue specifies the queue name

session.queueDeclare(arg::queue="message_queue");

// bind "message_queue" to "amq.direct" exchange

session.exchangeBind(arg::exchange="amq.direct",
        arg::queue="message_queue",
        arg::bindingKey="routing_key");
```

Point-to-Point: Sending Messages

```
Message message;

// Set routing key

message.getDeliveryProperties().setRoutingKey("routing_key");

// Send some messages

for (int i=0; i<10; i++) {
    stringstream message_data;
    message_data << "Message " << i;

  message.setData(message_data.str());
    session.messageTransfer(arg::content=message,
              arg::destination="amq.direct");
  }

message.setData("That's all, folks!");
  session.messageTransfer(arg::content=message,
                          arg::destination="amq.direct");
```

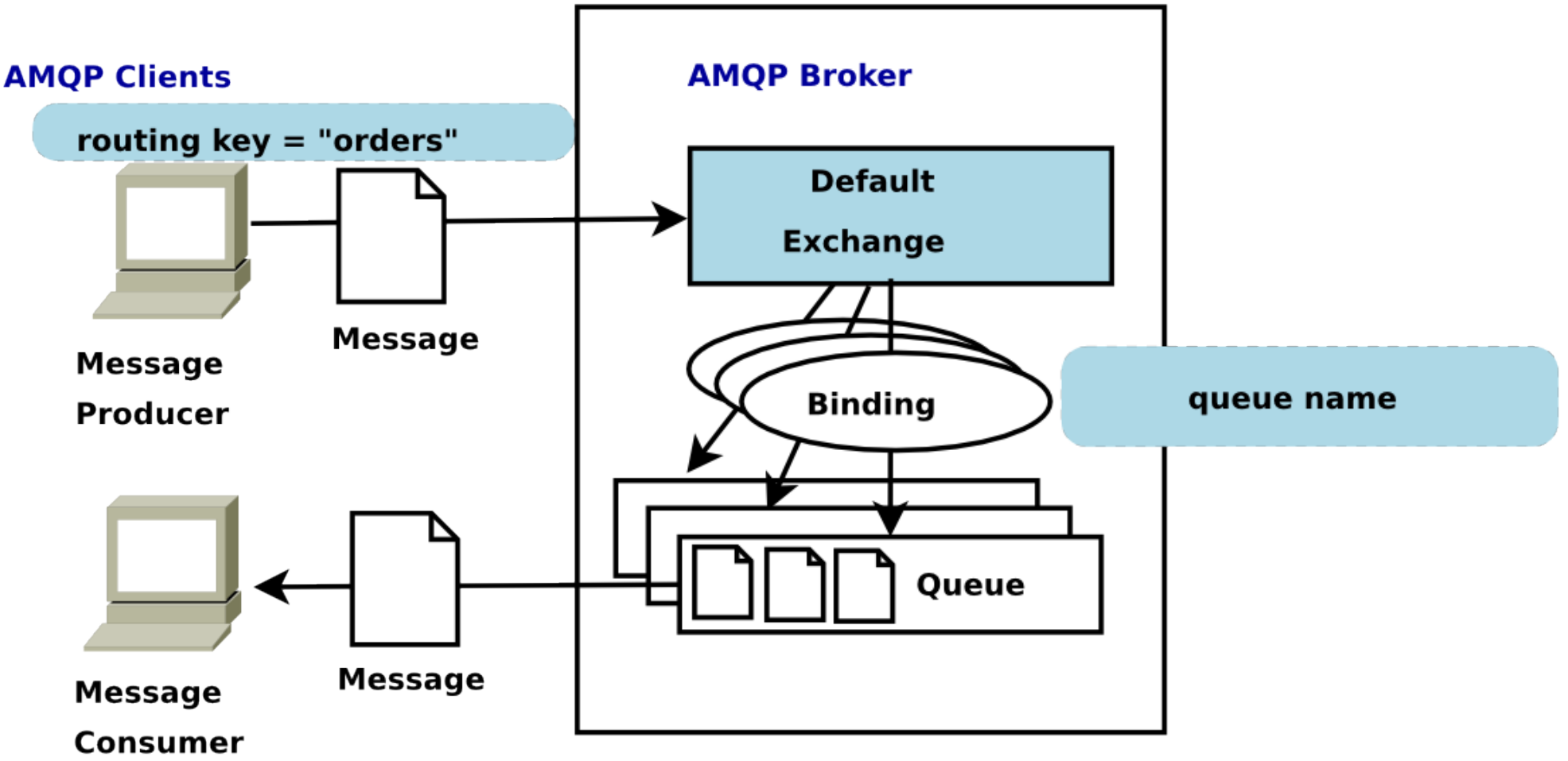
Point-to-Point: Receiving Messages (Listener)

```
// Create a Listener, Derived from MessageListener

class Listener : public MessageListener {
    private:
      SubscriptionManager& subscriptions;

  Public:
      Listener(SubscriptionManager& subscriptions);
      virtual void received(Message& message);
  };

