JUDCon

JBoss Users & Developers Conference

2012:Boston

Message Groups and MRG-M

Ajay Madhavan

Ajay.Madhavan@cmegroup.com

Joel Tosi

jtosi@redhat.com

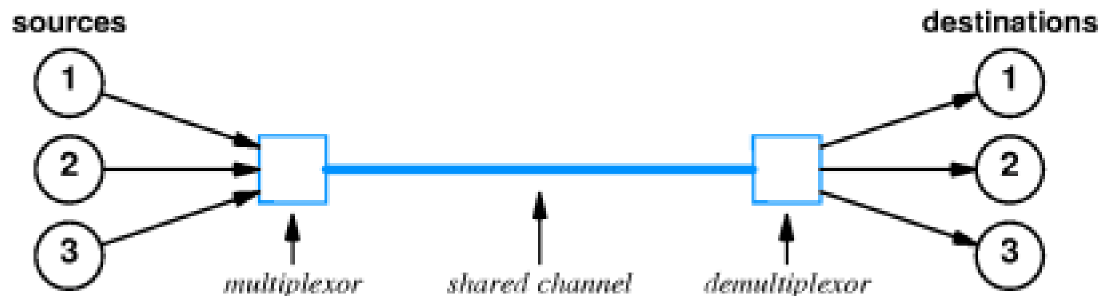
Ken Giusti

kgiusti@apache.org

June 25, 2012

The Case Of The Fat Pipe

- Sharing the same communication channel (physical or logical)
- Also called multiplexing and de-multiplexing electronics
- Referred to as packet switching in data networks



We See It All Around Us

- Cable TV/Satellite
  - Multiple channels on the same cable
- Home Internet (DSL/NAT)
  - Multiple computers using the Internet
  - Phone and DSL on the same wire
- Mail (yes - snail mail)!!
  - Single USPS truck with lots of mail

How is it done generally?

- Address on the transmission unit (CDMA)
  - Packet switched networks
  - email
- Timing sender and receiver (TDMA)
  - Can’t think of one.. Seriously…
- Sub dividing the communication channel, if possible (e.g. FDMA)
  - OTA TV/Radio

Why The Need For The Fat Fella.. er.... Fat Pipe

- To shared an expensive, otherwise under utilized, communication channel
- Scaling is easy

Weren't You Going To Talk About Messaging Systems...

