



RED HAT ENTERPRISE MRG MESSAGING 2.0: GETTING STARTED GUIDE HIGH-PERFORMANCE, RELIABLE, OPEN AMQP MESSAGING

1	WHAT IS MRG-M?	3
1.1	The MRG-M Model	3
1.2	The MRG-M Wire-level Format	5
1.3	Messages in MRG-M	6
2	GETTING STARTED	6
2.1	Prerequisites	6
2.2	Install the example code	7
3	HANDS-ON: SINGLE BROKER	7
3.1	Starting the Message Server	7
3.2	Point-to-Point Messaging using the Java JMS API	8
3.3	Persistent Point-to-Point Messaging using the Java JMS API	11
3.4	Point-to-Point Messages using the Java JMS API and Python Client API	14
3.5	Publish/Subscribe Messaging with Topics using the Python Client API	15
4	HANDS-ON: MESSAGING CLUSTERS	20
4.1	Starting Clustered Message Servers	20
4.2	Publish/Subscribe Messaging with Topics using the Python Client API	21
	Appendix A: Installing MRG-M on Red Hat Enterprise Linux 6 (RHEL 6)	24
	Appendix B: MRG-M Clustering Setup	26



MRG MESSAGING: GETTING STARTED GUIDE

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

Linux is a registered trademark of Linus Torvalds. Red Hat, Red Hat Enterprise Linux and the Red Hat “Shadowman” logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation.

UNIX is a registered trademark of The Open Group.

Intel, the Intel logo, Xeon and Itanium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

© 2011 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E



1 WHAT IS RED HAT ENTERPRISE MRG MESSAGING?

JPMorgan Chase, a financial industry leader, decided to find a better way to enable messaging across its organization. Rather than implementing yet another proprietary messaging system, the company decided to sponsor an approach that could be replicated industry-wide and become the benchmark for the financial sector.

The new protocol had to be simple and language neutral. To achieve this, JPMorgan Chase decided that it must be ubiquitous and easily adopted. Eventually developers created AMQP, the network protocol for the messaging solution.

AMQP is an open Internet protocol for business messaging that enables complete interoperability for messaging middleware. Designed as a standard, it defines both the networking protocol and the semantics of broker services.

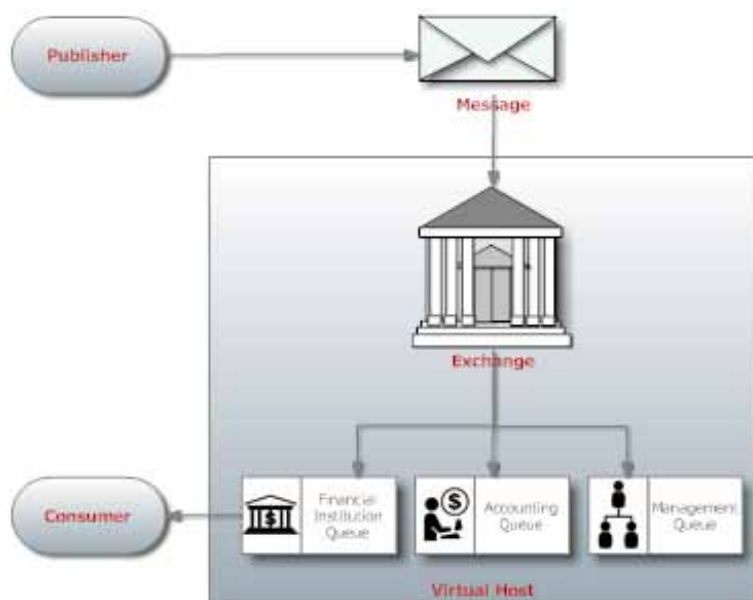
AMQP defines an efficient wire-level protocol with modern features that allows message producers and consumers to choose whatever technology they wish to envelop it. Red Hat® Enterprise MRG Messaging is Red Hat's implementation of AMQP.

1.1 THE RED HAT ENTERPRISE MRG MESSAGING MODEL

The Red Hat Enterprise MRG Messaging model explicitly defines a server's semantics because interoperability demands the same semantics for any server implementation. The model specifies a modular set of components and standard rules for connecting these components. It emulates the classic messaging concepts of store-and-forward queues and topic subscriptions. It is then enhanced by more advanced capabilities such as content-based routing, message queue forking, and on-demand message queues.

There are three main types of components that are connected into processing chains in the server to create the desired functionality:

- The **exchange** receives messages from publisher applications and routes these to message queues based on arbitrary criteria—usually message properties or content.
- **Message queues** store messages until they can be safely processed by a consuming client application (or multiple applications).
- **Bindings** define relationships between message queues and exchanges and provide the message routing criteria.





