TOWARDS UNIFIED MESSAGING

Rafael Schloming (Red Hat)

Gordon Sim (Red Hat)

František Řezníček (Red Hat)

Friday 17th February, 2012

Agenda

- The quiz
- Asynchronous Messaging: What is it? Why use it?
- AMQP: The missing protocol?
- Apache Qpid: Open Source AMQP Messaging
- Red Hat Enterprise MRG
- Join us!
- Discussion

The quiz for professionals

- What do have in common following dates? What happened those days?
    - 31st August 2011
    - 7th October 2011

The quiz for dummies

- What is the other term commonly used for AMQP software server (qpidd)?

AMQP 1.0

An **open** and **pervasive messaging infrastructure** offering **rich capabilities** for developing distributed systems

Asynchronous Messaging: What is it?

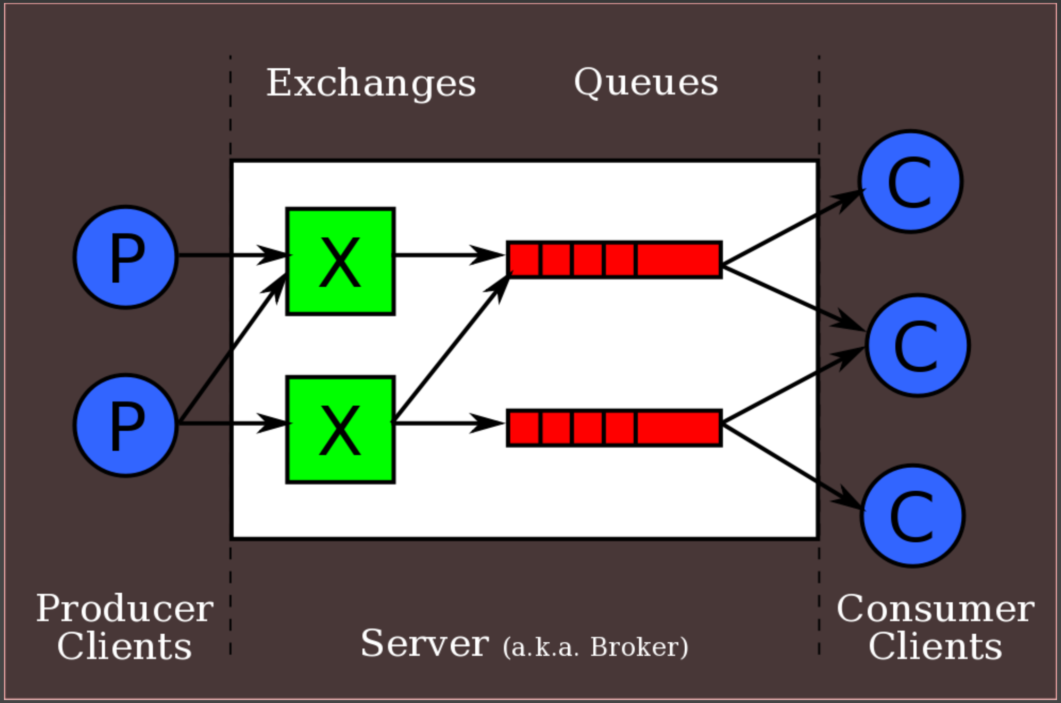
- An abstraction for communication that is:
  - **Higher level** than sockets
  - **More flexible** than RPC or distributed objects
  - **More general** than HTTP
  - **More complete** than SMTP or XMPP

Asynchronous Messaging: Why use it?

- **Natural fit for events, notifications and coordination**
- Enables intermediation
- Decouples communicants “in space and time”
- Separation of concerns
  - Avoid implementing mechanics; retain control over policy

Asynchronous Messaging: The Fundamentals

- **Message as the 'unit' of communication**
- **Transfer of responsibility**
- **Flow control**
- **Addressing**

