AMQP/Qpid

JUG 2011

Arnaud Simon
asimon@redhat.com

AMQP: Advanced Message Queuing Protocol

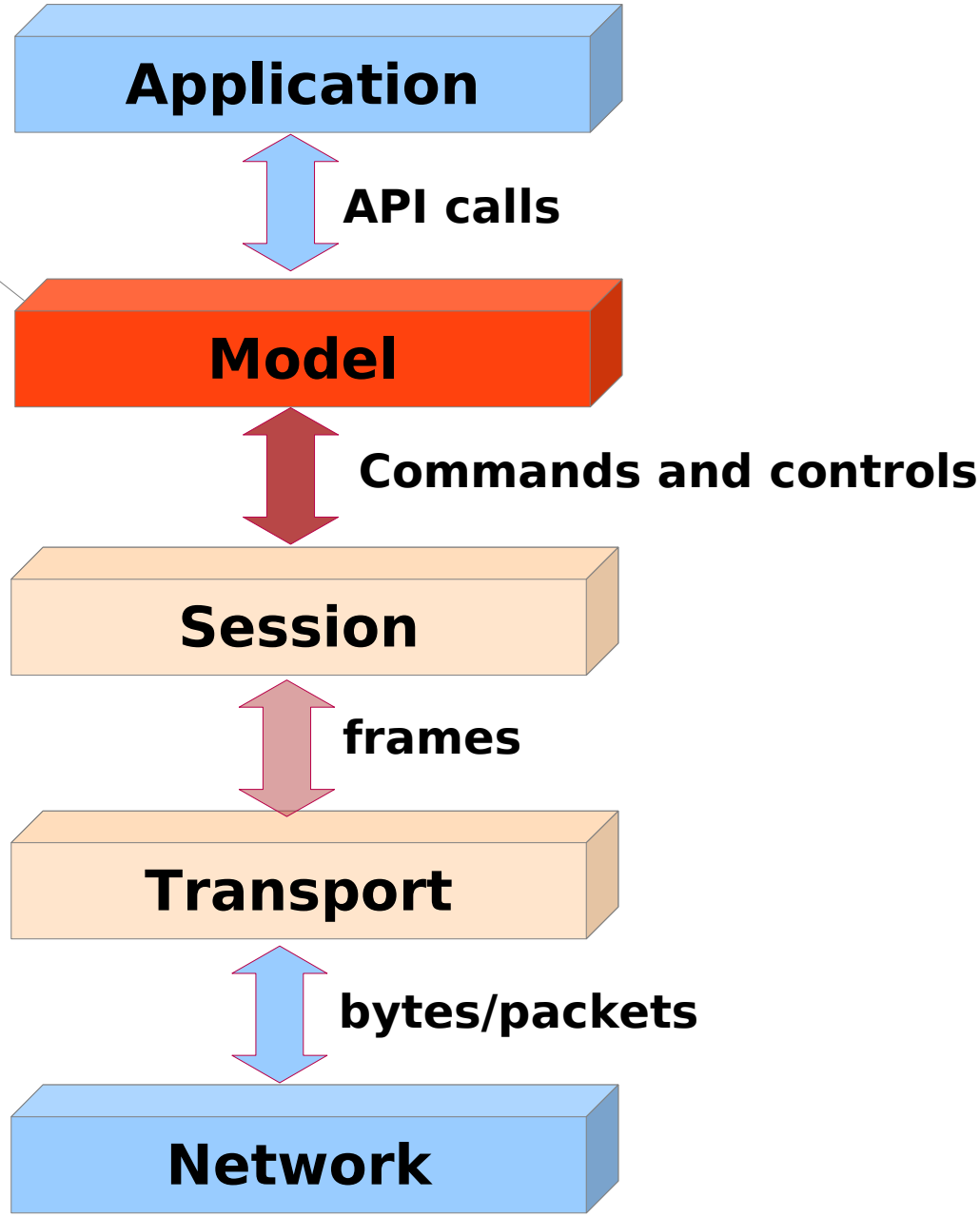
- An open standard for Messaging Middleware
- Pervasive deployment:
  - full interoperability
    - Across platforms, languages, vendors
    - Drop-in compatible with Java JMS
  - Message exchange semantics
  - Network protocol
- Complete solution for business messaging:
  - High performances, Robust, available, Scalable, Secure,Transacted, secure, resilient, ...
- Created by users and technologists

Who is behind AMQP?

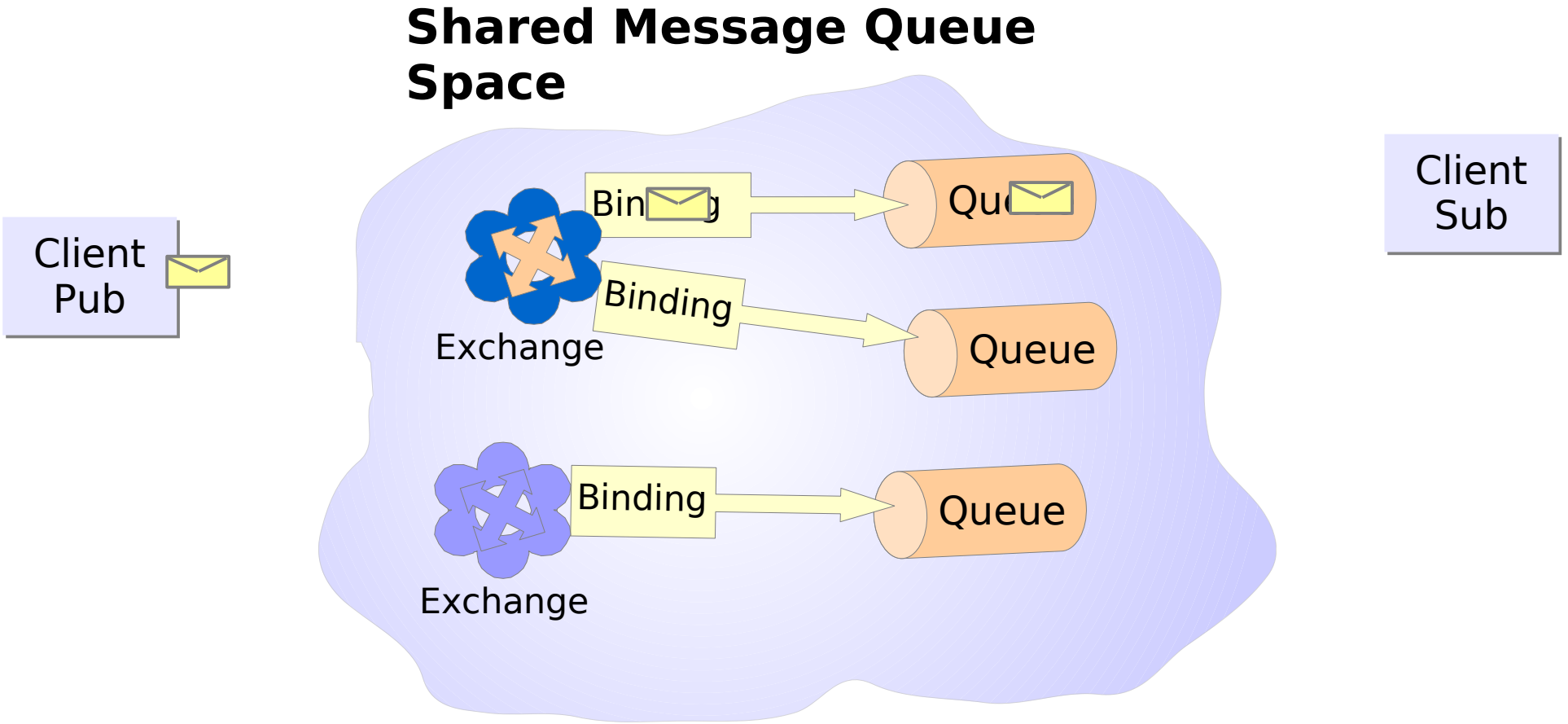
- JPMorgan
- Red Hat
- Deutsche Boerse
- Credit Suisse
- Goldman Sachs
- Cisco
- Iona
- Novell
- Microsoft
- Vmware
- ...

AMQP Layers

Queues, Exchanges,
Messages, Transactions,...