You can think of an Red Hat Enterprise MRG Messaging server much like an email server:

- Exchanges act as message transfer agents
- Each message queue is a mailbox
- Bindings define the routing tables in each transfer agent
- Publishers send messages to individual transfer agents
- Transfer agents route the messages into mailboxes
- Consumers take messages from mailboxes
- According to the specification, the implementation must:
- Guarantee interoperability between conforming implementations
- Provide explicit control over the quality of service
- Support any middleware domain: messaging, file transfer, streaming, RPC, etc.
- Accommodate existing open messaging API standards
- Be consistent and explicit in naming
- Allow complete configuration of server wiring via the protocol
- Use a command notation that maps easily into application-level API's
- Limit each operation to exactly one process

Red Hat Enterprise MRG Messaging supports a variety of message queues, including private or shared, durable or transient, and permanent or temporary. By selecting the desired properties, you can use a message queue to implement conventional middleware entities such as:

1. A standard store-and-forward queue, which holds messages and distributes these between subscribers on a round-robin basis. Store-and-forward queues are typically durable and shared between multiple subscribers.
2. A temporary reply queue, which holds messages and forwards them to a single subscriber. Reply queues are typically temporary and private to one subscriber.
3. A “pub-sub” subscription queue, which holds messages collected from various “subscribed” sources and forwards them to a single subscriber. Subscription queues are typically temporary and private to one subscriber.

There is no formal definition of these queues in AMQP; they are simply examples of how message queues can be defined. According to the specification, creating new entities such as durable, shared subscription queues, and those with persistence should be a minor task.

Prior to AMQP, most messaging architectures had several issues with their routing models:

- Opaque routing models were not explicitly defined
- Hidden semantics made changing the routing model through the protocol difficult
- Proprietary routing engines had limited or no extensibility or compose-ability



One of the design goals of the AMQP standard was to include explicitly-defined semantics supporting multiple routing models. Therefore, complex routing is well-supported in Red Hat Enterprise MRG Messaging.

In addition, part of the lure of Red Hat Enterprise MRG Messaging comes from its ability to create transient queues, exchanges, and routings at runtime and chain these together in ways that go far beyond a simple mapping of destinations as you would with JMS, for example.

In contrast, the challenge in Red Hat Enterprise MRG Messaging is routing and storing messages within and between servers. Routing within a server and routing between servers are distinct problems and have distinct solutions, if only for maintaining transparent performance.

To route between Red Hat Enterprise MRG Messaging servers with different owners, you set up an explicit bridge where one Red Hat Enterprise MRG Messaging server acts as the client of another Red Hat Enterprise MRG Messaging server for the purpose of transferring messages between owners. This fits early Red Hat Enterprise MRG Messaging adopters, since those bridges are likely to be preceded by complex business processes, contractual obligations and security concerns. This model also makes spamming with Red Hat Enterprise MRG Messaging more difficult.

1.2 THE RED HAT ENTERPRISE MRG MESSAGING WIRE-LEVEL FORMAT

The Red Hat Enterprise MRG Messaging wire-level format is a binary framing with modern features: It is multi-channel, negotiated, asynchronous, secure, portable, neutral, and efficient. It is compliant with the AMQP specification.

The wire-level format is split into two layers, a functional layer and a transport layer. The functional layer defines a set of commands (grouped into logical classes of functionality) that do useful work on behalf of the application. The transport layer carries these methods from application to server and back again, and also handles channel multiplexing, framing, content encoding, heart beating, data representation, and error handling. Both the transport layer and high-level layers are pluggable, allowing evolution of the protocol and the adoption of emerging technologies.

According to the specification, the wire-level format must:

- Be compact, using a binary encoding that packs and unpacks rapidly
- Handle messages of any size without significant limit
- Permit zero-copy data transfer (e.g. remote DMA)
- Carry multiple sessions across a single connection
- Allow sessions to survive network failure, server failover, and application recovery
- Be long lived, with no significant in-built limitations
- Be asynchronous
- Be easily extensible to handle new and changed needs
- Be forward compatible with future versions
- Be repairable, using a strong assertion model
- Be neutral with respect to programming languages
- Fit a code generation process



1.3 MESSAGES IN RED HAT ENTERPRISE MRG MESSAGING

A message is the atomic unit of routing and queuing. Messages have a header consisting of a defined set of properties, and a body that is an opaque block of binary data.

Messages in Red Hat Enterprise MRG Messaging have these characteristics:

- They may be persistent—a persistent message is held securely on disk and guaranteed to be delivered even if there is a serious network failure, server crash, overflow, etc.
- They can be prioritized—a high priority message may be sent ahead of lower priority messages waiting in the same message queue
- The server may modify specific message headers prior to forwarding them to the consumer

There are generally two types of messages that you may wish to send through a messaging system:

1. Transient messages have a contract that says messages may be lost if the messaging system itself loses transient state (e.g. in the case of a power outage).
2. Durable messages must make the guarantee that the message will be held in the most durable store available for future triage after adverse runtime conditions are mitigated.