// Implement constructor

Listener::Listener(SubscriptionManager& subs) : subscriptions(subs) {}

// Implement Listener::received()

void Listener::received(Message& message) {
    std::cout << "Message: " << message.getData() << std::endl;

  if (message.getData() == "That's all, folks!") {
        std::cout << "Shutting down listener for "
                  << message.getDestination()
                  << std::endl;

      subscriptions.cancel(message.getDestination());
    }
  }
```

Point-to-Point: Receiving Messages (Listener)

```
// Subscribe Listener to "message_queue"

SubscriptionManager subscriptions(session);
  Listener listener(subscriptions);
  subscriptions.subscribe(listener, "message_queue");

// Receive messages until Listener::received() cancels subscription

subscriptions.run();
```

Default Exchange

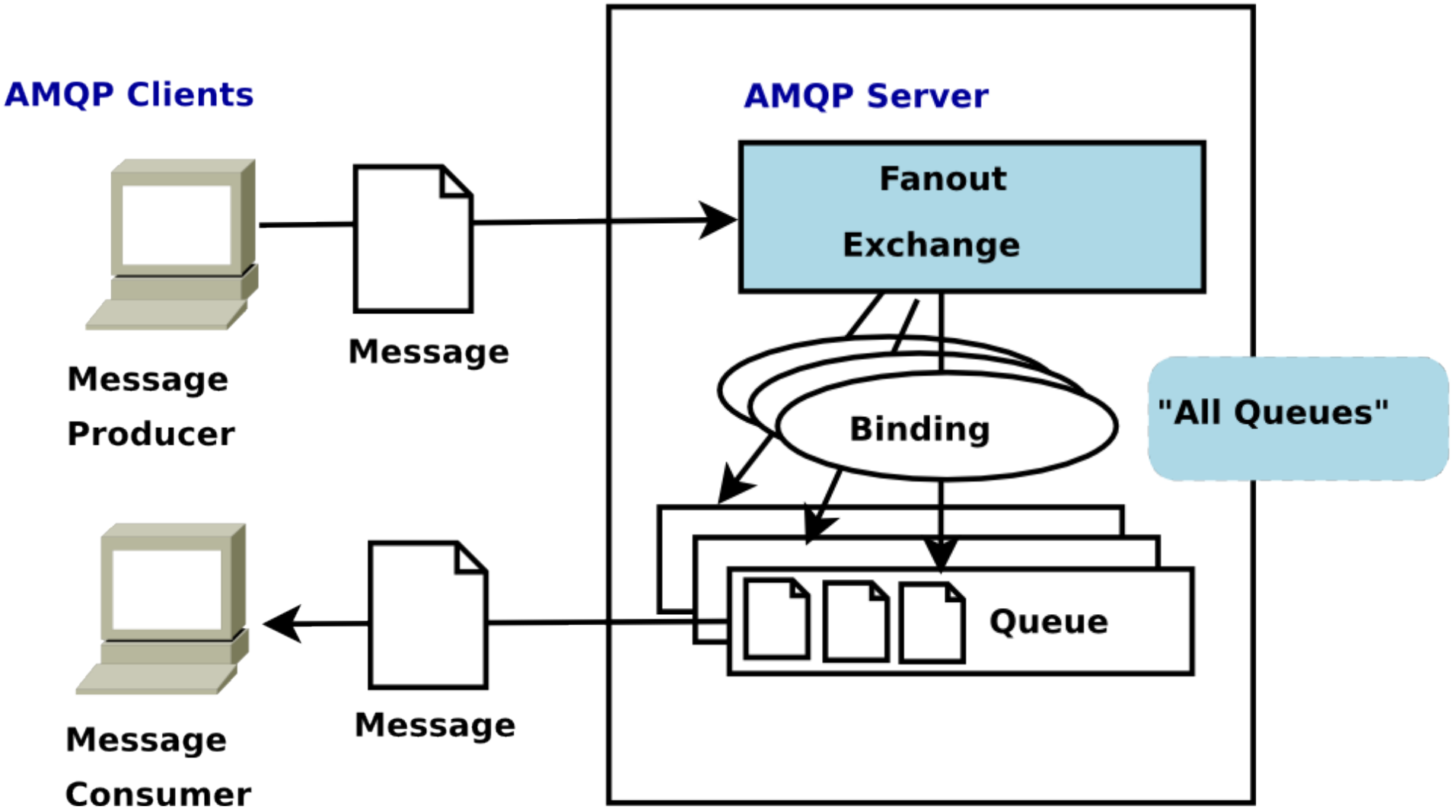


Default Exchange vs. Direct Exchange

- Default Exchange automatically gets a binding for each queue, routing key = queue name
- Same use cases, very similar to program
- Less code than Direct Exchange – but less explicit