AMQP Model

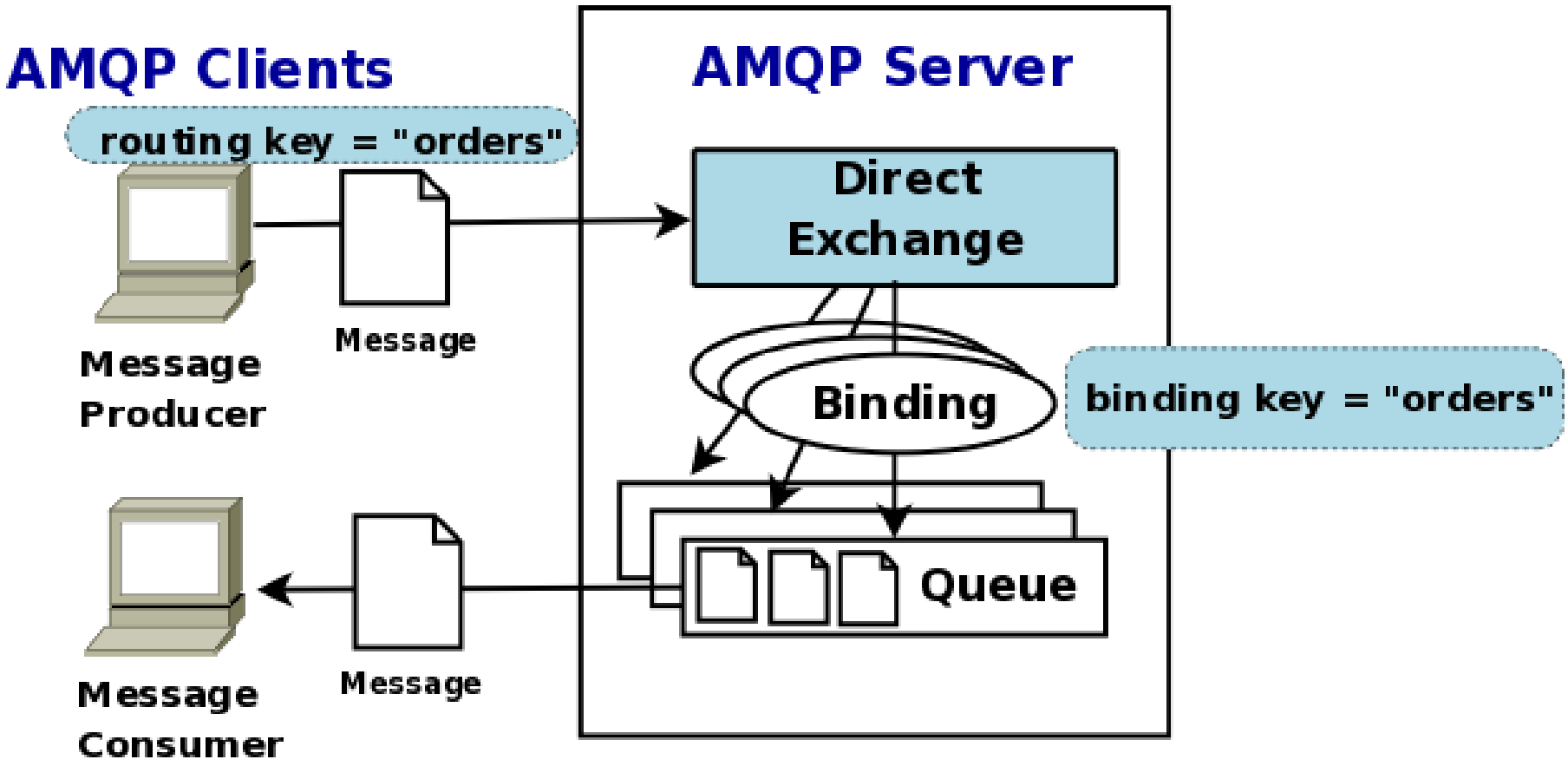

Shared Message Queue Space
Client Pub
Client Sub
Exchange
Binding
Queue
Exchange
Binding
Queue

AMQP Model

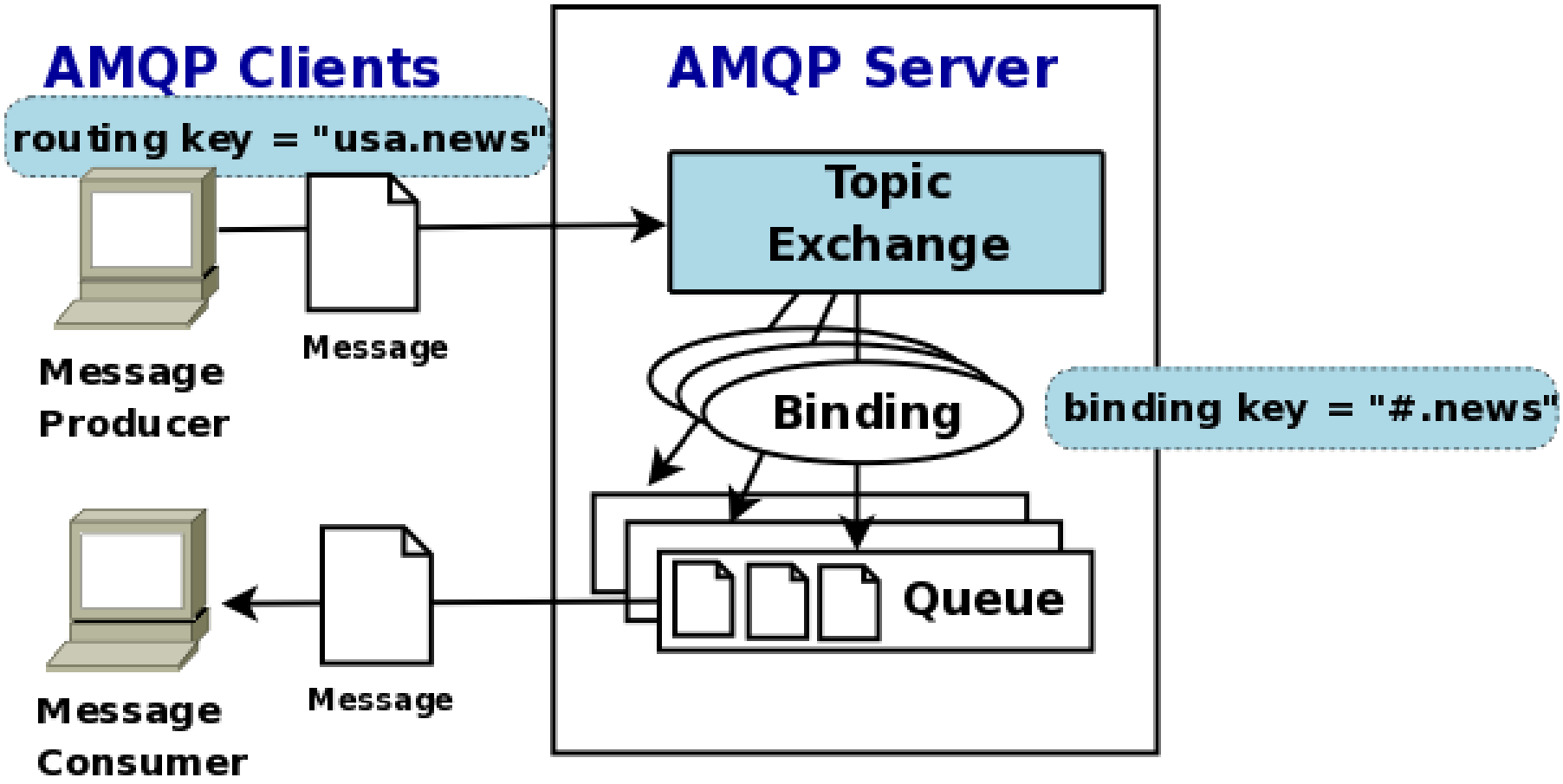
Provides a “Shared Queue Space” that is accessible to all interested applications.

- Message are published/sent to an **Exchange**
- Each message has an associated **Routing Key**
- Exchange forward messages to one or more Message Queues based on the Routing Key
- Consumers get messages from named Message Queues
- Only one consumer can get a given message from each Message Queue