Red Hat Enterprise MRG Messaging supports both of these message types.

Red Hat Enterprise MRG Messaging also supports a variety of messaging transport architectures:

1. Store-and-forward with many writers and one reader
2. Transaction distribution with many writers and many readers
3. Publish-subscribe with many writers and many readers
4. Content-based routing with many writers and many readers
5. Queued file transfer with many writers and many readers
6. Point-to-point connection between two peers

2 GETTING STARTED

2.1 PREREQUISITES

The first hands-on section in this document features technical examples designed to demonstrate three basic uses of a stand-alone Red Hat Enterprise MRG Messaging server:

- Point-to-point messaging
- Publish/subscribe messaging
- Persistent messaging

The second hands-on section extends the publish and subscribe messaging example to use a pair of clustered Red Hat Enterprise MRG Messaging servers.



The hands-on sections that follow assume that you have a machine with Red Hat Enterprise Linux® 6.1 installed and that you have the appropriate access to install applications and modify system configuration files. In addition, the examples assume that both the Red Hat Enterprise MRG Messaging client and server are running on the same machine.

To install Red Hat Enterprise MRG Messaging and the supporting software for this Getting Started Guide, follow the instructions **Installing Red Hat Enterprise MRG Messaging in the Appendix**.

2.2 INSTALL THE EXAMPLE CODE

To install the example code for this getting started guide, download the zipped archive from

www.redhat.com/f/zip/getting-started-mrgm2-examples.zip and unzip to your home directory. You should now have a directory structure similar to the following: `~/mrg-m/examples`

Note: If you would like to verify the download, the GPG signature can be downloaded from www.redhat.com/f/zip/getting-started-mrgm2-examples.zip.asc. The signing key used is Red Hat's release key 2 (see: access.redhat.com/security/team/key/).

3 HANDS-ON: SINGLE BROKER

3.1 STARTING THE MESSAGE SERVER

All of the examples require a running messaging service. This section will show you how to get the message server to run.

1. Open a new Terminal window and run the following command (as a normal user):

```
$ /usr/sbin/qpid --auth=no
```

2. This will start the messaging server. If the server started successfully, one of the last lines in the Terminal window should say Broker running.

```
11:30:37 notice SASL disabled: No Aut
11:30:37 notice Listening on TCP port
11:30:37 notice Broker running
11:30:47 warning Timer callback overr
12-31-38-04-C5-24 -j# █
```

3. You are ready to try one of the example programs. Minimize the broker window.

Important: Make sure not to close this window, because it will shut down the server and none of the examples will work!



3.2 POINT-TO-POINT MESSAGING USING THE JAVA JMS API

This example illustrates simple point-to-point functionality, using the default exchange and default binding to route a message directly to a queue. In these steps you will first run a command that populates a queue on the server with five messages. Next you will run a command that reads all available messages on the queue and prints them to the screen.

1. Open a new Terminal window.
2. First, we need a queue to use to send messages. We will create a queue that will automatically be bound to the default exchange using the message queue's name as the binding key. Run the following command:

```
$ qpid-config add queue stockprices
```

3. Run the following command to verify that the queue has been successfully created:

```
$ qpid-config -b queues stockprices
```

4. You should see the queue you just created along with a binding to the default exchange ('') using [stockprices] as the binding key.

```
$ qpid-config -b queues stockprices  
Queue 'stockprices'  
  bind [stockprices] => ''
```

5. Change to the examples directory you unzipped previously (e.g., ~/mrg-m/examples)



6. Run the command: `./run.sh P2PSender.java`

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh P2PSender.java
Building class path...Done.
Running example:
Sending message: 1 GOOG 616.47
Sending message: 2 RHT 42.46
Sending message: 3 VMW 78.25
Sending message: 4 CSCO 23.29
Sending message: 5 IBM 141.43
[root@domU-12-31-38-04-C5-24 examples]#
```

7. The P2PSender.java code added five messages through the default exchange. The messaging client uses the Java JMS API to communicate to the MRG-M broker.

- a. First, we initialize and then create a JNDI Initial Context:

```
... set jndi properties ...
properties.put("connectionfactory.qpidconnectionfactory",
               "amqp://guest:guest@clientid/test?
               brokerlist='tcp://localhost:5672'");
properties.put("destination.stockPrices", "stockprices");
Context context = new InitialContext(properties);
```

- b. Next, lookup a connection factory from JNDI and create and connect to the Red Hat Enterprise MRH Messaging broker:

```
ConnectionFactory connectionFactory = (ConnectionFactory)
    context.lookup("qpidconnectionfactory");

connection = connectionFactory.createConnection();

connection.start();
```

- c. Then we create a session from the connection, route the session through a queue, and create a message producer for that session and queue:

```
Session session = connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
Destination destination =
    (Destination) context.lookup("stockPrices");

MessageProducer producer =
    session.createProducer(destination);
TextMessage message = session.createTextMessage();
```



d. We then loop through all the messages and send them:

```
for (int i = 0; i < messages.length; i++) {  
    message.setText((i + 1) + " " + messages[i]);  
    System.out.println("Sending message: " + message.getText());  
    producer.send(message);  
}
```

e. Finally, we send the control message, which tells the consumer that the end of messages has been received. This is simply a text message containing the pattern "END." There is nothing special about this message to the messaging system; the consumer example code is simply programmed to terminate when it sees that message:

```
message.setText("END");  
producer.send(message);
```

8. Now that messages are waiting on the server, we can run the command to retrieve them. To do so, run the command: `./run.sh P2PReceiver.java`

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh P2PReceiver.java  
Building class path...Done.  
Running example:  
Reading message: 1 GOOG 616.47  
Reading message: 2 RHT 42.46  
Reading message: 3 VMW 78.25  
Reading message: 4 CSCO 23.29  
Reading message: 5 IBM 141.43  
[root@domU-12-31-38-04-C5-24 examples]#
```



9. The P2PReceiver.java code connects to the queue on the server and retrieves messages until the control message "END" is received.
 - a. The sender and receiver commands are almost identical until the MessageConsumer—rather than the MessageProducer—is created. In the receiver we start the connection and create a consumer object instead of a producer object.

```
connection.start();

Session session = connection.createQueueSession(false,
                                                Session.AUTO_ACKNOWLEDGE);
Destination destination =
    (Destination) context.lookup("stockPrices");

MessageConsumer consumer = session.createConsumer(destination);
TextMessage message = null;
```

- b. Finally, the command loops indefinitely until the control message "END" is received:

```
while (true) {

    message = (TextMessage) messageConsumer.receive(1000);

    if (message != null) {
        if (((TextMessage) message).getText().equals("END")) {
            break;
        } else {
            System.out.println("Reading message: "
                               + message.getText());
        }
    }
}
```

3.3 PERSISTENT POINT-TO-POINT MESSAGING USING THE JAVA JMS API

This example illustrates point-to-point messaging functionality with persistence. These steps are identical to the previous point-to-point messaging example with one exception: here the messages have been destined for a durable (persistent) queue, and we will restart the broker between sending and receiving messages.

1. Open a new Terminal window.



2. First, we need to create a persistent (durable) queue to send messages to. We will create a durable queue named `durable_q`. Run the following command:

```
$ qpidd-config add queue durable_q --durable
```

3. Run the following command to verify that the queue has been successfully created:

```
$ qpidd-config -b queues durable_q
```

4. You should see the queue you just created along with a binding to the direct exchange.

```
$ qpidd-config -b queues durable_q  
Queue 'durable_q'  
  bind [durable_q] => ''
```

5. Change to the examples directory you unzipped previously (e.g., `~/mrg-m/examples`).
6. Run the command: `./run.sh P2PSenderP.java`

```
[root@domU-12-31-38-04-CS-24 examples]# ./run.sh P2PSenderP.java  
Building class path...Done.  
Running example:  
Sending message: 1 GOOG 616.47  
Sending message: 2 RHT 42.46  
Sending message: 3 VMW 78.25  
Sending message: 4 CSCO 23.29  
Sending message: 5 IBM 141.43
```



7. The `P2PSenderP.java` code added five messages through the default exchange to the `durable_q` persistent queue.
 - a. The code for `P2PSenderP.java` is virtually identical the `P2PSender.java` code from the previous example. The only change is the queue name used in the JNDI properties file. This begins to illustrate some of the power of the AMQP model, namely that the broker abstracts the message producer from the underlying semantics of the messaging system via an exchange. In this case the queue is persistent.
8. Switch to the Terminal window running the Red Hat MRG Messaging server and shut it down by pressing `Ctrl+C`.
9. Start the MRG Messaging server again by typing the following command (as a normal user):

```
# /usr/sbin/qpid --auth=no
```

10. To verify that the messages were actually persisted, try to consume the messages by typing the following command: `./run.sh P2PReceiverP.java`

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh P2PReceiverP.java
Building class path...Done.
Running example:
Reading message: 1 GOOG 616.47
Reading message: 2 RHT 42.46
Reading message: 3 VMW 78.25
Reading message: 4 CSCO 23.29
Reading message: 5 IBM 141.43
```

11. The `P2PReceiverP.java` code connects to the queue on the server and retrieves messages until the control message "END" is received.



3.4 POINT-TO-POINT MESSAGES USING THE JAVA JMS API AND PYTHON CLIENT API

This next example illustrates how messaging can occur between clients with different languages. A Java client will generate and send five messages to a queue. Then, a Python client will connect and read those same five messages from the queue.

1. Open a new Terminal window.
2. First, we need a queue to use to send messages. We will create a queue bound to the default exchange. Run the following command:

```
# qpid-config add queue crosslang
```

3. Run the following command to verify that the queue has been successfully created:

```
# qpid-config -b queues crosslang
```

4. You should see the queue you just created along with a binding to the direct exchange.