```
// Default Exchange vs Direct Exchange

 session.queueDeclare(arg::queue="message_queue");
  session.exchangeBind(arg::exchange="amq.direct",
      arg::queue="message_queue",
      arg::bindingKey="routing_key");

 message.getDeliveryProperties().setRoutingKey("routing_key");
   session.messageTransfer(arg::content=message,
             arg::destination="amq.direct");
```

Fanout Exchange



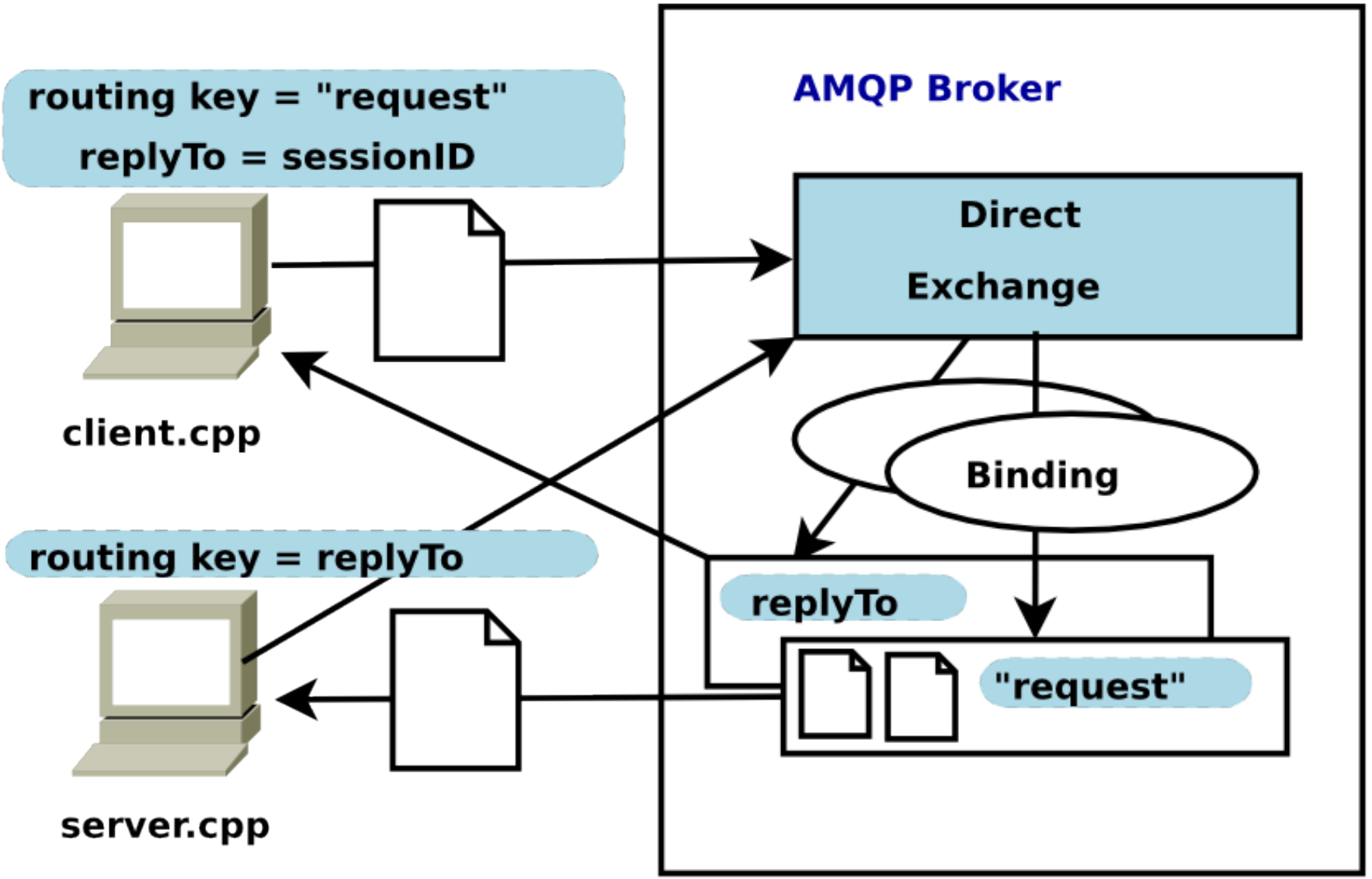
Fanout Exchange: Publisher

```
// Same as direct, except for the destination

session.messageTransfer(arg::content=message,
                        arg::destination="amq.fanout");
```

Fanout Exchange: Subscriber

```
// Exclusive autodelete queue using Session ID as name

std::string myQueue=session.getId().getName();

session.queueDeclare(arg::queue=myQueue,
          arg::exclusive=true,
          arg::autoDelete=true);

session.exchangeBind(arg::exchange="amq.fanout",
          arg::queue=myQueue);

// Create a listener and subscribe it to the queue.

SubscriptionManager subscriptions(session);
  Listener listener(subscriptions);
  subscriptions.subscribe(listener, myQueue);
  subscriptions.run();
```

Direct Exchange: Request / Response Apps



Request / Response Client

```
// Exclusive autodelete queue using Session ID as name
  stringstream response_queue;
  response_queue << "client" << session.getId().getName();

session.queueDeclare(arg::queue=response_queue.str()
          arg::exclusive=true,
          arg::autoDelete=true);

session.exchangeBind(arg::exchange="amq.direct",
          arg::queue=response_queue.str(),
          arg::bindingKey=response_queue.str());

// Create a listener for the response queue and listen for responses

SubscriptionManager subscriptions(session);
  Listener listener(subscriptions);
  subscriptions.subscribe(listener, response_queue.str());