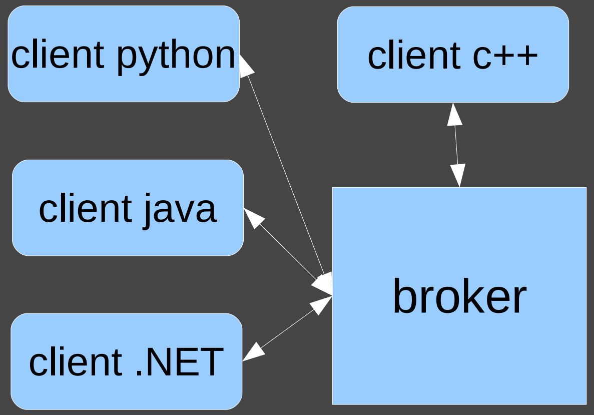


Asynchronous Messaging: The Fundamentals

- **Message as the 'unit' of communication**
  - Payload (body data)
  - Standard annotations (i.e. headers or properties)
    - message identity & correlation
    - description of message (e.g. subject) and payload (e.g. content-type)
    - reply-to address, identity of publisher, time to live etc
  - Application defined annotations
- Transfer of responsibility
- Flow control
- Addressing

Asynchronous Messaging: The Fundamentals

- Message as the 'unit' of communication
- **Transfer of responsibility**
  - Acknowledgements & 'in-doubt' messages
  - Replay, de-duplication & idempotence
  - Delivery guarantees: at-least-once, at-most-once, exactly-once
- Flow control
- Addressing



Asynchronous Messaging: The Fundamentals

- Message as the 'unit' of communication
- Transfer of responsibility
- **Flow control**
  - Propagates receiver's constraints to sender
  - Sender can transmit knowing receiver is operating within its limits
  - Aids efficiency and reliability.
- Addressing

Asynchronous Messaging: The Fundamentals

- Message as the 'unit' of communication
- Transfer of responsibility
- Flow control
- **Addressing**
  - What messages go where?
    - Address as alias for specific routing path:
      - *sender* ► **Exchange** ► **Binding** ► **Queue** ► *receiver*
      - *sender* ► **Address** ► *receiver*
  - In messaging an address is a *rendezvous point*
  - *Competing* v *non-competing* consumers

Standardizing Asynchronous Messaging

- JMS is not sufficient
  - An API not a protocol
    - Leads to closed systems where all components are from the same vendor
  - Language specific (java)
- Need a wire level protocol to achieve full potential
  - Combine components from different vendors, written in different languages
  - Achieve network effects and create a new ecosystem
- To be universal need to be simple ***and*** flexible

AMQP: Standardized asynchronous messaging

- http://www.amqp.org
- Evolving for past 5 years (driven by actual implementations in real deployments)
- Final 1.0 specification released in October 2011
- Open alternative to JMS and WCF

AMQP: Working group members

Bank of America,

Barclays Bank PLC,

Cisco Systems,

Credit Suisse,

Deutsche Börse Systems,

Goldman Sachs,

HCL Technologies Ltd,

INETCO Systems Limited,

Informatica Corporation,

JPMorgan Chase Bank Inc.,

Microsoft Corporation,

VMware, Inc.,

Novell,

Progress Software,

Rabbit Technologies Ltd.,

Red Hat Inc.,

Software AG,

Solace Systems Inc.,

StormMQ Ltd.,

Tervela Inc.,

TWIST Process Innovations Ltd,

WS02 Inc.

29West Inc.