```
# qpid-config -b queues crosslang  
Queue 'crosslang'  
  bind [crosslang] => ''
```

5. Change to the examples directory you unzipped previously (e.g., `~/mrg-m/examples`).



6. Now we are going to place some messages on the queue using a Java client.

Run the command: `./run.sh P2PCrossLangSender.java`

```
[root@rhel-5 examples]# ./run.sh P2PCrossLangSender.java
Running example:
Sending message: 1 GOOG 616.47
Sending message: 2 RHT 42.46
Sending message: 3 VMW 78.25
Sending message: 4 CSCO 23.29
Sending message: 5 IBM 141.43
[root@rhel-5 examples]#
```

7. Next we will get those messages from the queue, this time using a Python client.

Run the command: `./run.sh P2PCrossLangReceiver.py`

```
[root@rhel-5 examples]# ./run.sh P2PCrossLangReceiver.py
Running example:
1 GOOG 616.47
2 RHT 42.46
3 VMW 78.25
4 CSCO 23.29
5 IBM 141.43
[root@rhel-5 examples]#
```

8. If you were to open `P2PCrossLangReceiver.py` and inspect the code, you would see a similar pattern to receive messages as used in the Java JMS example `P2PReceiver.java`. However in this case, it would use Python as the language and the Red Hat Enterprise MRG Message Python client libraries instead.

3.5 PUBLISH/SUBSCRIBE MESSAGING WITH TOPICS USING THE PYTHON CLIENT API

This next example illustrates a publish-subscribe functionality using the default `amq.topic` exchange type. In these steps you will create a new queue and then bind the queue to the `amq.topic` exchange with a binding key named `stock.#`. You will then create a subscriber process that listens indefinitely for message on this queue. The publisher will publish messages using `stock.quote` as the subject (routing key). Since the routing key matches the binding key, the message will be routed to the queue, and then you will observe what the subscriber client outputs to the console.

1. Open a new Terminal window.



2. First, we need to create a queue to store our messages. Run the following command:

```
# qpid-config add queue stockprices_topic
```

3. Next, we will create a route by binding the `amq.topic` exchange (an out-of-the-box topic exchange available in every AMQP broker) with the queue created in the previous step, which uses the binding key named `stock.#`. Run the following command:

```
# qpid-config bind amq.topic stockprices_topic stock.#
```

4. Run the following command to verify that the exchange, queue, and binding has been successfully created:

```
# qpid-config -b exchanges amq.topic
```

5. You should see the queue you just created along with a binding to the direct exchange.

```
# qpid-config -b exchanges amq.topic  
Exchange 'amq.topic' (topic)  
  bind [stock.#] => stockprices_topic
```

6. Change to the examples directory you unzipped previously (e.g., `~/mrg-m/examples`).



7. Run the command: `./run.sh Subscriber.py`

```
[root@domU-12-31-38-04-CS-24 examples]# ./run.sh Subscriber.py
Building class path...Done.
Running example:
To end program, press Ctrl-c
█
```

8. Note that the subscriber will appear to hang as it waits for messages to be published. Now let us take a look at the client code for `Subscriber.py`:
 - a. First, we configure a connection to the broker, and then we start the connection.

```
connection = Connection(broker);
try:
    connection.connect()
```

- b. Next we create a session and generate a receiver handle:

```
session = connection.session()
receiver = session.receiver(address)
```

- c. Finally, loop indefinitely while receiving messages, print them to screen, and acknowledge receipt to the broker:

```
print "To end program, press Ctrl-c"

while True:
    try:
        message = receiver.fetch()
        print "Reading message " + message.content
        session.acknowledge(message)
    except Exception,m:
        print m
    finally:
        connection.close()
```

9. Open a new Terminal window



10. Change to the examples directory you unzipped previously (e.g., ~/mrg-m/examples)
11. Run the command: ./run.sh Publisher.py

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh Publisher.py
Building class path...Done.
Running example:
Sending message: 1 GOOG 616.47
Sending message: 2 RHT 42.46
Sending message: 3 VMW 78.25
Sending message: 4 CSCO 23.29
Sending message: 5 IBM 141.43
[root@domU-12-31-38-04-C5-24 examples]#
```

12. In the Terminal window running the subscriber client, you will notice that the screen now shows messages being received:

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh Subscriber.py
Building class path...Done.
Running example:
To end program, press Ctrl-c
Reading message: 1 GOOG 616.47
Reading message: 2 RHT 42.46
Reading message: 3 VMW 78.25
Reading message: 4 CSCO 23.29
Reading message: 5 IBM 141.43
```



13. Run `./run.sh Publisher.py` again. You will notice that the subscriber continues to receive messages.