// Use name of the response queue as the Reply To for requests

Message request;
  request.setData("ping!");
  request.getDeliveryProperties().setRoutingKey("request");
  request.getMessageProperties().setReplyTo(ReplyTo("amq.direct",
          response_queue.str()));

subscriptions.run();
```

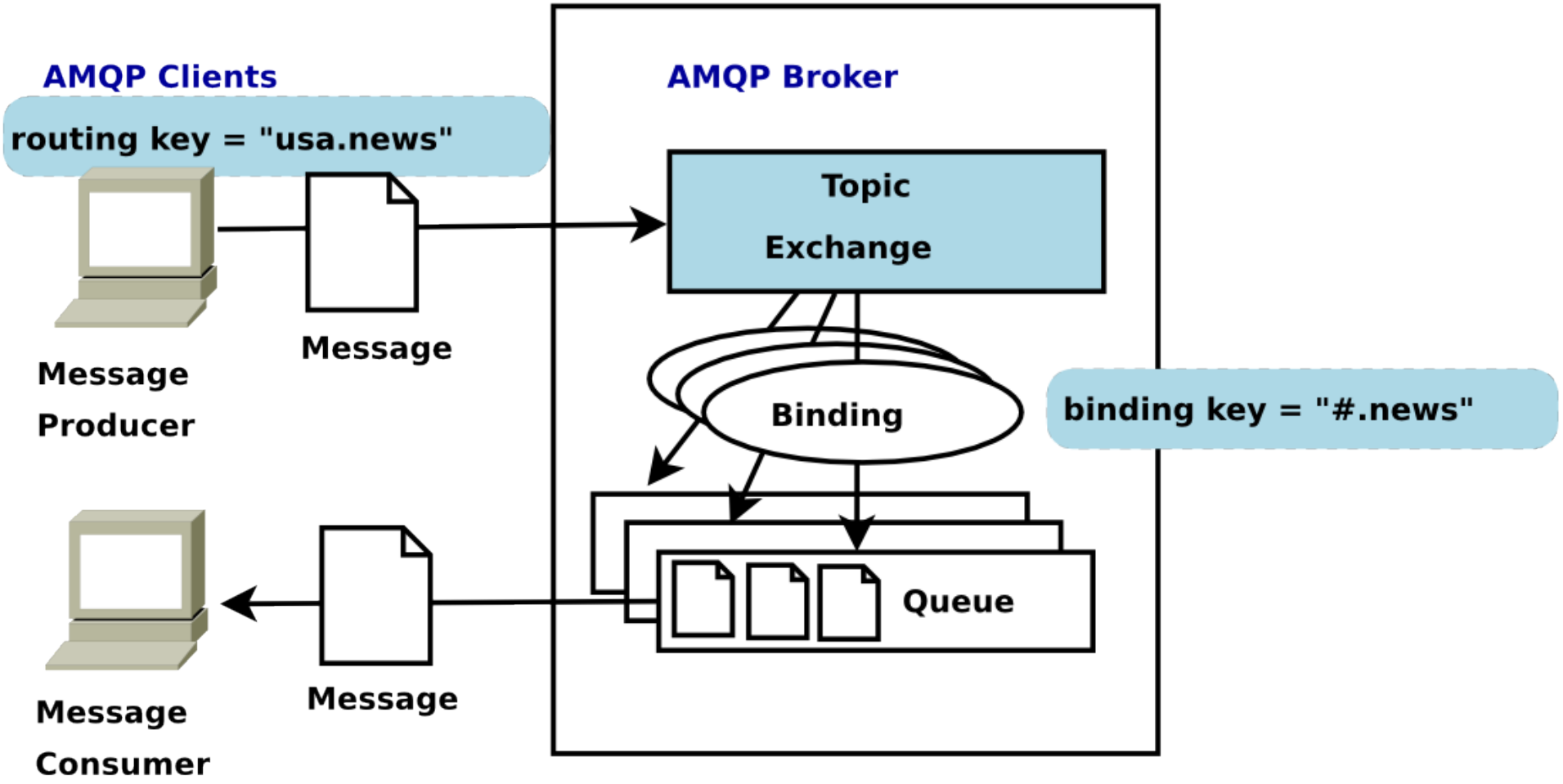
Request / Response Server

```
// Create a request queue for clients to use

string request_queue = "request";

session.queueDeclare(arg::queue=request_queue);

session.exchangeBind(arg::exchange="amq.direct",
      arg::queue=request_queue,
      arg::bindingKey=request_queue);

// Create a listener and subscribe it to the request queue

SubscriptionManager subscriptions(session);

Listener listener(subscriptions, session);
  subscriptions.subscribe(listener, request_queue);
  subscriptions.run();
```

Request / Response Server – Listener::received()

```
void Listener::received(Message& request) {
      Message response;

    // Routing key for response is in request's replyTo
      string routingKey;
      if (request.getMessageProperties().hasReplyTo()) {
          routingKey =
          request.getMessageProperties().getReplyTo().getRoutingKey();
      } else {
          std::cout << "Error: " << "No replyTo ..." << std::endl;
          return;
      }