Messaging Systems P2P/Pub-Sub implement these concepts too

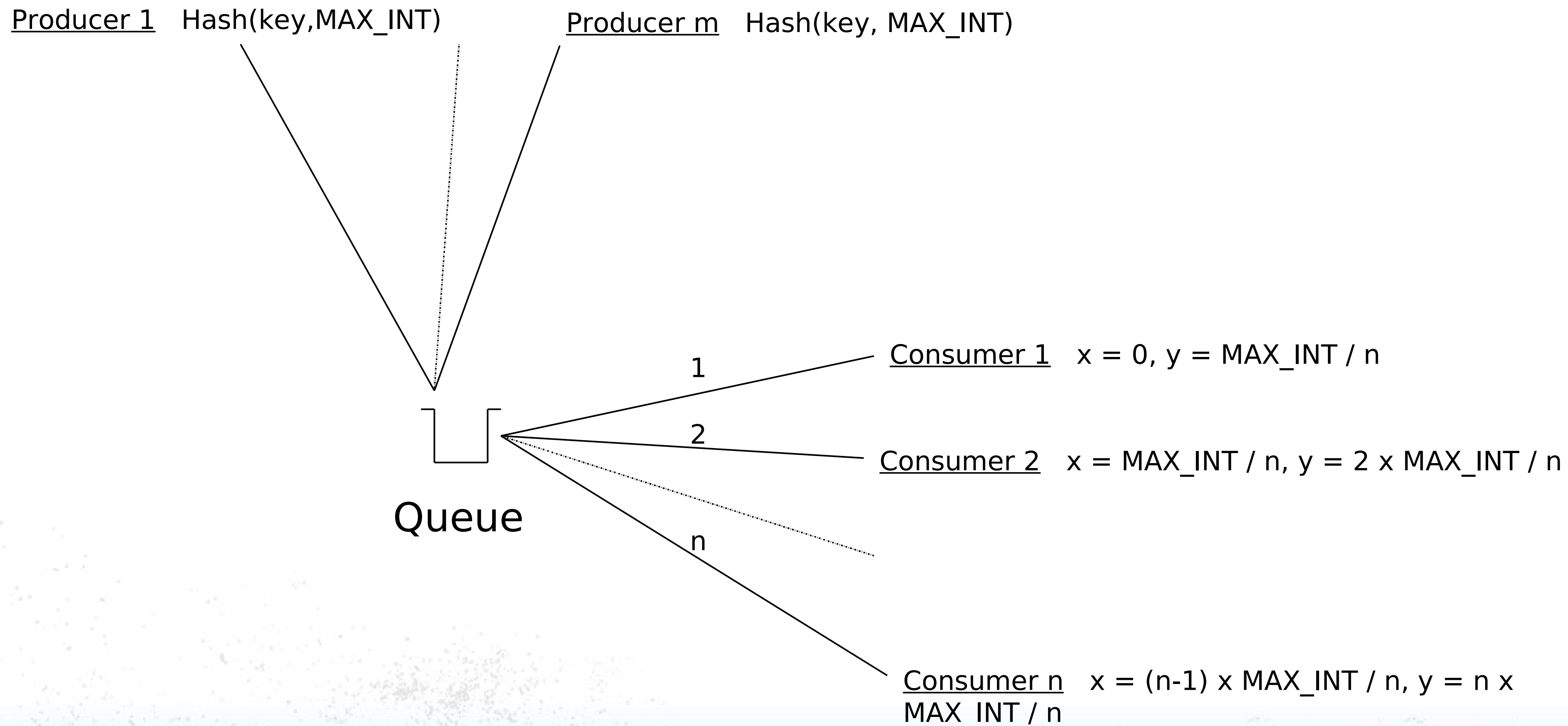
Consider Pub/Sub

- They share a single bus (logical communication channel, maybe a multicast address)
- Subjects are used to DEMUX

Consider P2P

- A broker that routes based on header is essential
- Can also be done by using message selector