AMQP 1.0: Failure handling fundamentals

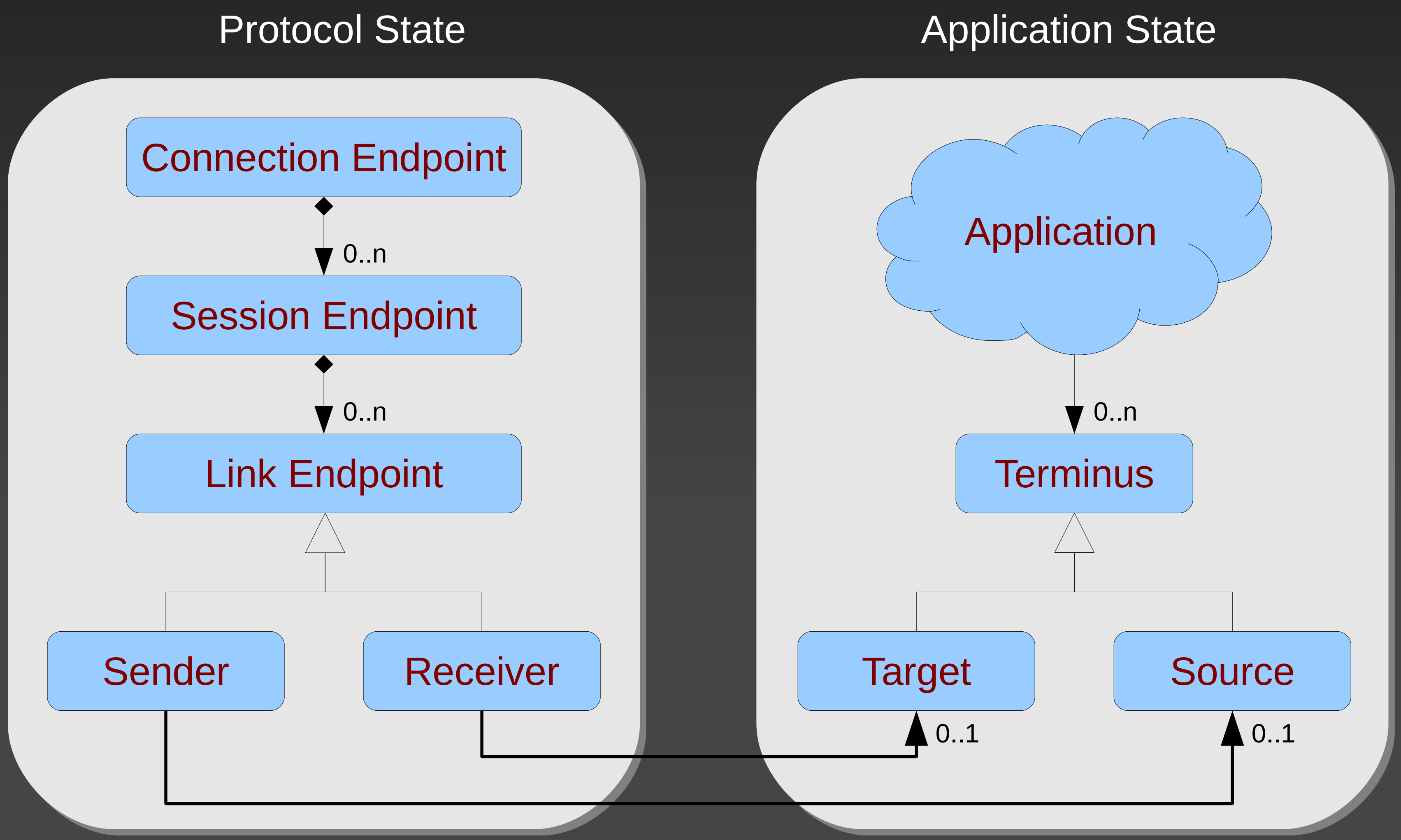
- **Identity**
  - Serial Numbers, UUIDs, DB Keys, Semantic Keys, ...
- **Retry**
  - Deals with possible loss
    - Line noise, congestion, hardware failure
  - Generates duplicates
- **Deduplication**
  - Requires Identity (message property)

AMQP 1.0: Failure handling fundamentals

- **Network protocols make a particular choice**
  - If this doesn't fit your application, you're out of luck.
  - Hence the proliferation of application specific protocols built on TCP ► mess of non-universal protocols

- **AMQP leaves the choice up to the application**
  - The application can focus on domain semantics rather than communication semantics
  - Essential for universal messaging protocol ► flexibility

AMQP 1.0: Endpoint State



AMQP 1.0: Terminology

- **Logical "overlay" network of Nodes & Links**
  - Nodes are points in the AMQP network that can produce, consume, relay, or even transform messages
  - Links are unidirectional paths on which messages can flow between Nodes
- **Physical "underlay" network for active communication**
  - Connections, Sessions, Links
- **Links divided into logical and physical aspects**
  - Terminus vs Link Endpoint

AMQP 1.0: An Overview

- Protocol primitives (performatives)