Direct Exchange



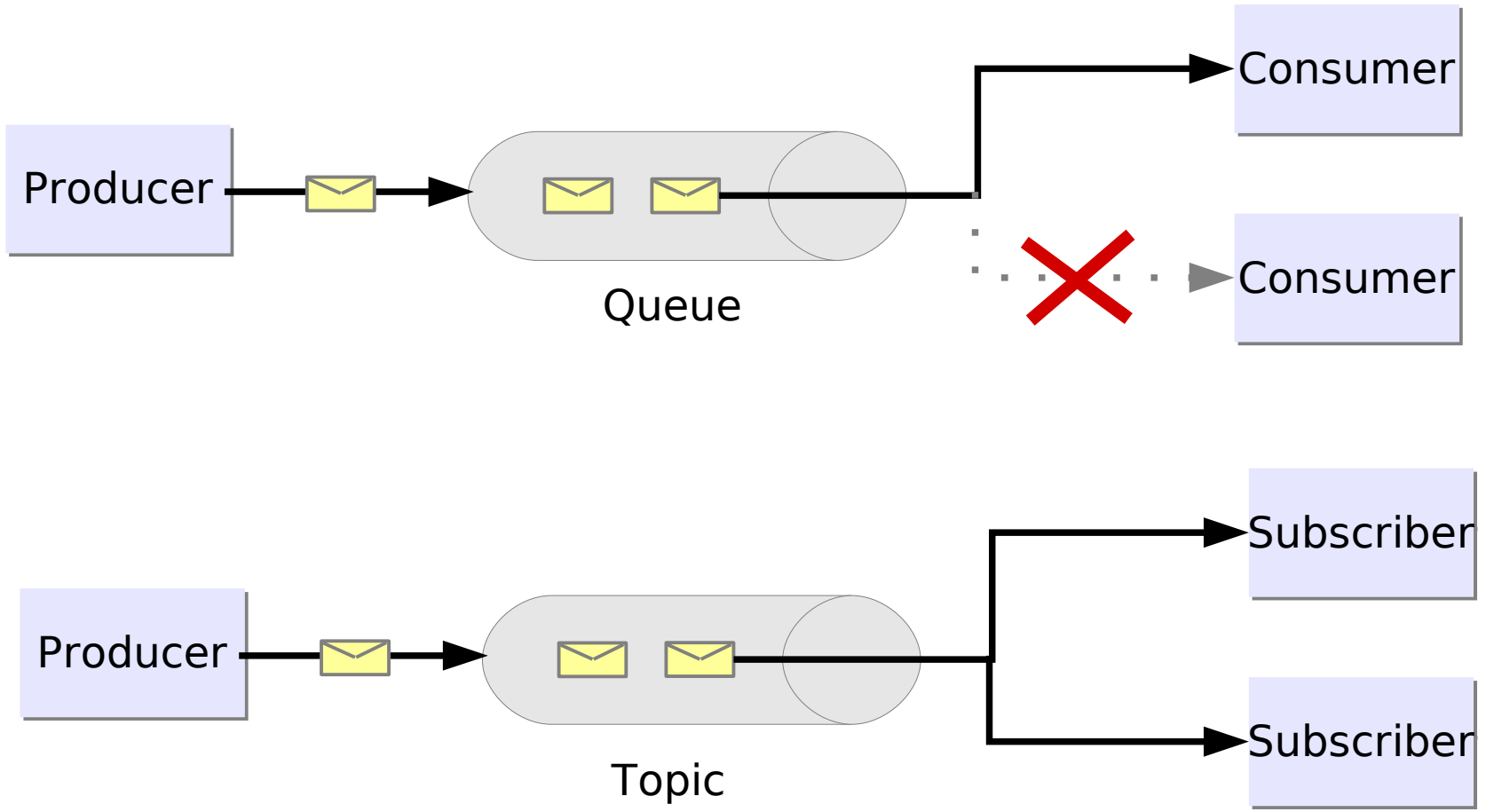
Topic Exchange



Exchange Types

- Direct
- Topic
- Fanout
- Headers
- XML
- Custom
- System

JMS Model


Producer
Queue
Consumer
Consumer
Producer
Topic
Subscriber
Subscriber

Mapping AMQP to JMS

- JMS Queue
  - Exchange type = Direct
  - One single queue
    - Routing key = Queue name = biding key
  - Queue is Public
    - All consumers consume from the same queue
  - Queue is Durable
    - Restored and Kept even if there isn't any consumer

Mapping AMQP to JMS

- JMS Topic
  - Exchange type = Topic
  - One queue per consumer
    - Routing key = topic name
    - Binding key = wildcard
  - Queues are Private and
    - Volatile = standard subscriber
    - Durable = durable subscriber

What about JMS Queue browsing ?

- JMS defines the notion of queue browsing
  - Messages are accessed but not consumed
- Can we push the concept even further?
  - The consumer should be able to decide whether
    - The message is of interest
    - Potentially consumed it

Transfer of Responsibility

- **No-acquire mode:**
  - Only data is transfered, **NOT** responsibility
  - No exclusive access to process the message
    - Another client may see, acquire and consume
  - Need to explicitly acquire before processing
- **Pre-acquire mode:**
  - Both data AND responsibility are transferred
  - Exclusive access to process the message
  - No other client can see the message
  - Can release to relinquish responsibility

Accept Mode

- **Explicit**
  - Successful transfer is signaled by semantic ack
- **None**
  - Successful transfer upon acquisition
- **Release**
  - Relinquish responsibility for processing message that can be safely delivered to other clients
- **Reject**
  - Indicates a problem with processing a message (DLQ of the message)

JMS Queue Browser

- JMS Queue browser
  - no-acquire - accept-mode = explicit

    send; look into message
  - More than JMS Queue browsing: consume messages of interest

    send; look into message; acquire; ack
- JMS Queue with guaranty delivery
  - pre-acquire - accept-mode = explicit
- Fast unreliable JMS Queue
  - pre-acquire - accept-mode = none

Flow control issue

- **How can we handle fast producers?**
  - Messages are pilling up on the server
  - Memory exhaustion
- **How can we speed up message delivery?**
  - Messages are sent one after each other
  - Increased latency
- **How can we handle small memory footprint?**
  - Cannot consume and or store big messages

Message Flow

- Credit Based
  - Sender maintains credit balance with recipient
  - Credit Balance consist of a
    - Message Count
    - Byte Count
  - When a Message is sent both counts are decremented
  - When either value is zero no more messages are sent until further credit is received from peer.
  - If byte count is insufficient no partial messages can be sent

Message Flow

- Window
  - Message acknowledgment implicitly grants:
    - a single unit of message credit
    - size of message in byte credits
  - Controls the window of un-acked messages

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'
```

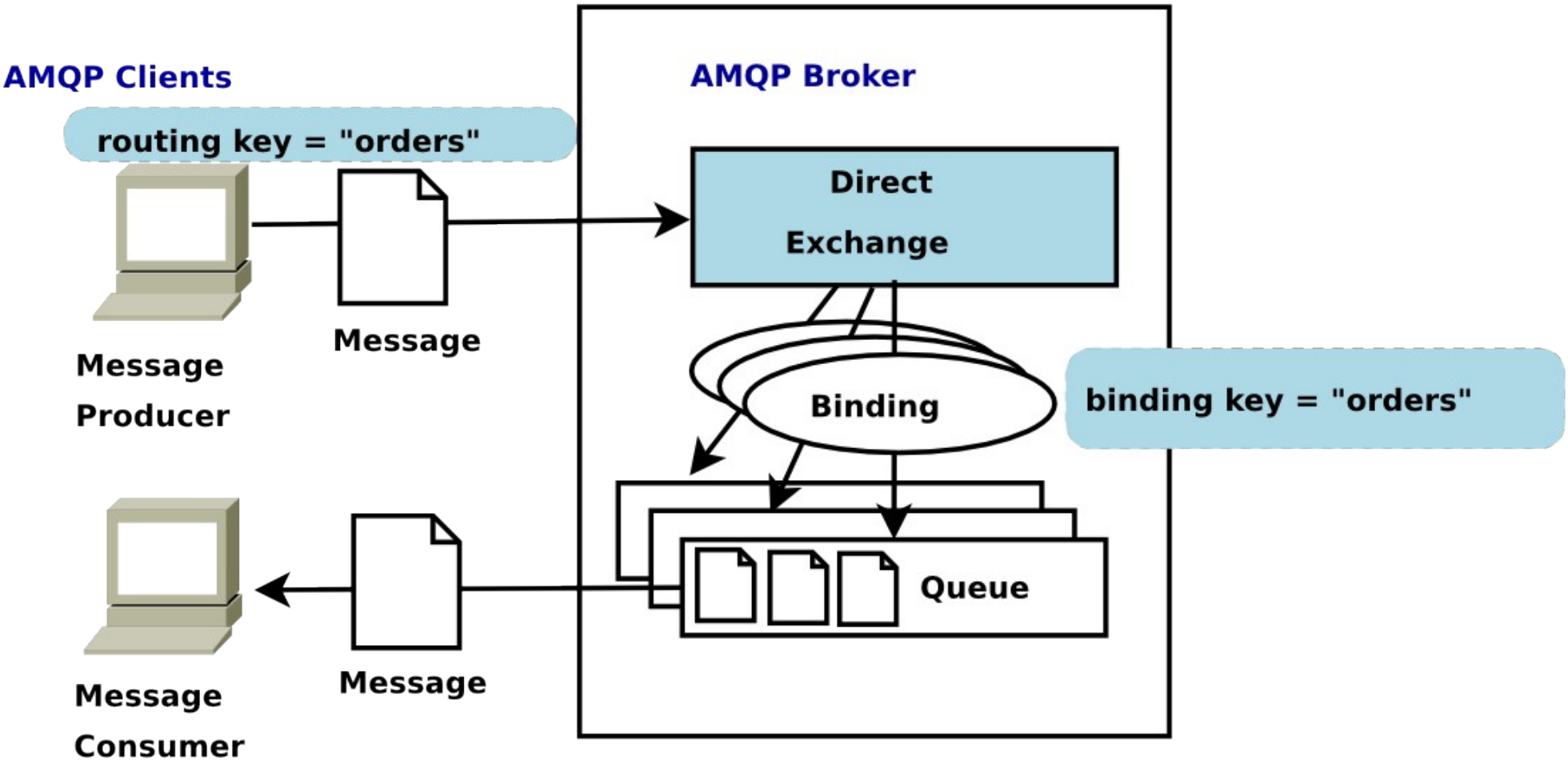
Using JNDI to create Java JMS Session, Connection, Destination