      // Send response to the client
      response.setData("pong!");
      response.getDeliveryProperties().setRoutingKey(routingKey);
      asyncSession.messageTransfer(arg::content=response,
         arg::destination="amq.direct");
  }
```

Topic Exchanges



Topic Exchange: Wildcard Syntax

Routing Keys are divided into “words”

- “.” is a delimiter
- “usa.news” has two words
- “usa.news.breaking” has three words

Binding Keys can have wildcards

- “*” matches any one word
- “#” matches one or more words

Examples

- “usa.news” matches “usa.news”, “*.news”, “usa.*”, “usa.#”
- “usa.news.breaking” matches “usa.#” but not “usa.*”

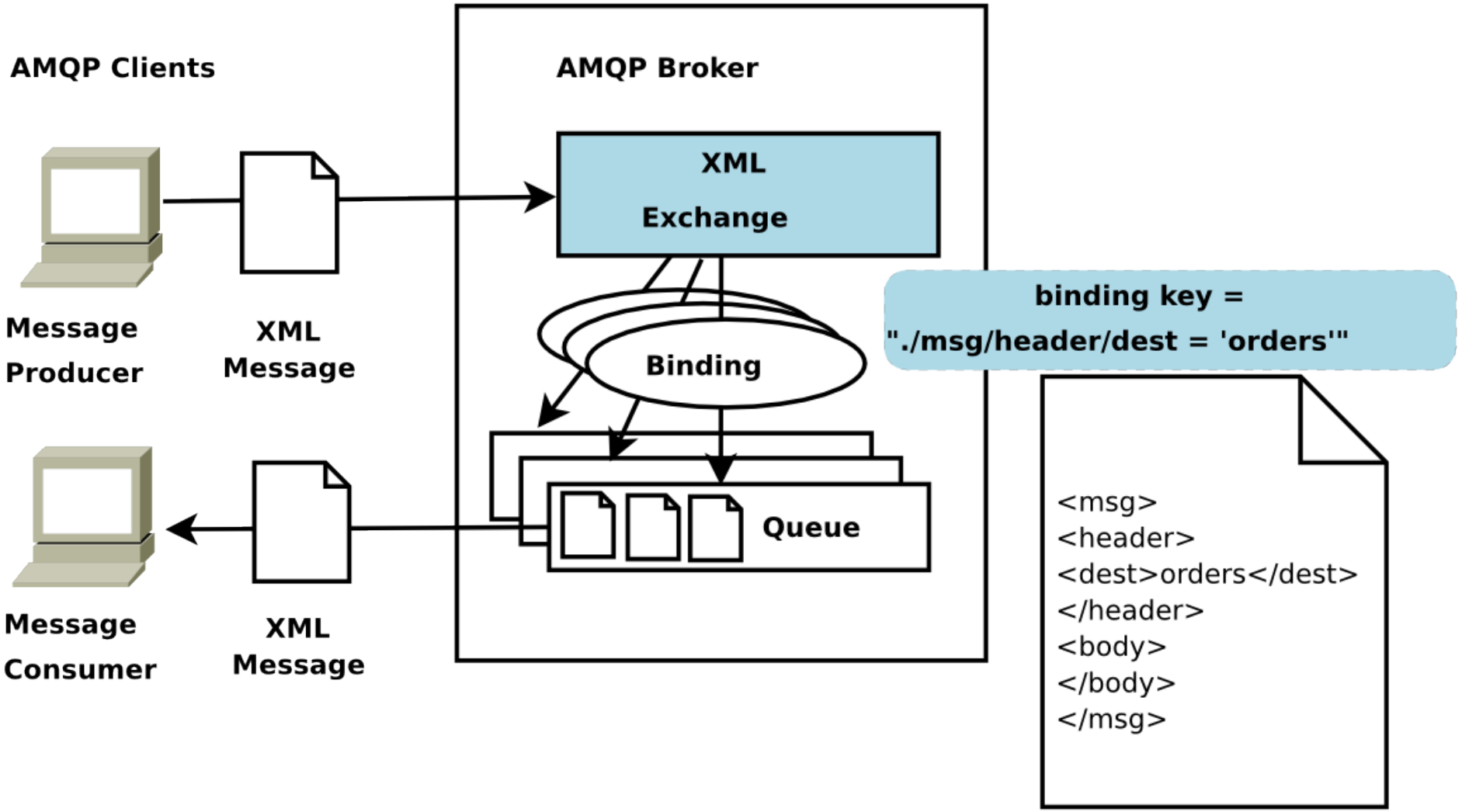
Topic Exchange: Subscriber

```
// Exclusive autodelete queue using Session ID as name

std::string myQueue=session.getId().getName();
  session.queueDeclare(arg::queue=myQueue,
      arg::exclusive=true,
      arg::autoDelete=true);

session.exchangeBind(arg::exchange="amq.topic",
      arg::queue=myQueue,
      arg::bindingKey="usa.*");

// Create a listener and subscribe it to the queue.