```
[root@dc0mU-12-31-38-04-C5-24 examples]# ./run.sh Subscriber.py
Building class path...Done.
Running example:
To end program, press Ctrl-c
Reading message: 1 GOOG 616.47
Reading message: 2 RHT 42.46
Reading message: 3 VMW 78.25
Reading message: 4 CSCO 23.29
Reading message: 5 IBM 141.43
Reading message: 1 GOOG 616.47
Reading message: 2 RHT 42.46
Reading message: 3 VMW 78.25
Reading message: 4 CSCO 23.29
Reading message: 5 IBM 141.43
```

Note that `Publisher.py` and `Subscriber.py` differ after the session handle is created. The primary differences are that a 'sender' handle is created from the session, instead of a 'receiver' handle, and messages are sent rather than read:

```
session = connection.session()

sender = session.sender(address)

for i in range(len(messages)):
    message = str(i + 1) + " " + messages[i]
    print "Sending message: " + message
    sender.send(Message(message))
```



4 HANDS-ON: MESSAGING CLUSTERS

Let us now experiment with the same publish and subscribe messaging example used in the last example, but with messaging clusters. A messaging cluster is a group of brokers that act as a single broker. Changes on any broker are replicated to all other brokers in the same messaging cluster, so if one broker fails, its clients can fail-over to another broker without loss of state. The brokers in a messaging cluster may run on the same host or on different hosts. Two brokers are in the same cluster if:

1. they use the same Corosync mcastaddr, mcastport, and bindnetaddr, and
2. they use the same cluster name.

4.1 STARTING CLUSTERED MESSAGE SERVERS

The messaging clusters example uses two messaging services running on the same host. This section will show you how to get two Red Hat Enterprise MRG Messaging servers running in a cluster.

IMPORTANT: You must have first completed the setup and configuration in **Appendix B: Red Hat Enterprise MRG Messaging Clustering Setup** before proceeding.

Note: Before moving on, ensure that you do not have any previously running Red Hat Enterprise MRG Messaging servers running. If you do, switch to the Terminal window running the Red Hat Enterprise MRG Messaging server and press Ctrl+C.

1. Open a new Terminal window.
2. Change to the examples directory you unzipped previously (e.g., ~/mrg-m/examples).
3. Run the following command (as a normal user):

```
$ /usr/sbin/qpid -p5671 --auth=no --cluster-name=My_Cluster  
--data-dir=cluster1
```

4. The first messaging server will be started. If the server starts successfully, one of the last lines printed to the screen should tell you that it's running.

```
11:30:37 notice SASL disabled: No Aut  
11:30:37 notice Listening on TCP port  
11:30:37 notice Broker running  
11:30:47 warning Timer callback overr  
12-31-38-04-C5-24 -]#
```



5. Next, we will start up the second server in the cluster.
6. Open another new Terminal window.
7. Run the following command (as a normal user):

```
$ /usr/sbin/qpid -p5672 --auth=no --cluster-name=My_Cluster  
--data-dir=cluster2
```

8. The second messaging server will be started. If the server starts successfully, one of the last lines printed to the screen should tell you that it is running.

```
11:30:37 notice SASL disabled: No Aut  
11:30:37 notice Listening on TCP port  
11:30:37 notice Broker running  
11:30:47 warning Timer callback overr  
12-31-38-04-CS-24 ~]#
```

9. You are now ready to run the clustering examples.

4.2 PUBLISH/SUBSCRIBE MESSAGING WITH TOPICS USING THE PYTHON CLIENT API

This example illustrates Publisher/Subscriber functionality with a topic destination type over clustered brokers. In these steps you will create a topic to receive messages, then create two subscriber processes that listen indefinitely to your topic via a connection to broker two (port 5672). You then will generate messages and send them to the topic via a connection to broker one (port 5671).

1. Open a new Terminal window
2. Create a queue to store messages. Run the following command:

```
$ qpid-config add queue stockprices_cluster
```



3. Then, create a route by binding the `amq.topic` exchange (an out-of-the-box topic exchange available in every AMQP broker) with the queue created in the previous step, which uses the binding key named `stock.#`. Run the following command:

```
$ qpid-config bind amq.topic stockprices_cluster stock.#
```

4. Run the following command to verify that the exchange, queue, and binding has been successfully created:

```
$ qpid-config -b exchanges amq.topic
```

5. You should see the queue you just created along with a binding to the direct exchange.