```
// 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

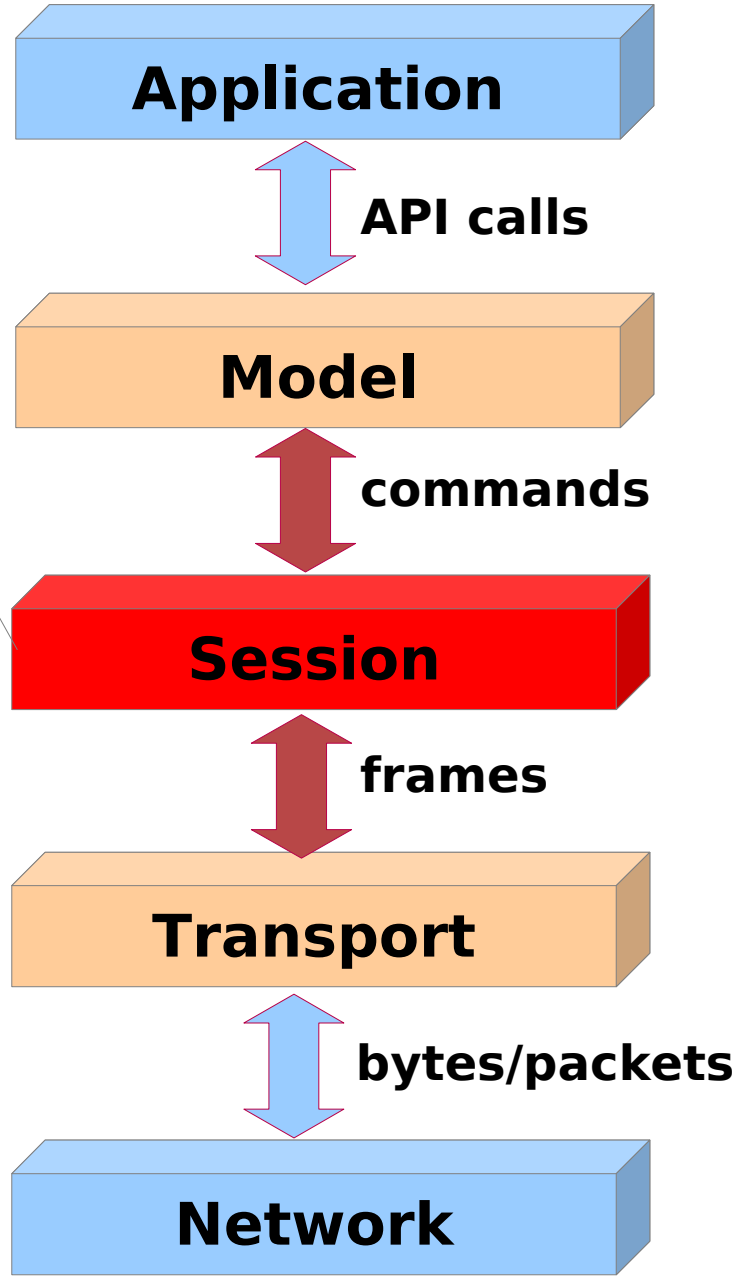
```
// Create a Listener, Derived from MessageListener

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