SubscriptionManager subscriptions(session);
  Listener listener(subscriptions);
  subscriptions.subscribe(listener, myQueue);

subscriptions.run();
```

Topic Exchange: Publisher

```
Message message;

message.setData("Glaciers have returned to Italy and Texas!");

// Send the same message wherever relevant

message.getDeliveryProperties().setRoutingKey("usa.news");
  session.messageTransfer(arg::content=message,
                arg::destination="amq.topic");

message.getDeliveryProperties().setRoutingKey("usa.weather");
  session.messageTransfer(arg::content=message,
                arg::destination="amq.topic");

message.getDeliveryProperties().setRoutingKey("europe.news");
  session.messageTransfer(arg::content=message,
                arg::destination="amq.topic");

message.getDeliveryProperties().setRoutingKey("europe.weather");
  session.messageTransfer(arg::content=message,
                arg::destination="amq.topic");
```

XML Exchange – a Custom Exchange



XML Content Routing with the XML Exchange

```
# XML Exchange is a custom exchange – must be declared

session.exchangeDeclare(arg::exchange="xml", arg::type="xml");

session.queueDeclare(arg::queue="message_queue");

# XML Bindings are written in XQuery
  FieldTable binding;

binding.setString("xquery", "./msg/header/dest = 'orders'");

session.exchangeBind(arg::exchange="xml",
      arg::queue="message_queue",
      arg::bindingKey="content_feed",
      arg::arguments=binding);
```

Message Flow

Drinking from the firehose - how many messages or bytes can your client accept?

SubscriptionManager sends infinite messages / bytes by default.

SubscriptionManager.setFlowControl() sets limits on what should be sent to a client - “credit”

- How many messages?
- How many bytes?

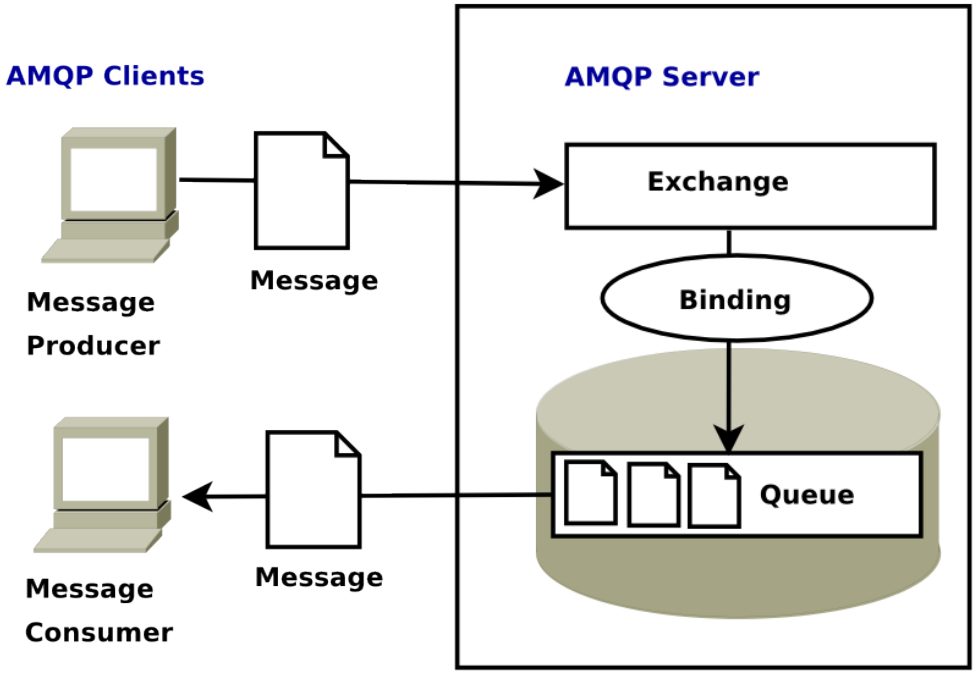
Ask for more credit when you want more

If “window=true”, acknowledging messages allocates credit automatically

Message Flow

```
SubscriptionManager subscriptions(session);
  Listener listener(subscriptions);
  subscriptions.subscribe(listener, myQueue);

// Accept up to 100 messages, 10,000 bytes, enable windowing
subscriptions.setFlowControl("myQueue", 100, 10000, true);