```
$ qpid-config -b exchanges amq.topic  
Exchange 'amq.topic' (topic)  
  bind [stock.#] => stockprices_cluster
```

6. Change to the examples directory you unzipped previously (e.g., `~/mrg-m/examples`).
7. In the examples directory, run the command: `./run.sh SubscriberCluster.py`

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh SubscriberCluster.py  
Building class path...Done.  
Running example:  
To end program, press Ctrl-c
```

8. Open up yet another Terminal window for the second subscriber.



9. Change to the examples directory you unzipped previously (e.g., ~/mrg-m/examples).
10. In the examples directory, run the command: `./run.sh SubscriberCluster.py` again

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh SubscriberCluster.py
Building class path...Done.
Running example:
To end program, press Ctrl-c
```

11. Open up yet another Terminal window, this time for the publisher.
12. Change to the examples directory you unzipped previously (e.g., ~/mrg-m/examples).
13. Run the command: `./run.sh PublisherCluster.py`

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh PublisherCluster.py
Building class path...Done.
Running example:
Sending message: 1 GOOG 616.47
Sending message: 2 RHT 42.46
Sending message: 3 VMW 78.25
Sending message: 4 CSCO 23.29
Sending message: 5 IBM 141.43
```

14. You will notice that both subscriber screens now have message output.
15. Run `./run.sh PublisherCluster.py` again. You will notice that your subscribers continue to receive messages.

```
[root@domU-12-31-38-04-C5-24 examples]# ./run.sh SubscriberCluster.py
Building class path...Done.
Running example:
To end program, press Ctrl-c
1 GOOG 616.47
2 RHT 42.46
3 VMW 78.25
4 CSCO 23.29
5 IBM 141.43
```

Note that `PublisherCluster.py` and `SubscriberCluster.py` differ from `Publisher.py` and `Subscriber.py` only in the port number used to connect to the broker service. `Publisher.py` and `Subscriber.py` both use the default port number of 5672. `PublisherCluster.py` and `SubscriberCluster.py` use the ports 5671 and 5672 respectively, which are the ports that the two clustered message server instances are listening on.



APPENDIX A: INSTALLING RED HAT ENTERPRISE MRG MESSAGING ON RED HAT ENTERPRISE LINUX 6

To install Red Hat Enterprise MRG Messaging, you will need to have previously registered your system with Red Hat Network. Read more about how to register your system with Red Hat Network at access.redhat.com/knowledge/docs/Red_Hat_Network/Reference_Guide/s1-up2date-register.html

Once registered with Red Hat Network, you have to configure your system to have access to a few additional channel entitlements. Using your favorite web browser navigate to rhn.redhat.com and log in using your Red Hat Network credentials. Once logged in, navigate to the system on which you will be installing Red Hat Enterprise MRG Messaging. From your system's main page, select **Alter Channel Subscriptions** in the **Subscribed Channels** section.

This will take you to the **Software >> Software Channels** page. From this page expand the **Release Channels** for **Red Hat Enterprise Linux 6** for **x86_64** under **Software Channel Subscriptions** and select the following channels:

- RHEL Server High Availability (v. 6 for 64-bit x86_64)
- RHEL Server Optional (v. 6 64-bit x86_64)

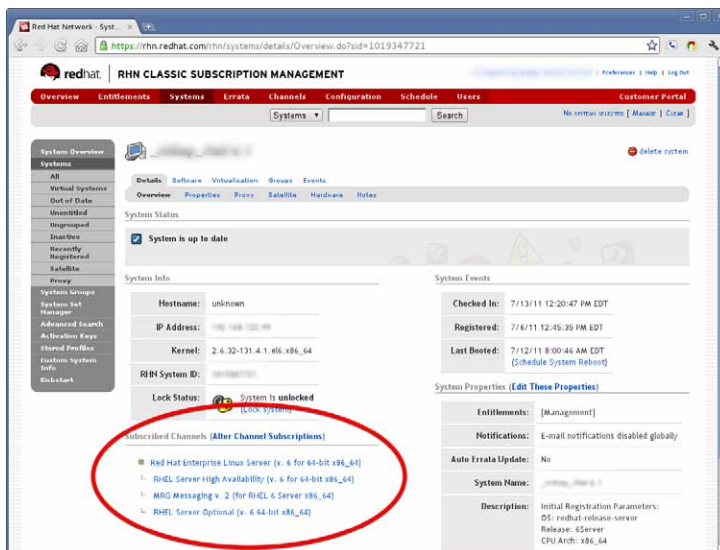
Next, expand the **Additional Services Channels** for **Red Hat Enterprise Linux 6** for <32 or 64 bit> under **Software Channel Subscriptions** and select the following channel:

- MRG Messaging v. 2 (for RHEL 6 Server x86_64)

Note: In the above example, the two packages selected were for the Red Hat Enterprise Linux 6 64-bit version. If your system is 32-bit then the names will indicate 32-bit instead.

Once selected, click the **Change Subscriptions** button at the bottom of the page.

Your system's main page should now look like the following (the three channels selected above should now be listed in the **Subscribed Channels** section):





The Red Hat Network configuration is now complete. Next, install the software packages on your Red Hat Enterprise Linux 6 system:

1. Install the Red Hat Enterprise MRG Messaging group using the yum command. In a Terminal window as root user (or sudo):

```
# yum install @mrq-messaging
```

2. Install the Qpid Tools for managing Red Hat Enterprise MRG Messaging via the command line using the yum command. In a Terminal window as root user (or sudo) run:

```
# yum install qpid-tools
```

3. To run the Java examples in this Getting Started guide, you need to install a Java Development environment as well. In a Terminal window as root user (or sudo) run:

```
# yum install java-1.6.0-openjdk-devel
```

You are now ready to continue with the hands-on examples.