```
// 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();
```

AMQP Layers

Commands, Controls
Exceptions, Confirmation
Completion, Replay
Synchronization



Reliability issues

- How can we handle network failures?
  - Message transfer can be interrupted
  - Clients needs to reconnect
- Example

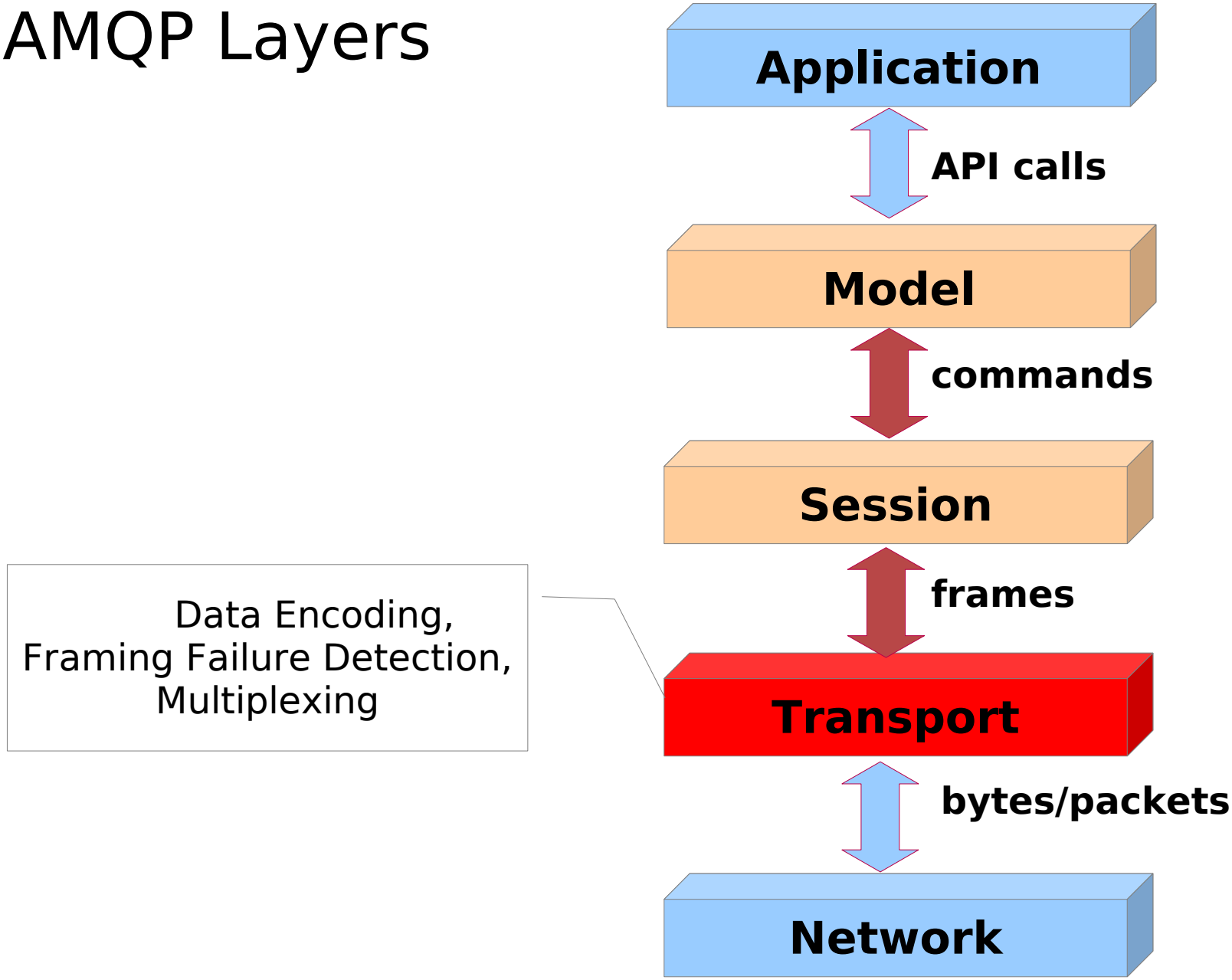
A train sends messages to the Railway company HQ

  - The train is moving!
  - The network nature is changing
    - Satellite, GSM, Etc.
- In cluster mode how can we transparently switch nodes

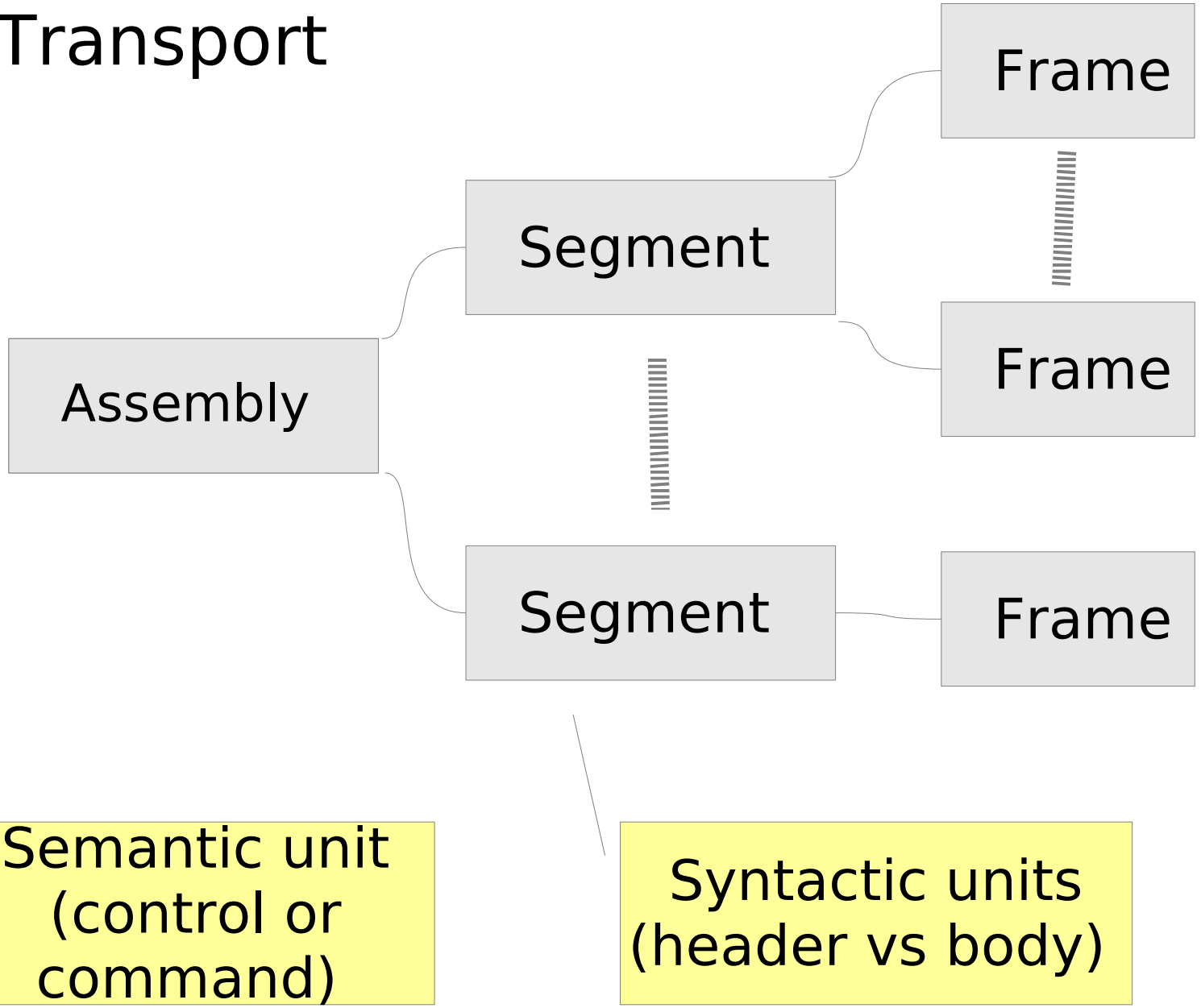
Session

- Sequential numbering scheme with rollover to identify each command uniquely within a session
- Session state
  - a replay buffer of full or partial commands which a peer does not yet have confirmation its partner has received
  - an idempotency barrier - a set of commands identifier which the peer knows that it has received but cannot be sure that its partner will not attempt to re-send.

AMQP Layers

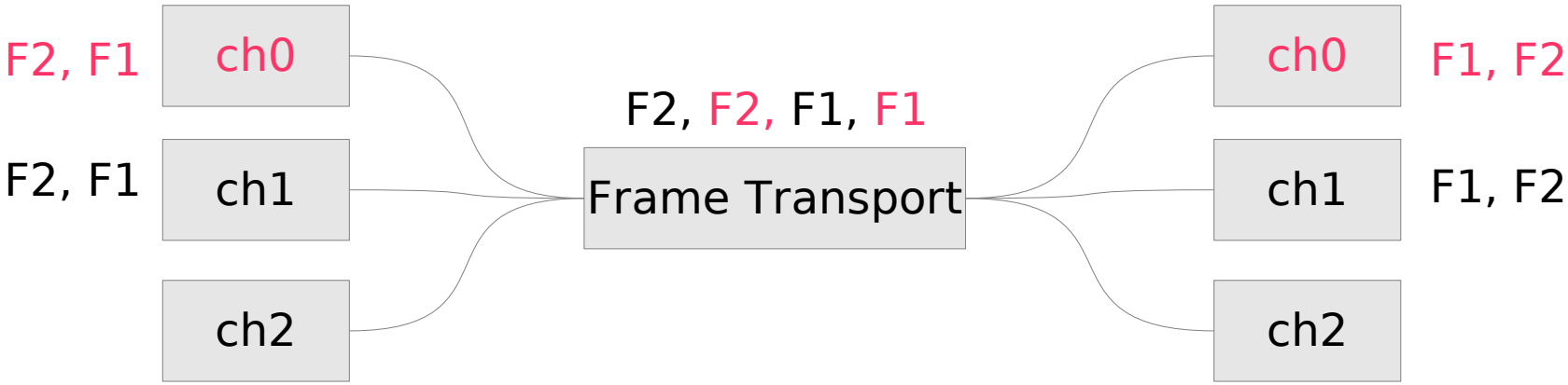

Application
API calls
Model
commands
