PRESENTED BY RED HAT

LEARN. NETWORK.
EXPERIENCE OPEN SOURCE.

www.theredhatsummit.com

TOWARDS UNIFIED MESSAGING

Rafael Schloming (Red Hat)

Gordon Sim (Red Hat)

Thursday 5th May, 2011

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

Overview

- Asynchronous Messaging: What is it? Why use it?
- AMQP: The missing protocol?
- Apache Qpid: Open Source AMQP Messaging

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
  - 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?
  - 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
- 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: The missing protocol?

"We have collectively noted that open standards efforts between companies to automate electronic transactions are often hindered by the need to incorporate proprietary solutions at the messaging layers of such protocol stacks. This lack of a capable, open transport undermines the aims of such standards and imposes unnecessary barriers to e-commerce, distributed systems, electronic marketplaces and ultimately to free trade itself.

Messaging middleware is also a vital internal component of many applications, yet until now the market has failed to provide a unified, standardized solution for this necessary utility."

~ from www.amqp.org

AMQP: Standardized asynchronous messaging

- http://www.amqp.org
- Evolving for past 5 years (driven by actual implementations in real deployments)
- Final 1.0 specification ready for publication this year
- Complimentary 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,

Novell,

Progress Software,

Rabbit Technologies Ltd.,

Red Hat Inc.,

Software AG,

Solace Systems Inc.,

StormMQ Ltd.,

Tervela Inc.,

TWIST Process Innovations Ltd,
VMware, Inc.,

WS02 Inc.

29West Inc.

AMQP 1.0: An Overview

- Protocol primitives
  - open
  - begin
  - attach
  - transfer
  - disposition
  - flow
  - detach
  - end
  - close