subscriptions.run();
```

Guaranteed Delivery



- Durable Queues are maintained on disk
- Persistent messages are written to journal before delivery
- Sent message is written to journal before ack is sent to Message Producer
- Message is deleted from the queue after Message Consumer acknowledges

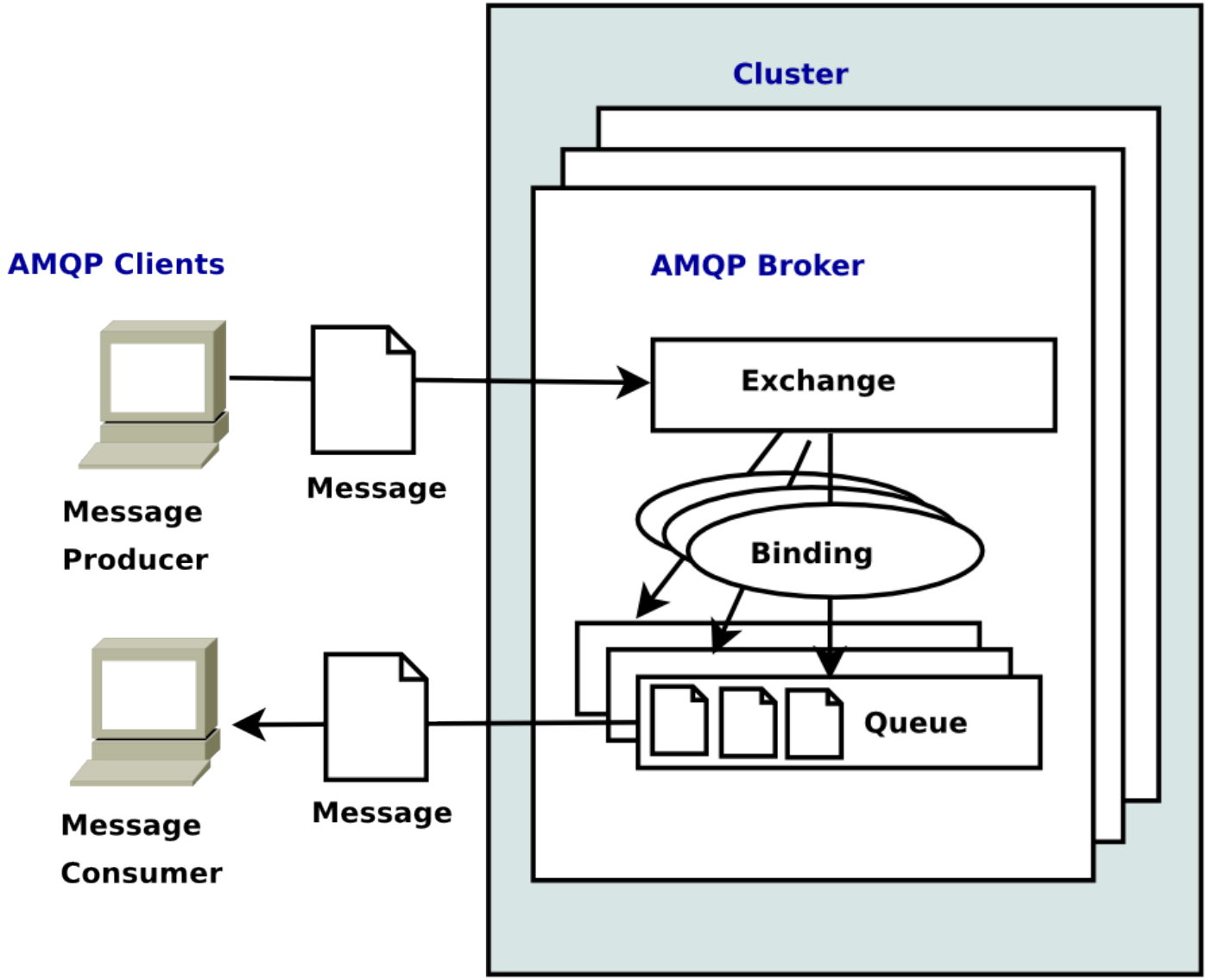
Guaranteed Delivery