Session
frames
Transport
bytes/packets
Network
Data Encoding,
Framing Failure Detection,
Multiplexing

Transport


Frame
Segment
Frame
Assembly
Segment
Frame
Semantic unit (control or command)
Syntactic units (header vs body)

Frames

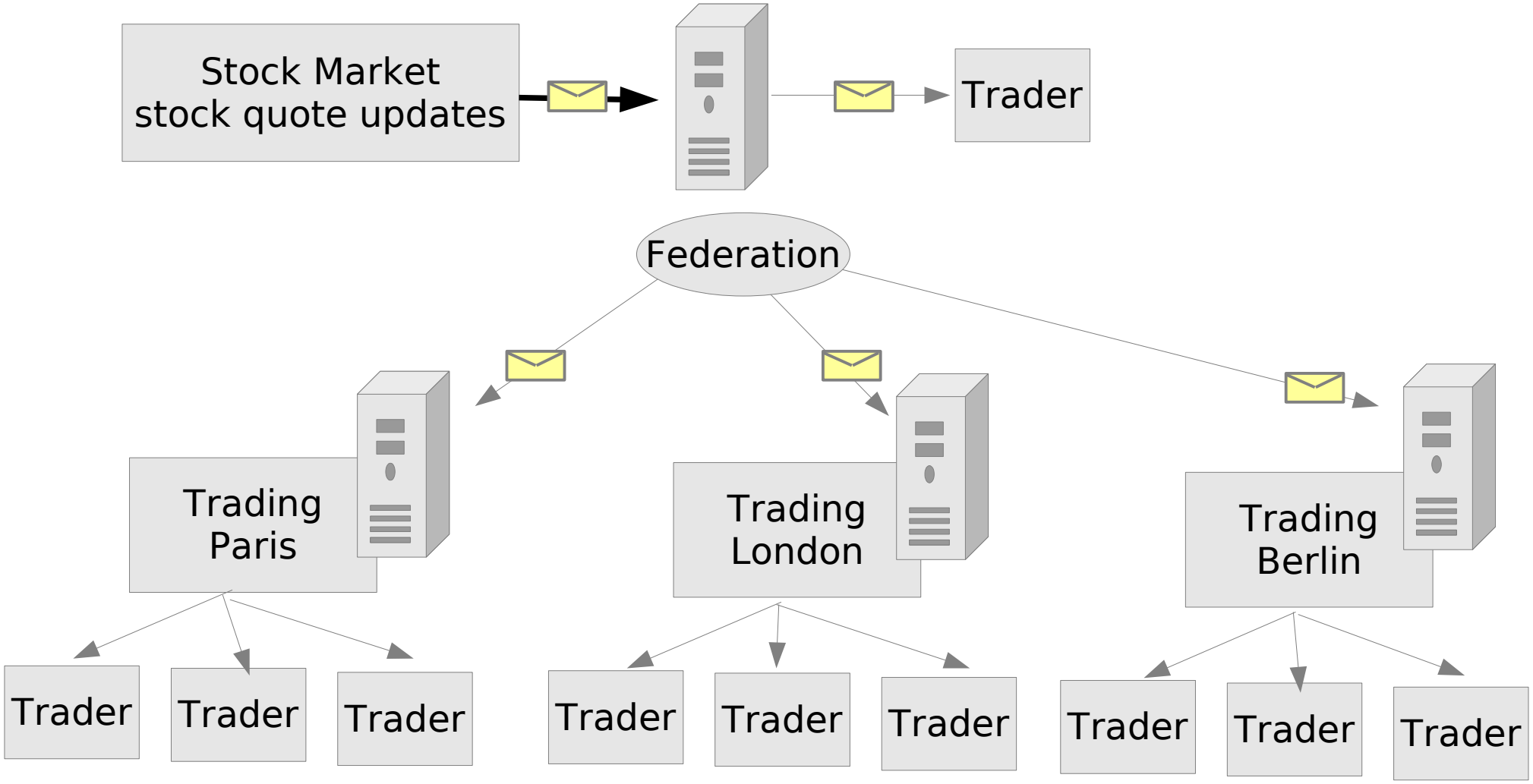
- Header:
  - Channel
    - divides a single frame transport into distinct channels (sessions)



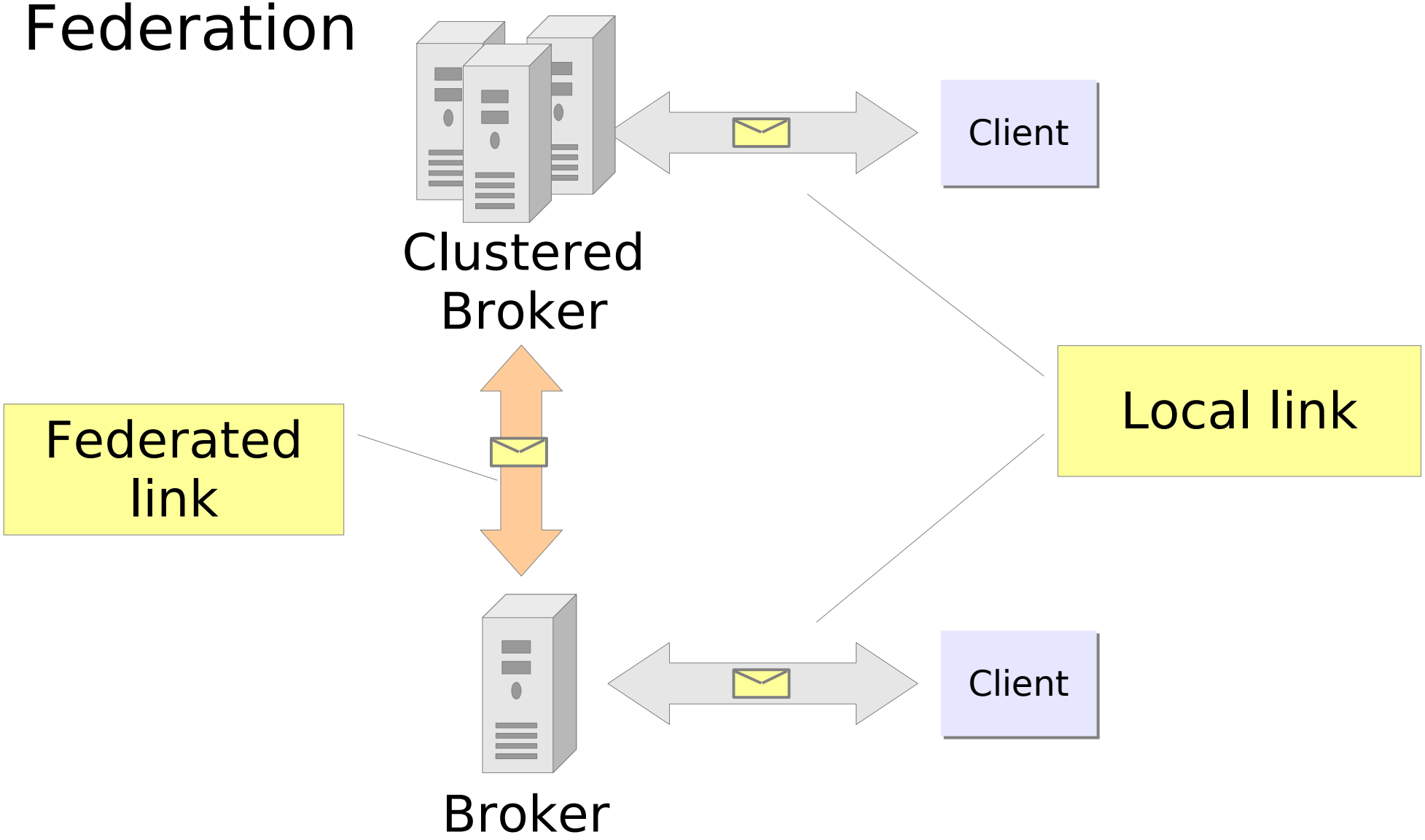
Federation

- Federation
  - Joining of multiple brokers together in a large functioning network
- Clustering
  - Several brokers deployed that act as a single broker
    - High availability
    - Performance

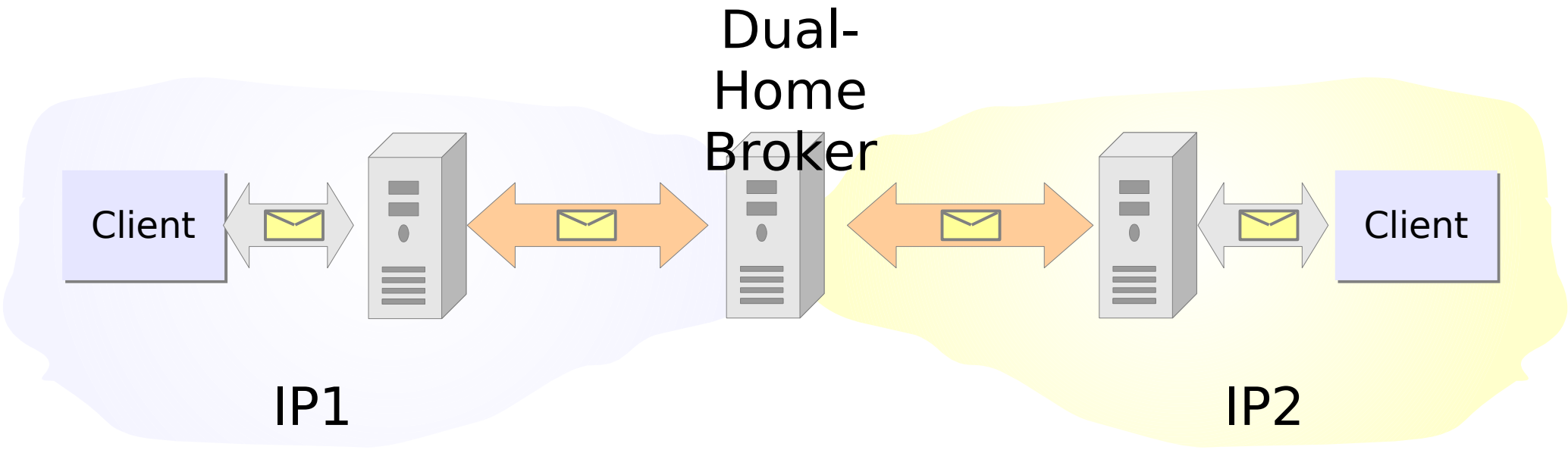
Federation


Stock Market
stock quote updates
Trader
Federation
Trading
Paris
Trading
London
Trading
Berlin
Trader
Trader
Trader
Trader
Trader
Trader
Trader
Trader
Trader

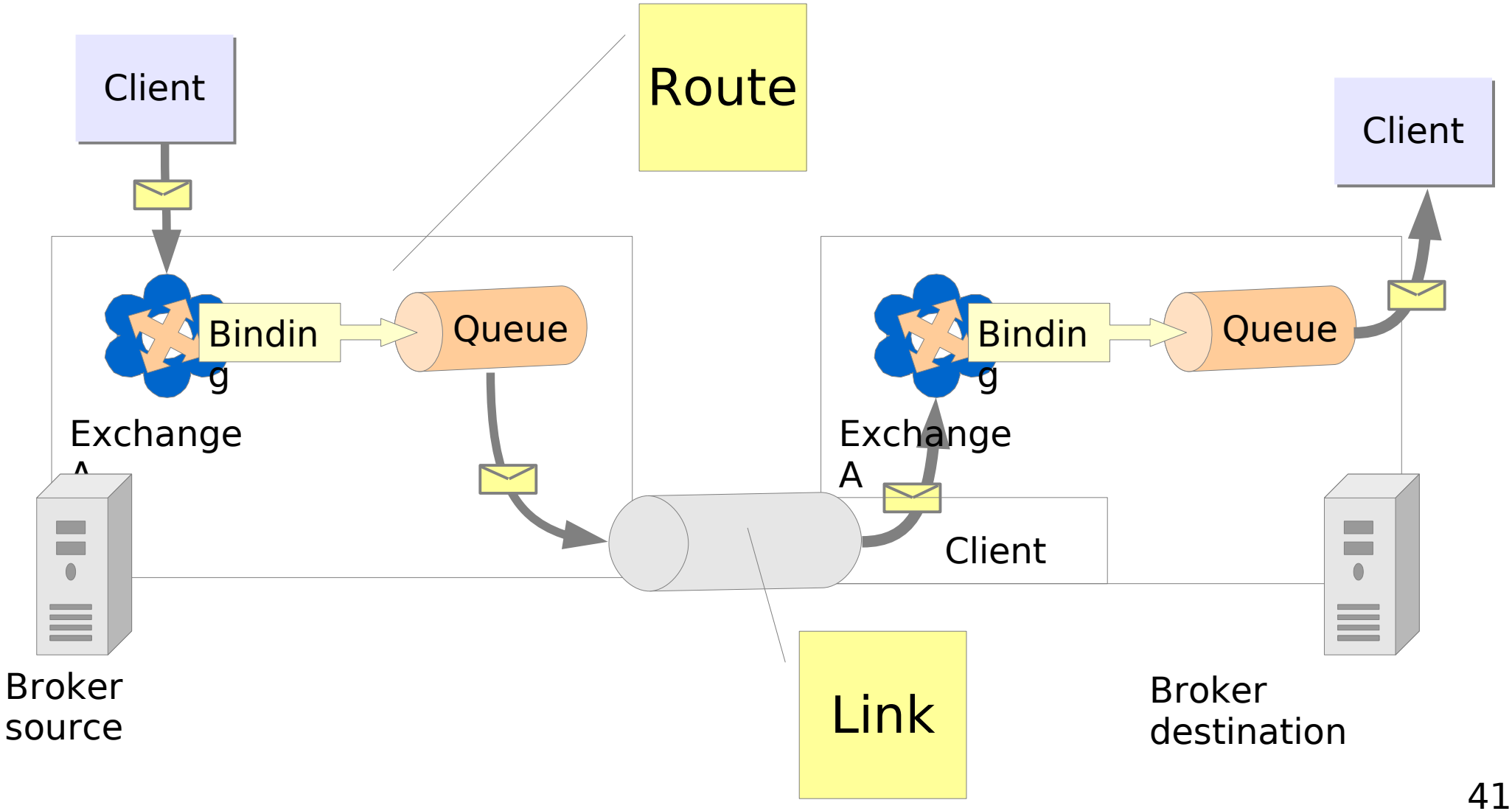
Federation


Clustered
Broker
Client
Local link
Federated
link
Broker
Client

Isolated Networks



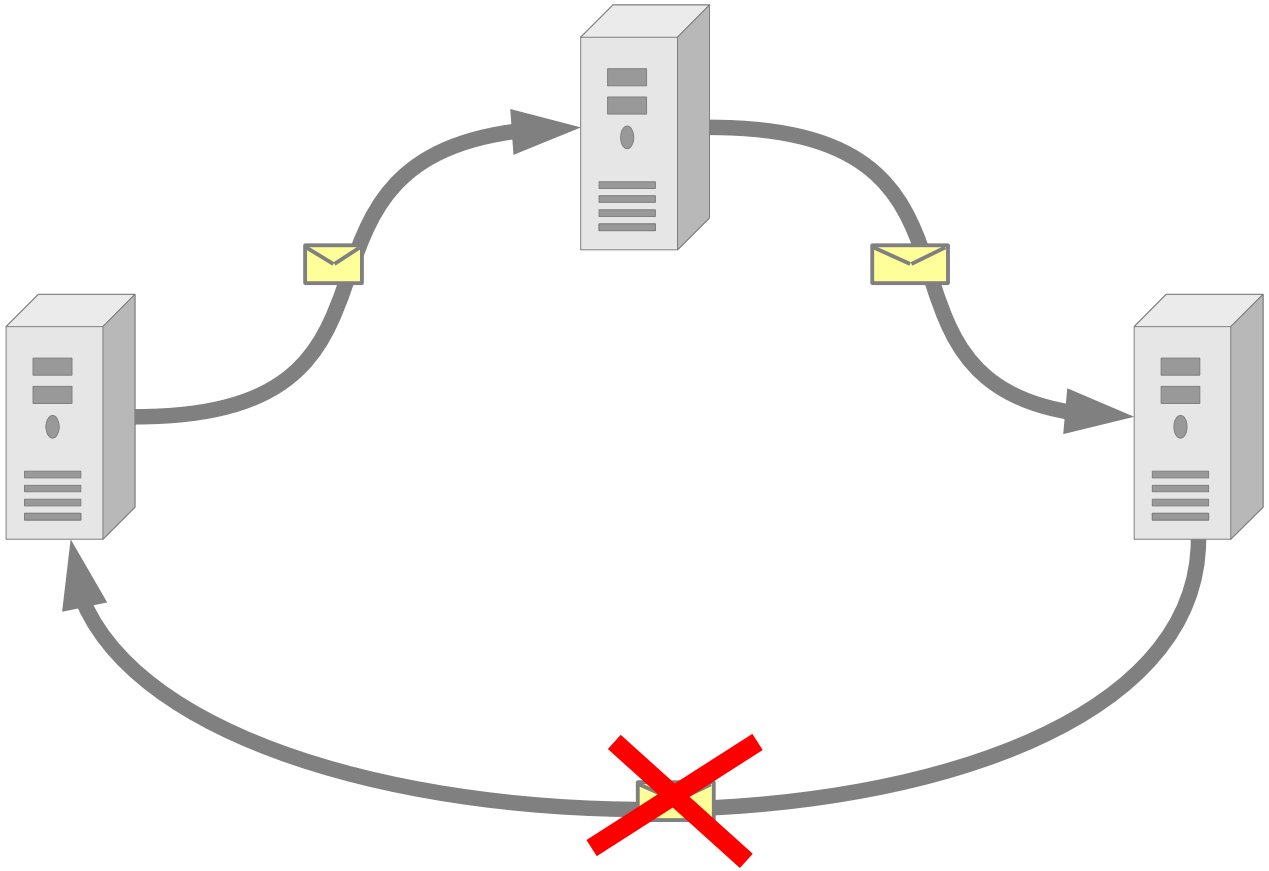
Distributed Exchange



Federation

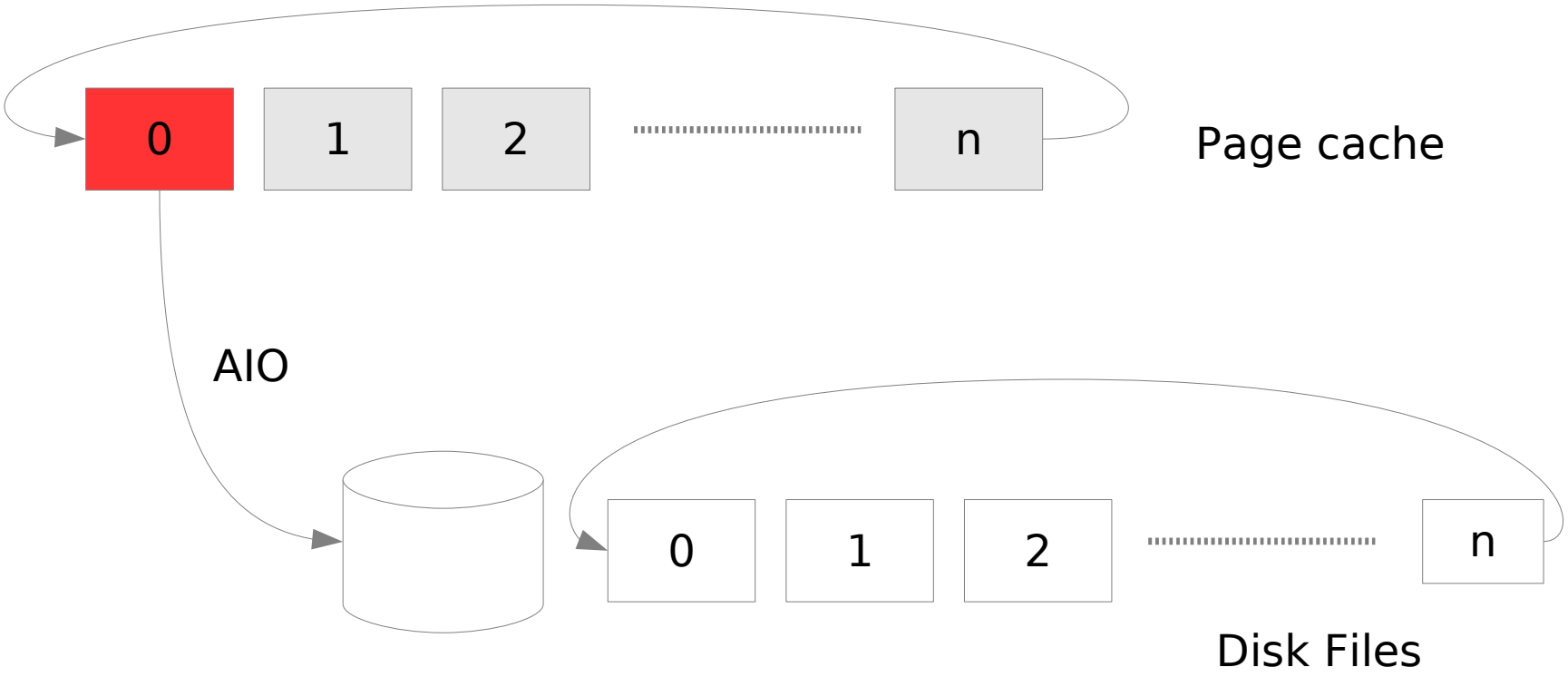