Setup

Message Transfer

Teardown

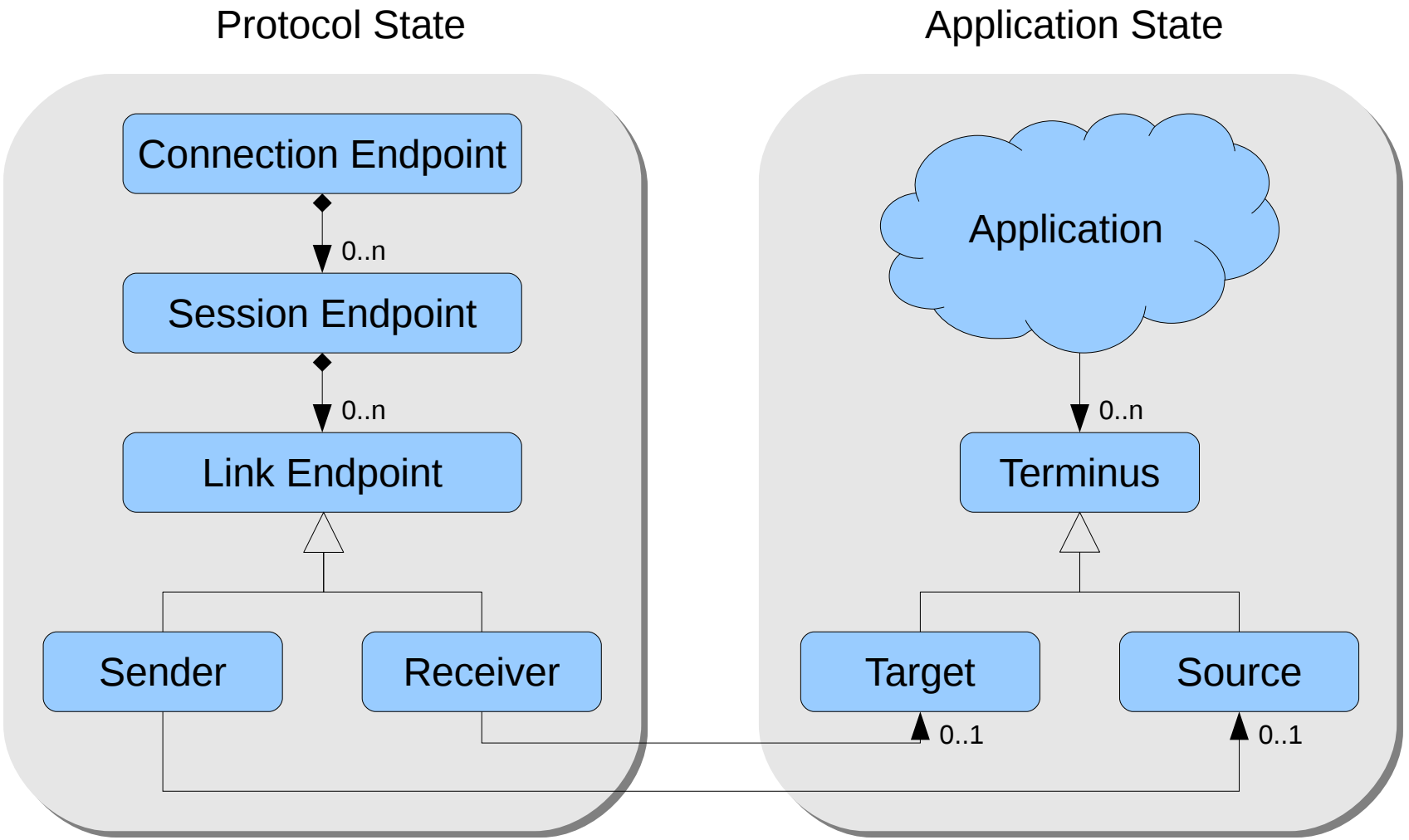
AMQP 1.0: Fundamentals of Communication

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

AMQP 1.0: Fundamentals of Communication

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

AMQP 1.0: Endpoint State



AMQP 1.0: Terminology

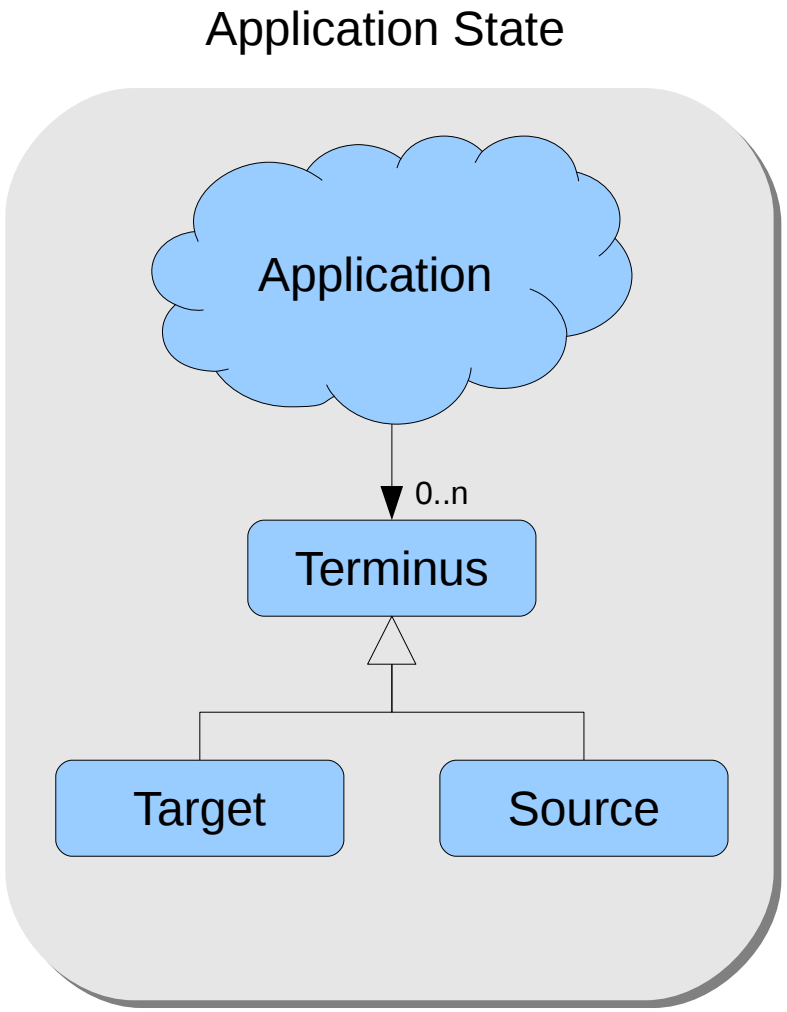
- Logical “overlay” network of Nodes & Links
  - Nodes are points in the AMQP network that can produce, consume, relay, or even transform messages
  - Links are unidirectional paths on which messages can flow between Nodes
- Physical “underlay” network for active communication
  - Connections, Sessions,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