MUX/DEMUX Concept In Messaging Systems With Un-equal Producers/Consumers



The Concept Of Message Groups

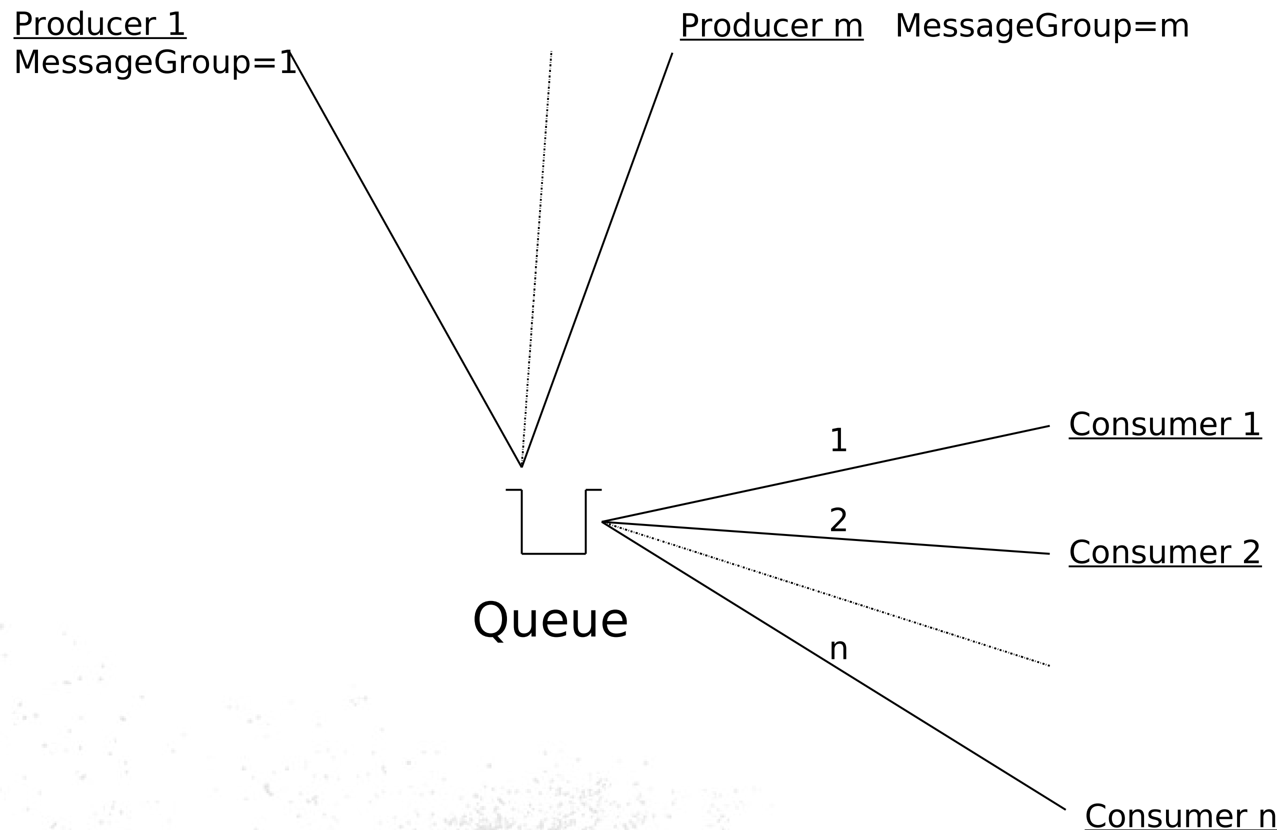
Premise

"Message belonging to a group will be processed mutually exclusively of other messages in the group"

Message Groups

- Well known "Message Group" header identifies mutual exclusivity
- Sender identifies the "Group" by populating the header
- The hub (broker/router) enforces mutual exclusion (external sync point)
- Start of critical section starts on read
- End of critical section upon transaction end (acknowledged/released/rejected)