- Link:
  - Source broker
  - Destination broker
- Route:
  - Source broker
  - Destination broker
  - Exchange name
  - Binding key

Federation - cycles



Message Store

- Asynchronous Journal
  - Asynchronous IO
  - O-direct flag (disable buffering)
  - “Circular buffer”



Authentication SASL

- Simple Authentication and Security Layer
  - Decouples authentication mechanisms from AMQP
  - Supports a rich set of mechanisms:
    - Anonymous, Plain, MD5 challenge/response
    - NTLM for Windows
    - GSSAPI for Kerberos v5
    - EXTERNAL for x.509 Certificate authentication
- Authentication of AMQP client

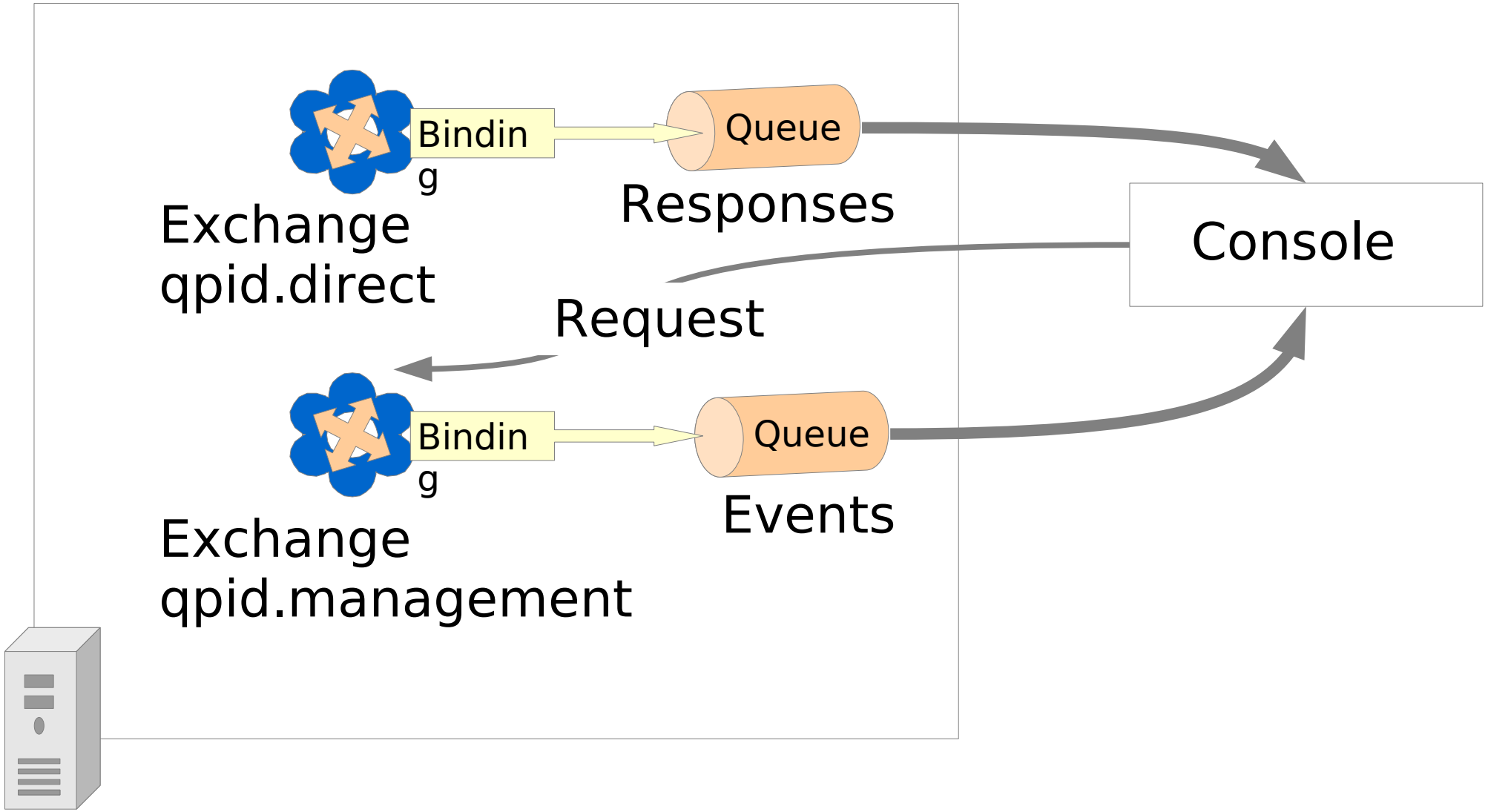
Message-level Security - Signing

- Protects data integrity end-to-end
- Signature covers Header and Body segments concatenated
- AMQP 1.0
  - Add a new footer segment to contain the signature of the concatenation of the header and body segments
  - Add a new field to message-properties: signature-control. This is used to identify the signing mechanism used for the message.