AMQP 1.0: Endpoint State

- Incoming links are explicitly modeled
  - Unique to AMQP
  - Vital for flow control
  - Provides a point for deduplication
- Symmetric, not client/server
  - Simplification
  - Required for composability
    - Enables chaining without explicit translation and bridging

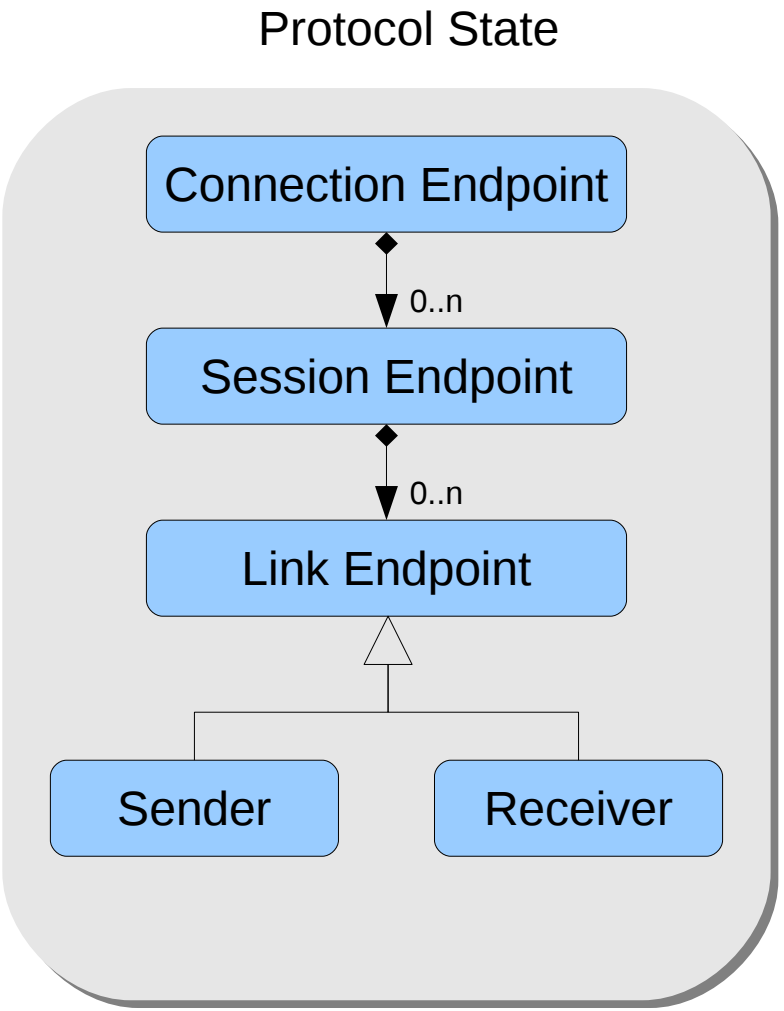
AMQP 1.0: Application State

- Application defines:
  - Lifespan
  - Identity
  - Retry Policy
  - Deduplication Policy