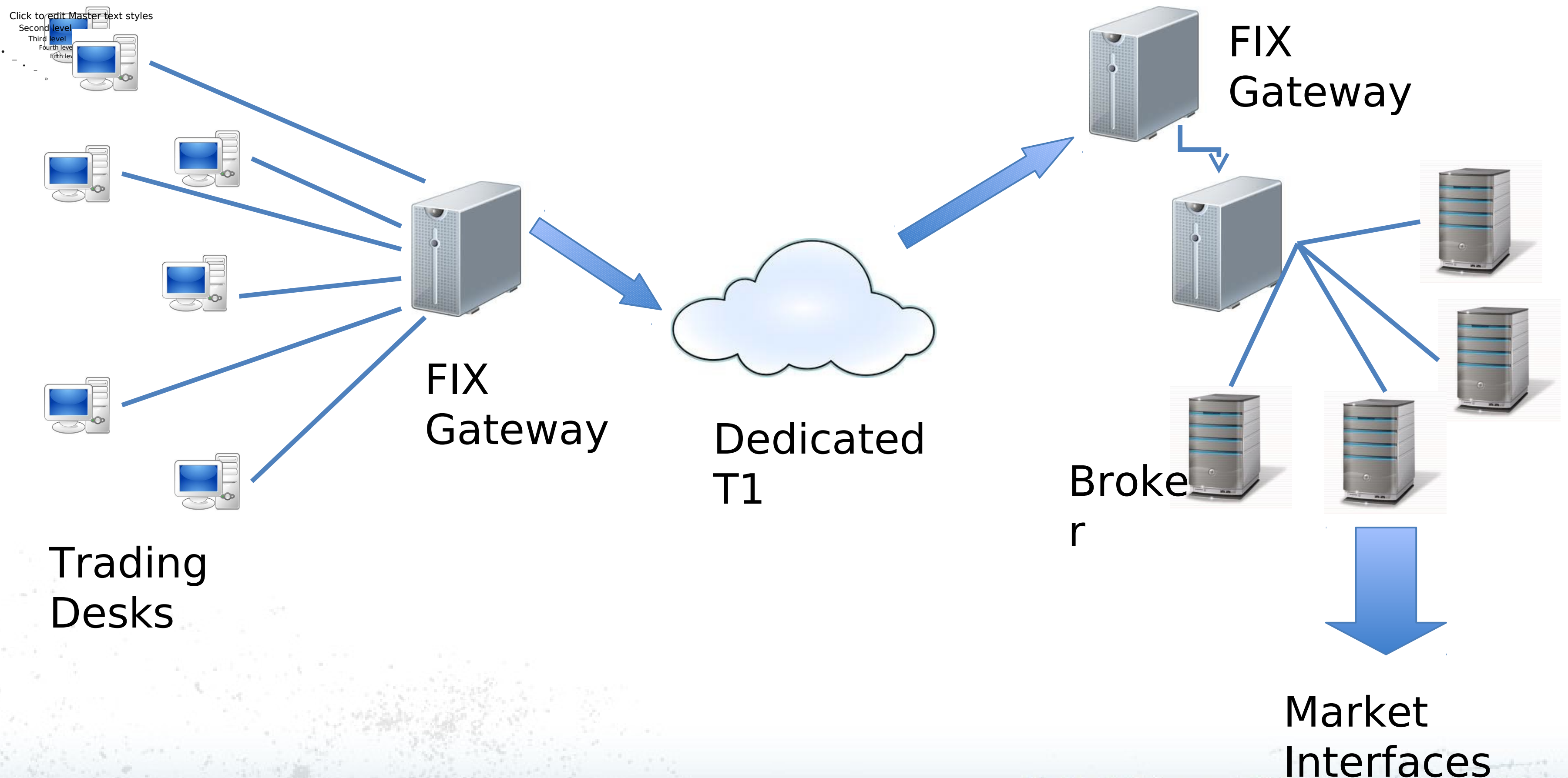
MUX/DEMUX With Message Groups



What Can We Do With Message Groups

- Scalability
  - Just add more consumers or producers
- Fault Tolerance
  - Death of a consumer has no implication
    - Consumer should be stateless
    - Consumer should always read/persist/write within a transaction boundary
- Automatic Workload Management
  - Any free consumer is free to take on any work available

Real World Scenario



Message Groups in QPID/MRG-M

QPID Message Group Implementation

Ken Giusti

kgiusti@apache.org

Developer – Apache QPID Project

Principal Software Engineer – Redhat/MRG-M

June 25, 2012

What is "QPID"?

Apache Software Foundation (ASF)

Apache Qpid™ is a cross-platform Enterprise Messaging system which implements the Advanced Message Queuing Protocol (AMQP), providing message brokers written in C++ and Java, along with clients for C++, Java JMS, .Net, Python, and Ruby.

Open Source – Apache License, Version 2.0

"Ready to Run" Brokers (C++/Java)

Client Tools and Libraries

http://qpid.apache.org

What is “AMQP”?

Advanced Message Queuing Protocol

- Open standard message-oriented middleware
- Industry Consortium/OASIS Technical Committee
- Messaging Protocol
  - Broker/Client model (v0.10)
  - Peer-to-Peer (v1.0)
- Message Structure
- Type System
- Wire-level Binary encoding

http://www.amqp.org

What is “MRG-Messaging”?