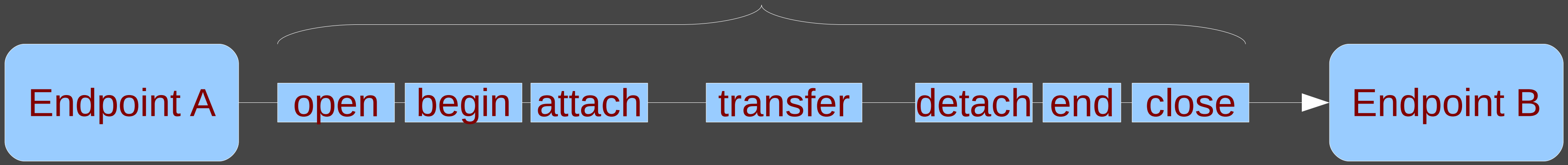
open	connection parameters negotiation	Setup
begin	session start on channel	
attach	attach link to session	
transfer	transfer a message	Message Transfer
disposition	inform peer about delivery changes	
flow	update link state	
detach	detach link from session	Teardown
end	session end	
close	close connection	

AMQP 1.0: Protocol Primitives

- Informational rather than instructional
  - Indicate facts about the endpoint state
- No lock-step dependencies
  - Semantics of incoming and outgoing frames decoupled to the greatest extent possible
- Easily Pipelined
  - Permits low per connection overhead

AMQP 1.0: Setup & Teardown

- Establish shared context for communication
  - Connection, Session, & Link Endpoints
    - Connection: open/close
    - Session: begin/end
    - Link: attach/detach
  - Setup/Teardown primitives are all “bookends”
    - Intervening transfers inherit context established by open, begin, and attach



AMQP 1.0: Setup & Teardown

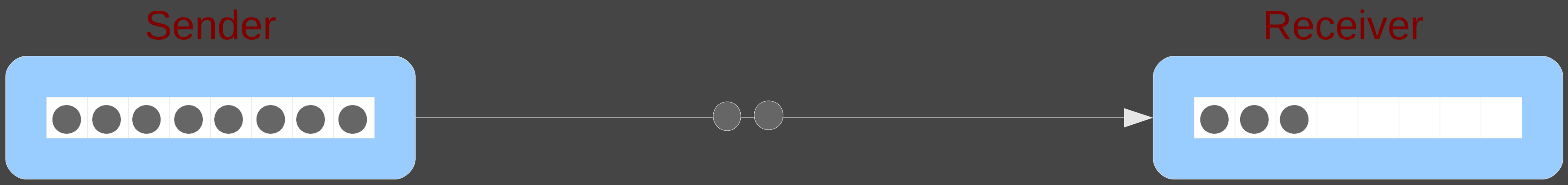
- Open – establishes properties of sending connection endpoint and binds container-id to physical connection
  - Includes basic facts, e.g. idle-timeout
  - Includes capabilities and limitations, e.g. channel-max
- Begin – establishes properties of sending session endpoint and binds session to channel
- Attach – establishes properties of sending link endpoint and binds link to handle
- Detach/End/Close – resource recovery and error codes

AMQP 1.0: Message Transfer

- Transfer – transmits message data
- Flow – communicates flow control state
  - Indicates available capacity at receiver
  - Indicates available transfers at sender
- Disposition – communicates outcome & settlement of transfer
  - Outcomes – Accepted, Rejected, Released
  - Settlement – transfer is done

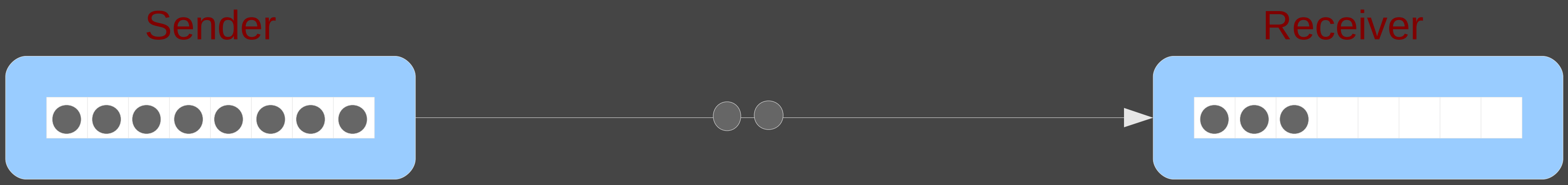
AMQP 1.0: Flow Control

- Propagates Receiver's constraints to Sender
- Two important categories of use
  - Network
    - Prevents redundant transmission
    - Enables maximum utilization of resources
  - Semantic
    - Unique to AMQP



AMQP 1.0: Flow Control

- Credit based scheme
  - Models receiver capacity at any given point
  - Advisory, enables simplistic implementation
- Augmented to indicate
  - Messages available at sender
  - Transient nature of receiver's capacity
    - Use it or lose it