AMQP 1.0: Protocol State

- Scoped to TCP Connection
- Shared context for:
  - Settlement
  - Flow Control



AMQP 1.0: An Overview

- Protocol primitives
  - open
  - begin
  - attach
  - transfer
  - disposition
  - flow
  - detach
  - end
  - close

Setup

Message Transfer

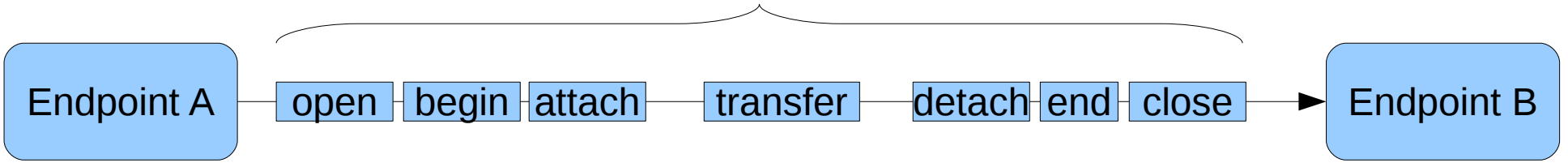
Teardown

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: Setup & Teardown

- Defaults are conservative & simple
  - One session per connection
  - One link per session
  - 4K max frame size
- Advanced behavior may be permitted once both local and remote capabilities are known
  - Multiple simultaneous sessions
  - Multiple simultaneous links
  - Larger frame size

AMQP 1.0: Message Transfer

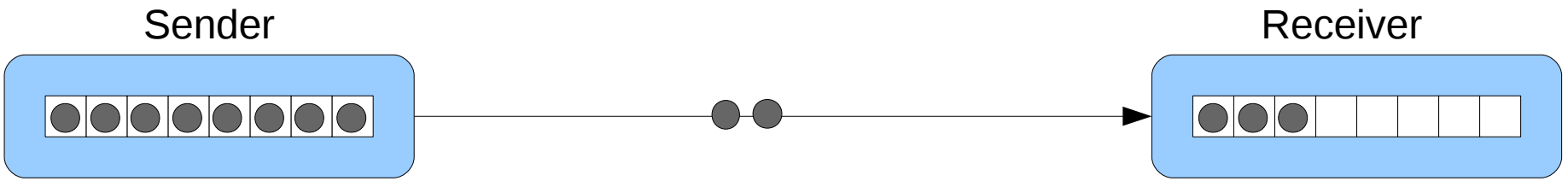
- Transfer – transmits message data
- Flow – communicates flow control state
  - Indicates available capacity at receiver
  - Indicates available transfers at sender
- Disposition – communicates outcome & settlement of transfer
  - Outcomes – Accepted, Rejected, Released
  - Settlement – transfer is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

- Flow primitives can be used to produce a variety of semantic behaviors needed by receiving applications
  - Polling
    - Is there anything there right now?
  - Fetch with timeout (also blocking fetch)
    - Give me the next message if available within the timeout.
  - Windowing
    - Give me messages as available up to a sliding limit.
  - Rate limiting
    - Give me messages as available within a maximum rate.

AMQP 1.0: Settlement

- The goal of the transport is an unambiguous resolution, not necessarily a successful one
  - Acknowledgment implies a positive outcome, AMQP also allows transfers to be refused.
- A transfer is settled at an endpoint when that endpoint no longer retains any knowledge of the transfer
  - Different settlement policies produce different reliability semantics

AMQP 1.0: Settlement Policies

- Sender settles on transmit
  - Fire and forget
- Sender settles after receiver settles
  - At least once with simple receiver
  - Exactly once with dedup receiver
- Dedup receiver can use settlement to trim its id sets
- More advanced behaviors are possible
  - Ring buffer, or timeout based settlement
    - Independent policies at sender or receiver

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

AMQP 1.0: Type System

- Self Describing
  - Required for infrastructure to understand messages
  - Ideal fit for distributed applications
    - No need to deal with low level versioned schemas
- Uniquely extensible via “descriptors” (type annotations)
  - Ideal for data binding & automatic marshaling systems
- Primitive types well matched to programming languages
  - Includes maps, lists, ints, floats, etc