Redhat Enterprise Messaging Product

Based on QPID

QA'd against RHEL

Long term support

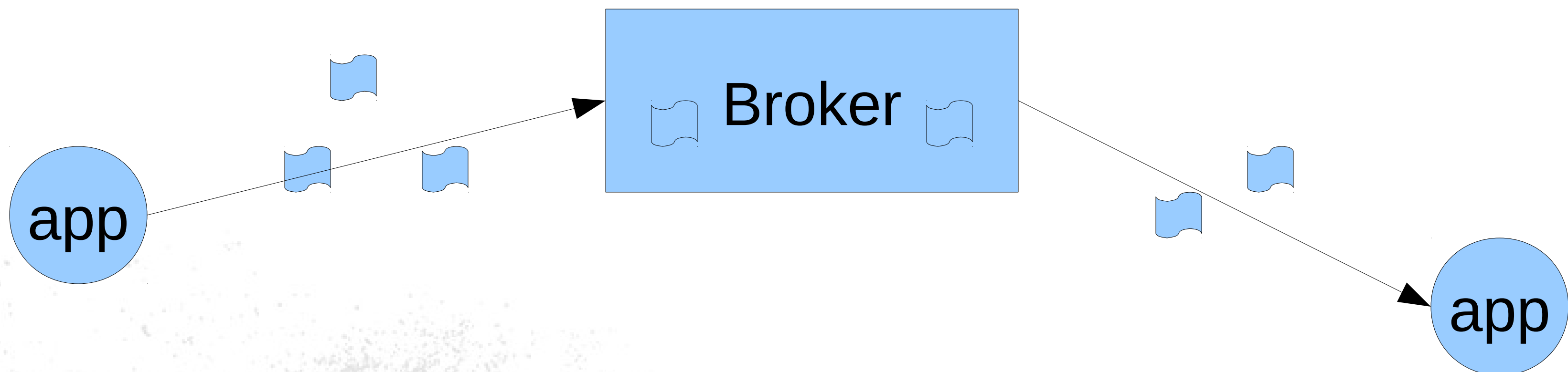
The QPID Model

Principal players:

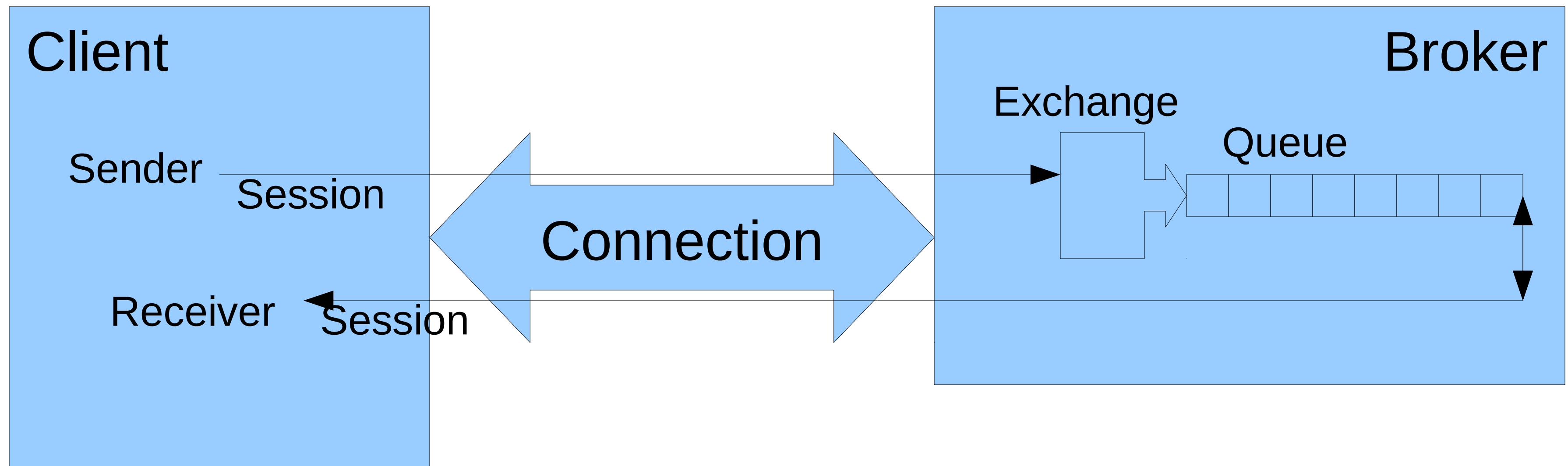
Client Applications

Broker

Messages



The QPID Model



The Queue Object

- Message Storage
- Different types (policies):
  - FIFO
  - Priority
  - Message Groups
- Single Queue Abstraction
  - Durability
  - Transactions

The Queue Object

Operations provided by the Queue abstraction:

enqueue:	done by producers, available for consumers
acquire:	by consumer, no longer available, but not yet fully transferred
delete:	remove from queue (acknowledged)
release:	(unacquired) put back on the queue, made available again.

QPID Message Group Queue

A FIFO Queue that is group aware.