AMQP 1.0: Messages

- AMQP defines the “bare” message as supplied by the application to consist of:
  - Standard Properties, Application Properties, and an opaque Body
- The “annotated” Message as produced by the network also includes:
  - Header & Footer Properties
- Properties are encoded using the AMQP type system
- The type system may also be used in the body

AMQP 1.0: Type System

- Goals of the type system
  - Expose enough of the message structure to AMQP infrastructure to enable key capabilities
    - Semantic Routing & Filtering, Message Archival, ...
  - Permit structured data exchange between endpoints written in different languages
    - JMS, WCF, Python, C++, Ruby, ...
- The type system is itself used to define all the protocol primitives

Apache Qpid: Open Source AMQP Messaging

- http://qpid.apache.org
- AMQP for everyone
  - Open source
  - Open community
  - Not controlled by any vendor
- The 'M' in Red Hat MRG

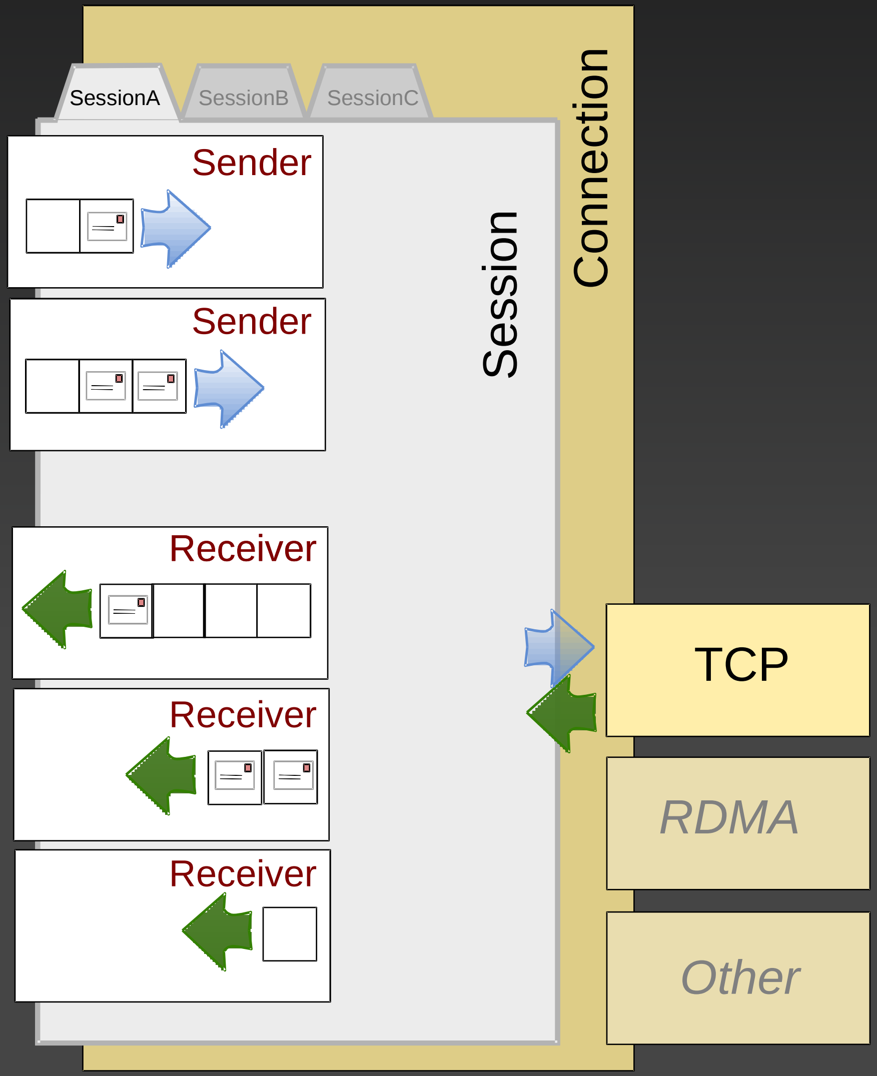
Goals of the Qpid Project

- Conceptually consistent API across a wide variety of languages
- Pluggable & composable IO
- Wide platform support
- Blocking & Non-blocking API
  - Easy to integrate
  - Support different threading models
- Support for structured messages