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

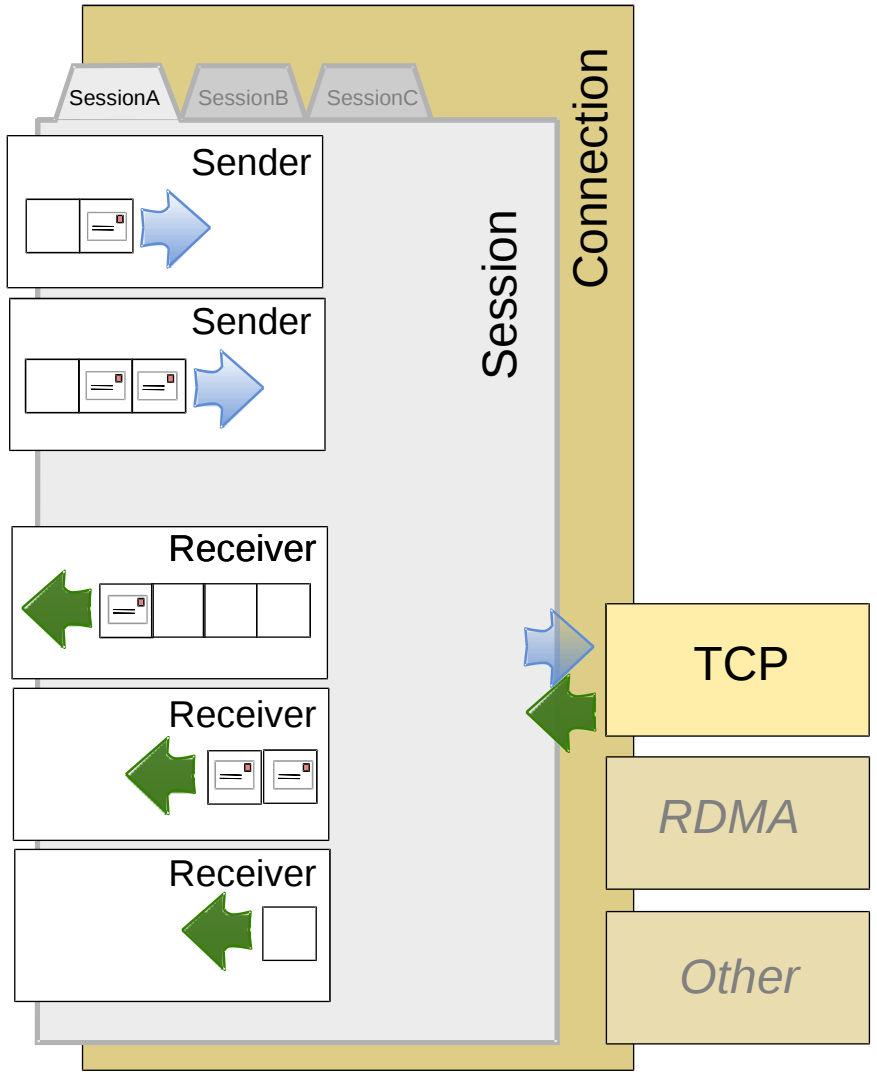
Goals of the Qpid Project

- 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:

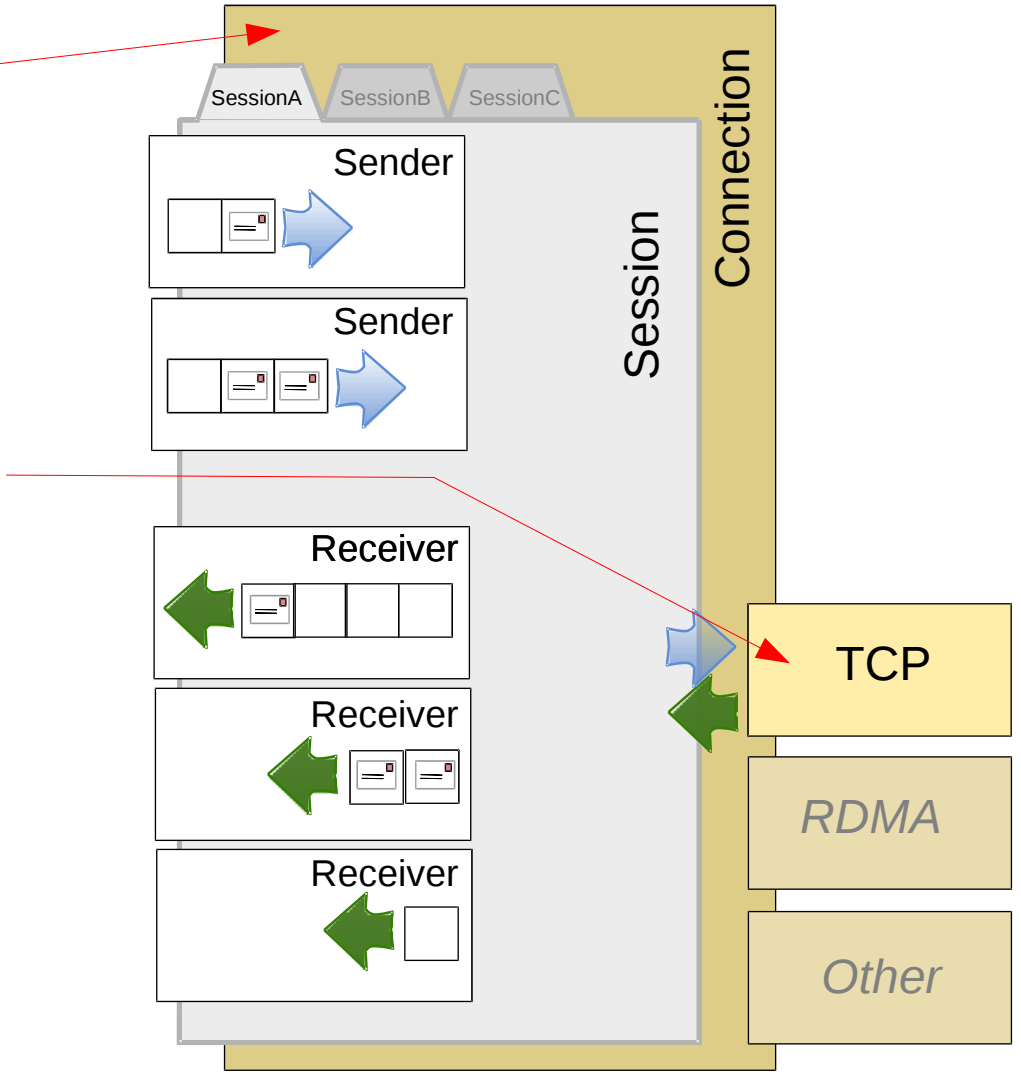
- Connection
- Session
- Sender
- Receiver
- Message



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.