```
// Declare the queue to be durable
session.queueDeclare(arg::queue=name, arg::durable=true);
// Choose PERSISTENT deliver mode for message
message.getDeliveryProperties().setDeliveryMode(PERSISTENT);
```

Transactions

- Transaction semantics cover enqueues and dequeues
  - session.txSelect();
  - session.txCommit();
  - session.txRollback();
- Publishing messages in a transaction
  - Commit: messages are placed on queues.
  - Rollback: messages are discarded.
- Consuming messages in a transaction
  - Commit: acked messages are removed from the queue
  - Rollback: message acks are discarded
  - Rollback does not release message - must explicitly release to return it to the queue

High Availability Messaging Clusters



High Availability Messaging Clusters

- Goal: High Availability
  - Any broker can continue work for a failed broker
  - Does not increase throughput or performance
- Built on AIS Closed Process Groups (CPG)
- State is replicated to all brokers in a cluster
  - Messages
  - Queues
  - Exchanges
- Use with Red Hat Clustering Services (RHCS)
  - Eliminates “split-brain” (two independent subclusters due to network failure)

Client Failover in Messaging Clusters

- Connection failure is detected when broker is crashes or is killed
- If heartbeat is enabled, failure is also detected when broker hangs, the machine running the broker goes down, or the network connection to the broker fails
- In-doubt messages
  - Messages that have been sent by the client, but not yet acknowledged as delivered
  - Messages that have been read by the client, but not acknowledged

Client Failover in Messaging Clusters (C++)

FailoverManager class supports reconnection

`MessageReplayTracker` class supports replay of in-doubt messages

Heartbeat is configured with connection settings

Client Failover in Messaging Clusters (Java)

Set heartbeat in the connection URL

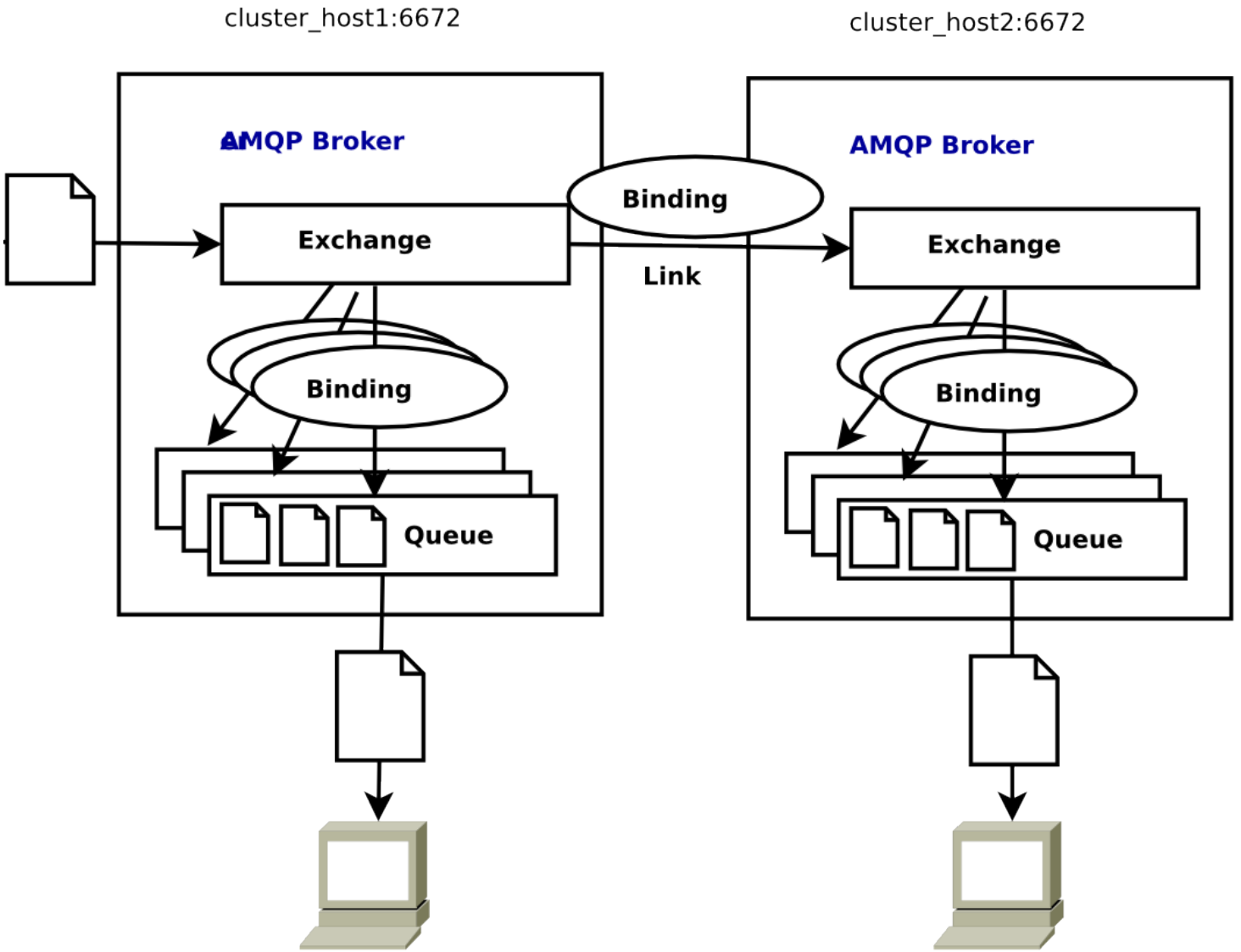
```
connectionfactory.qpidConnectionfactory =
  amqp://guest:guest@clientid/test?brokerlist=&#39;tcp://localhost:
  5672&#39;,idle_timeout=3
```

The Java JMS client automatically connects to another broker when it receives such an exception.

Handling of in-doubt messages by Java client library

- Any messages that have been sent by the client, but not yet acknowledged as delivered, are resent automatically.
- Any messages that have been read by the client, but not acknowledged, are delivered to the client automatically.

Federation


cluster_host1:6672
cluster_host2:6672
AMQP Broker
AMQP Broker
Binding
Exchange
Link
Exchange
Binding
Binding
Queue
Queue

Federation

- Large messaging networks can be built using multiple brokers, using tools or API
- Each destination broker is a client of the source broker
- Links distribute messages based on application architecture
  - Geography
  - Service Type
  - SLA
  - Reliability of connections
- Federation links can be reconfigured without affecting client code

Summary

- Infrastructure for Messaging
  - AMQP: Advanced Message Queuing Protocol
  - Apache Qpid APIs
  - MRG Messaging
- Programming with MRG Messaging
  - C++, Python, Java JMS
  - Point-to-Point, Publish-Subscribe, Broadcast, Request-Response, XML Content Routing
- Advanced Features
  - Guaranteed Delivery, Transactions, High Availability Messaging Clusters, Federation

QUESTIONS?

TELL US WHAT YOU THINK:
REDHAT.COM/SUMMIT-SURVEY

References

Red Hat MRG Messaging
http://www.redhat.com/mrg/messaging/

Red Hat MRG Messaging Documentation
http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/

Apache Qpid
http://qpid.apache.org

AMQP
http://amqp.org