Apache Qpid: The messaging API

5 key classes:

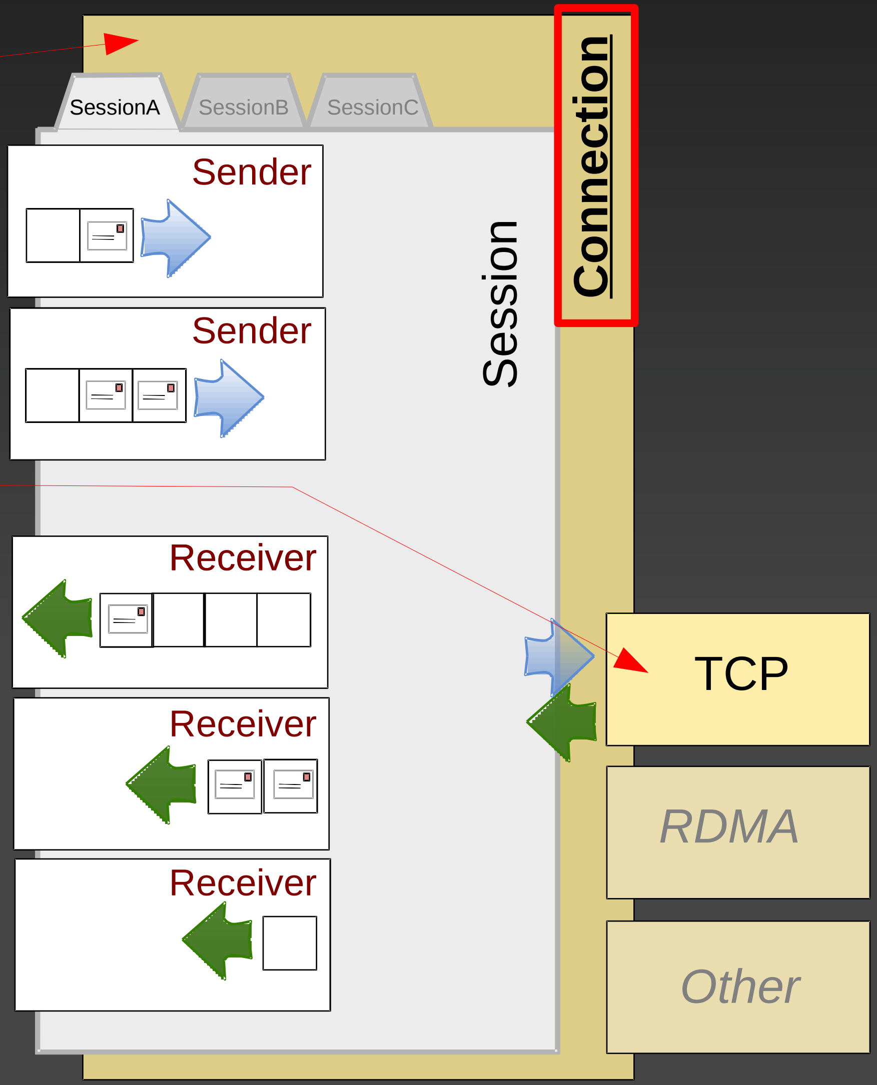
- Connection
- Session
- Sender
- Receiver
- Message



Apache Qpid: The messaging API

A **connection** provides a transport for getting messages across the network

It may for example use a TCP socket underneath, but other transports such as RDMA are possible.