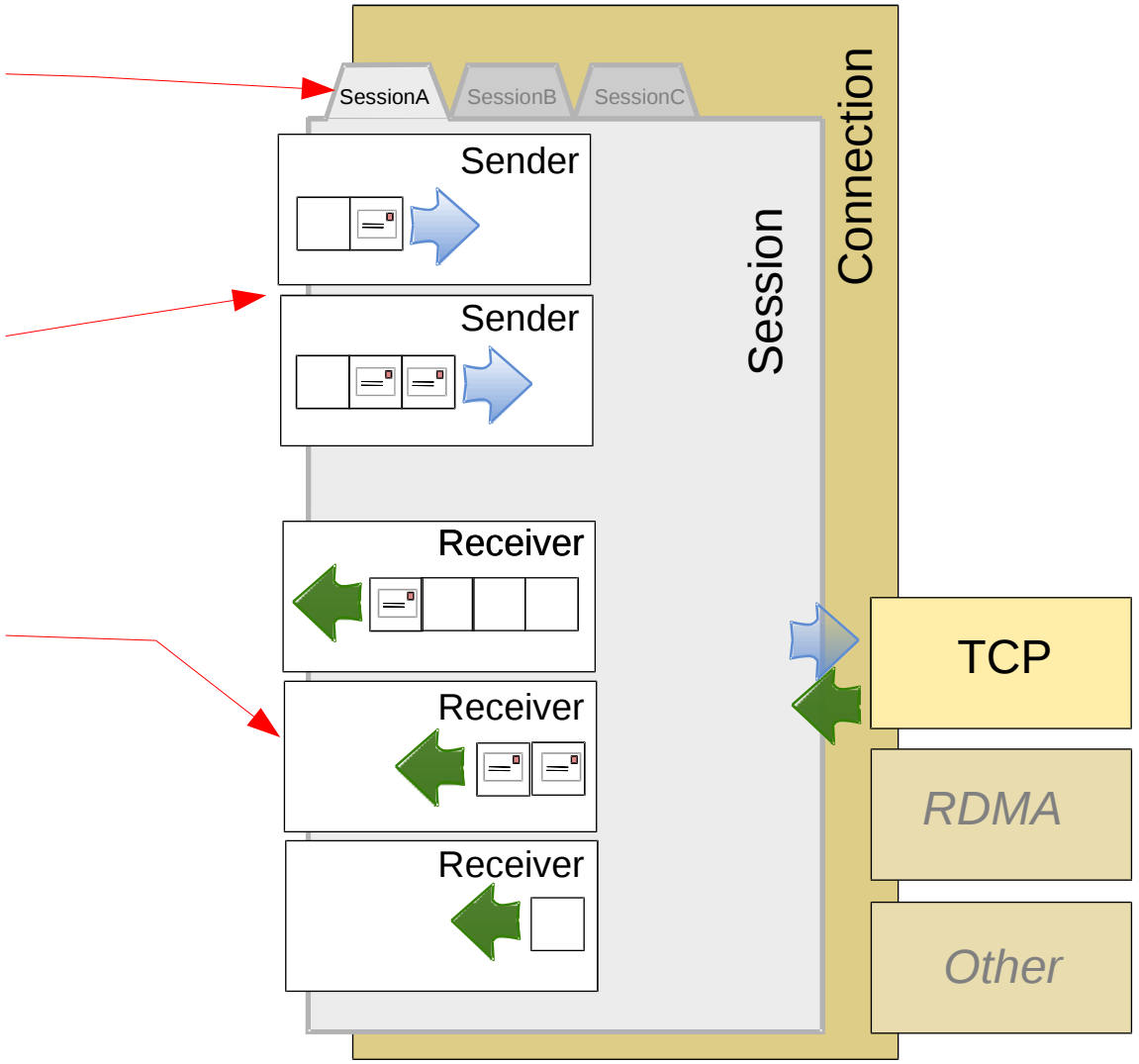


Apache Qpid: The messaging API

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

```
from qpid.messaging import *


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

c++

```
using namespace qpid::messaging;

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

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


Apache Qpid: Hello World!

```
from qpid.messaging import *

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


Apache Qpid: Hello World!

```
using namespace qpid::messaging;

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

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


Components in a Messaging Infrastructure

- *Applications*
- *Brokers*
  - Queuing, reliability and delivery guarantees
  - Routing & filtering
- *Concentrators*
- *Proxies*
- *Bridges*
- *Diagnostics*
- *Archiving*
  - Replay
  - Audit

Q&A

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

http://qpid.apache.org

LIKE US ON FACEBOOK

www.facebook.com/redhatinc

FOLLOW US ON TWITTER

www.twitter.com/redhatsummit

TWEET ABOUT IT

#redhat

READ THE BLOG

summitblog.redhat.com

GIVE US FEEDBACK

www.redhat.com/summit/survey