1) Classifies arriving messages by group.
2) Tracks the state of all known groups:
    - Creates states as necessary
    - Deletes states when no longer needed
3) Enforces ownership of a group by a consumer:
    - Determine the “next available message”

Group State Class

```
struct GroupState {

    std::string group;  // group identifier
    std::string owner;  // consumer with acquired messages
    uint32_t acquired;  // count of outstanding acquired messages

   struct MessageState {
        SequenceNumber position;
        bool           acquired;
        MessageState(const qpid::framing::SequenceNumber& p);
        bool operator<(const MessageState& b) const;
    };
    typedef std::deque<MessageState> MessageFifo;
    MessageFifo members;   // msgs belonging to this group

    GroupState() : acquired(0) {}
    bool owned() const {return !owner.empty();}
    MessageFifo::iterator findMsg(const SequenceNumber &);
};

typedef sys::unordered_map<std::string, struct GroupState> GroupMap;
typedef std::map<SequenceNumber, struct GroupState *> GroupFifo;

GroupMap messageGroups; // index: group name
GroupFifo freeGroups;   // ordered by oldest free msg
```

Message Arrival

```
void enqueued( const QueuedMessage& qm )
{
    GroupState& state = findGroup(qm);
    GroupState::MessageState mState(qm.position);
    state.members.push_back(mState);
    uint32_t total = state.members.size();
    QPID_LOG( trace, "group queue " << qName <<
              ": added message to group id=" << state.group <<
              " total=" << total );
    if (total == 1) {
        // newly created group, no owner
        assert(freeGroups.find(qm.position) == freeGroups.end());
        freeGroups[qm.position] = &state;
    }
}
```

Message Selection

```
bool nextConsumableMessage( Consumer& c, QueuedMessage& next )
{
    next.position = c->getPosition();
    if (!freeGroups.empty()) {
        const framing::SequenceNumber& nextFree = freeGroups.begin()->first;
        if (nextFree <= next.position) {  // take oldest free
            next.position = nextFree;
            --next.position;
        }
    }

    while (messages.browse( next.position, next, true )) {
        GroupState& group = findGroup(next);
        if (!group.owned()) {
            own( group, c );
            return true;
        } else if (group.owner == c->getName()) {
            return true;
        }
    }
    return false;
}
```

Message Acquire

```
void acquired( const QueuedMessage& qm )
{
    GroupState& state = findGroup(qm);
    GroupState::MessageFifo::iterator m = state.findMsg(qm.position);
    assert(m != state.members.end());
    m->acquired = true;
    state.acquired += 1;
}
```

Message Dequeue

```
void dequeued( const QueuedMessage& qm )
{
    GroupState& state = findGroup(qm);
    GroupState::MessageFifo::iterator m = state.findMsg(qm.position);
    if (m->acquired) {
        state.acquired -= 1;
    }
    state.members.erase(m);

    if (state.members.size() == 0) {
        messageGroups.erase( state.group );
    } else if (state.acquired == 0 && state.owned()) {
        disown(state);
    }
}
```

QPID Message Group Queue Configuration

- **Via qpid-config:**

```
qpid-config add queue <name>
                 --group-header="<key>"
                 --shared-groups
```

- **Via messaging API address string syntax:**

```
s = session.createSender("<name> {create:always,
    node:{x-declare: {arguments:
         {'qpid.group_header_key':'<key>',
          'qpid.shared_msg_group': true}}}}")
```

QPID Message Groups Producer Client Code

- **Java**

```
String groupKey = “<key>”;


TextMessage msg = ssn.createTextMessage(“data”);
msg.setStringProperty(groupKey, “group1”);
sender.send(msg);
```

QPID Message Groups Producer Client Code

- **C++**

```
std::string groupKey = “<key>”;

Message msg(“data”);
msg.getProperties()[groupKey] = std::string(“group1”);
sender.send(msg);
```

QPID Message Groups Consumer Client Code

- .... <crickets> ....

Nothing special needs to be done by the Consumer, except, of course:

Don't Ack a message until you are done processing that message!

“Well Behaved Consumer”

QPID Message Groups Debug-ability

- QMF Broker Query method:

```
rc = broker.query(“queue”, “<queue name>”);
```

- rc.outArgs[“results”] returns a map holding the state of the group queue:

```
{group_header_key: <key>,
   group_state: [ {group_id: <id>,
                   msg_count: <#>,
                   consumer: “ ... “},
```

QPID Message Groups

Questions and Demo...