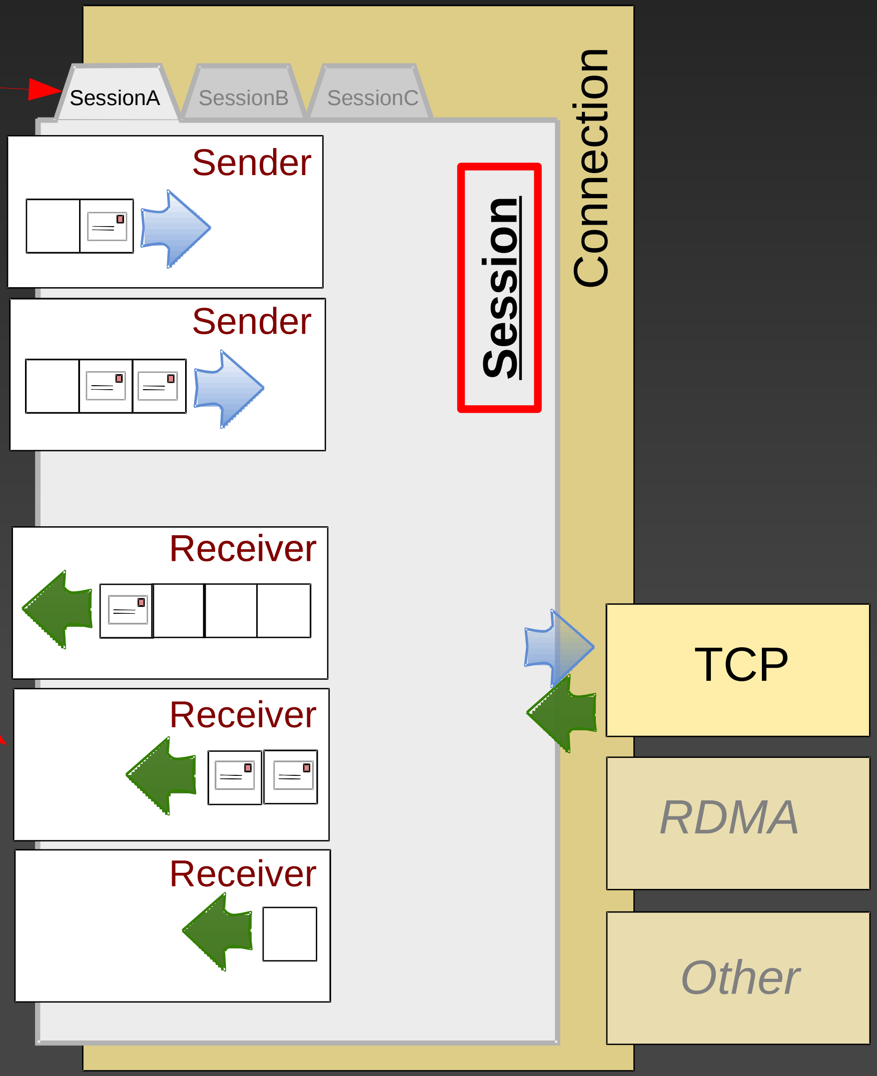


Apache Qpid: The messaging API

Several **sessions** can be multiplexed over a connection

Each session may have zero or more senders, through which outgoing messages are sent

Each session may have zero or more receivers, through which incoming messages are received



Apache Qpid: Hello World!

python client

```
from qpid.messaging import *                 #1



connection = Connection("localhost:5672") #2
try:
  connection.open()
  session = connection.session()           #3
  sender = session.sender(“my-queue”)      #4
  receiver = session.receiver(“my-queue”) #5
  sender.send(Message("Hello world!"),     #6
False)
  message = receiver.fetch(timeout=1)      #7
  print message.content
  session.acknowledge()                    #8
except MessagingError,m:
  print m
finally:
  connection.close()
```

c++ client

```
using namespace qpid::messaging;                                //1

int main(int argc, char** argv)
{
  Connection connection("localhost:5672");                      //2
  try {
    connection.open();
    Session session = connection.createSession();               //3
    Sender sender = session.createSender(“my-queue”);           //4
    Receiver receiver = session.createReceiver(“myqueue”);
    sender.send(Message("Hello world!"), false);                //6
    Message message = receiver.fetch(Duration::SECOND*1);//7
    std::cout << message.getContent() << std::endl;
    session.acknowledge();                                      //8

    connection.close();
    return 0;
  } catch(const std::exception& error) {
    std::cerr << error.what() << std::endl;
    connection.close();
    return 1;
  }
}
```

Red Hat Enterprise MRG

- Red Hat Enterprise **MRG** consist of
  - **M**essaging
    - AMQP asynchronous messaging system
    - Based on Apache Qpid
  - **R**ealtime
    - Realtime kernel
    - Based on Real Time “rt” kernel project
  - **G**rid
    - High-performance and high-throughput grid computing
    - Based on Wisconsin's project Condor

Join us!

Join us in building an open, rich and pervasive messaging infrastructure!

http://qpid.apache.org

Join us in making sure MRG / Messaging is high-quality enterprise messaging infrastructure!

http://redhat.com/careers

Discussion

Thank you for your attention!

Questions ?