  - Add reject codes were added to allow a consumer to reject a message (invalid-signature, missing-signature, untrusted-signature)

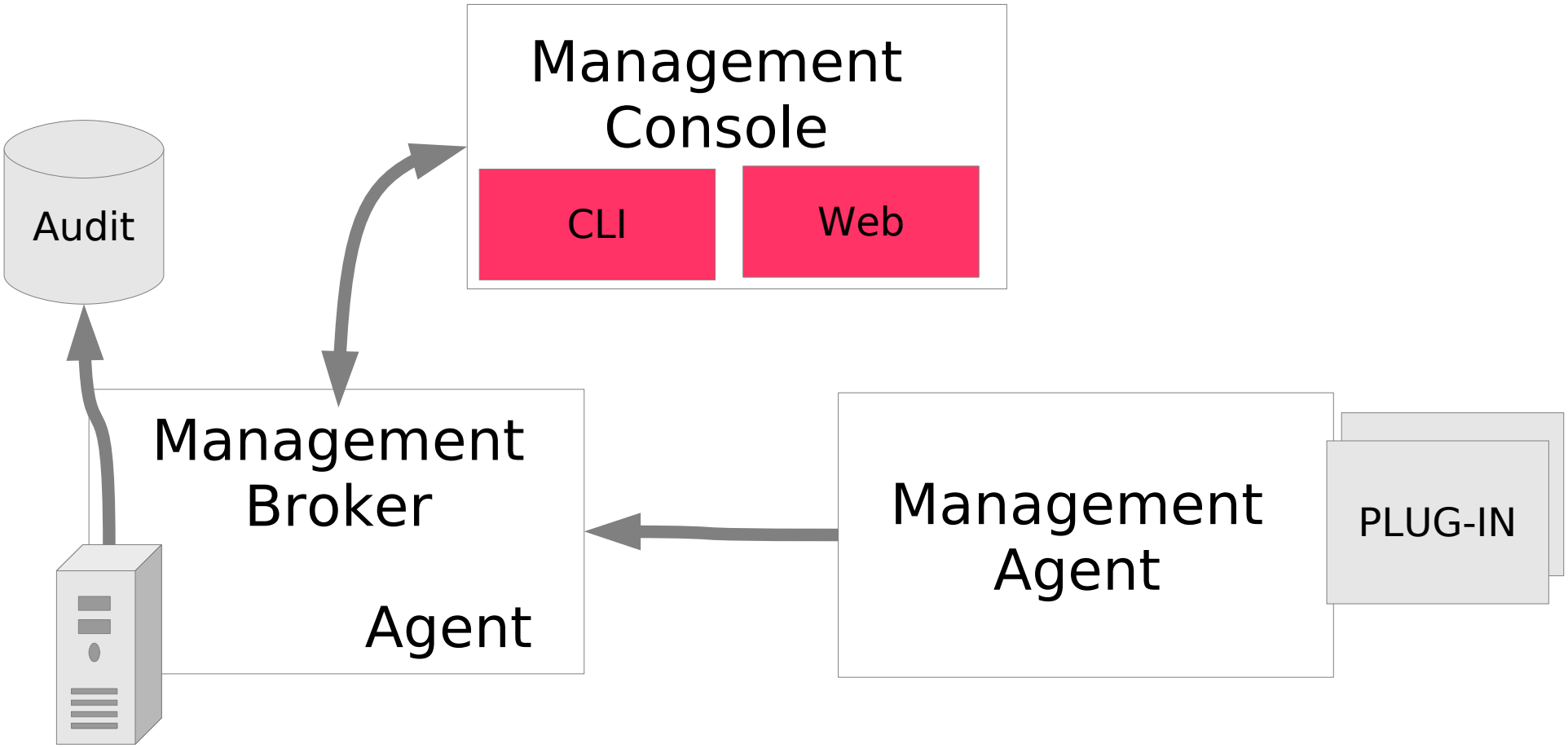
Message-level Security – Encryption

- Encryption may be applied to the entire body segment
  - Broker need not be trusted
- The header segment must not be encrypted:
  - Header contains information needed by the broker
  - Header contains information needed to decrypt the body
- AMQP SP1
  - Add field: encryption-control – used by consumer to identify the algorithm and key to decrypt the body (S-MIME, other mechanisms)
- Key exchange is outside the scope of AMQP

Management


Bindin
g
Queue
Responses
Exchange
qpid.direct
Console
Request
Bindin
g
Queue
Events
Exchange
qpid.management

Management


Management
Console
CLI
Web
Audit
Management
Broker
Agent
Management
Agent
PLUG-IN

Questions?

Sorting out the terms

Term	Who	What
AMQP	AMQP Working Group	Advanced Message Queuing Protocol
Qpid	ASF	Implementation of AMQP
BdbStore	Red Hat	RedHat's durable message store (QPID plug-in)
MRG	Red Hat	Product: Messaging, Real Time, Grid