APPENDIX B: RED HAT ENTERPRISE MRG MESSAGING CLUSTERING SETUP

In order to run the example in the Hands-On: Messaging Clusters section you must first perform some setup and configuration so the clustered network communication between multiple Red Hat Enterprise MRG Messaging servers will function properly.

1. The first task is to install the Red Hat Enterprise MRG Messaging Clustering support using the yum command. In a Terminal window as root user (or sudo) run:

```
# yum install qpid-cpp-server-cluster
```

2. You also need to create the /etc/corosync/corosync.conf configuration file. Corosync is a service used by Red Hat Enterprise MRG Messaging for its clustering support. For the purpose of this Getting Started Guide the quickest way to do this is to copy the provided example configuration. In a Terminal window as root user (or sudo) run:

```
# cp /etc/corosync/corosync.conf.example /etc/corosync/corosync.conf
```

3. Then modify the binding address of the Corosync configuration file with your machines network address. The following few steps will detail how to accomplish this.
4. Use ifconfig to find your machine's inet addr and the netmask for the interface you want to configure (in our example we are using the eth0 interface). Run the following command:

```
$ /sbin/ifconfig
```

```
eth0 Link encap:Ethernet HWaddr 00:E0:81:76:B6:C6
      inet addr:10.16.44.222 Bcast:10.16.47.255 Mask:255.255.248.0
      inet6 addr: fe80::2e0:81ff:fe76:b6c6/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:35914541 errors:6 dropped:0 overruns:0 frame:6
      TX packets:6529841 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:20294124383 (18.9 GiB)
      TX bytes:12925473031 (12.0 GiB)
      Interrupt:98 Base address:0x8000
```



5. The binding address used in `/etc/corosync/corosync.conf` should be the network address for the interface, which you can find by doing a bitwise AND of the inet addr (in this case, 10.16.44.222) and the network mask (in this case, 255.255.248.0). The result is 10.16.40.0.

Tip: You can use the following online utility to help calculate the network address:
www.subnetonline.com/pages/subnet-calculators/ip-subnet-calculator.php

Just enter your IP Address and your Subnet Mask and click Calculate. The value you need will be in the Network Address field.

6. As a sanity check, you can use `route` and make sure the network address you computed is associated with this interface (the network address should be listed as one of the Destination values):

```
# /sbin/route
```

```
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
20.0.10.0 * 255.255.255.0 U 0 0 0 eth1
192.168.122.0 * 255.255.255.0 U 0 0 0 virbr0
10.16.40.0 * 255.255.248.0 U 0 0 0 eth0
169.254.0.0 * 255.255.0.0 U 0 0 0 eth1
default 10.16.47.254 0.0.0.0 UG 0 0 0 eth0
```

```
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
20.0.10.0 * 255.255.255.0 U 0 0 0 eth1
192.168.122.0 * 255.255.255.0 U 0 0 0 virbr0
10.16.40.0 * 255.255.248.0 U 0 0 0 eth0
169.254.0.0 * 255.255.0.0 U 0 0 0 eth1
default 10.16.47.254 0.0.0.0 UG 0 0 0 eth0
```

7. To use `eth0` as the interface for the cluster, edit the file `/etc/corosync/corosync.conf` and find the setting for `bindnetaddr`. Then set it to the network address computed above (you must be root to edit this file):

```
totem {
    ...
    interface {
        ...
        # NOTE: Be sure to use your network address.
        bindnetaddr: 10.16.40.0
        ...
    }
}
```



8. Since we will be running the messaging broker under your local user/group account we need to let Corosync know what that user/group account name is. Edit the file `/etc/corosync/uidgid.d/qpid`. Specify your uid and gid. (You must be root to edit this file.)

```
uidgid {  
    uid: <your user id>  
    gid: <your group id (usually the same as uid)>  
}
```

9. Corosync requires some additional firewall modifications to work properly in your environment. For the purposes of this Getting Started Guide, however, we are simply going to turn off the firewall. In a Terminal window as root user (or sudo) run:

```
# /sbin/service iptables save  
# /sbin/service iptables stop
```

Note: When you have completed this Getting Started Guide, you can turn the firewall back on via `service iptables start` as root user (or sudo).

10. Last, you need to start Corosync. In a Terminal window as root user (or sudo) run:

```
# /sbin/service corosync start
```

You are now ready to continue with the Hands-On Messaging Clusters example.

RED HAT SALES AND INQUIRIES

NORTH AMERICA
1-888-REDHAT1
www.redhat.com
sales@redhat.com

**EUROPE, MIDDLE EAST
AND AFRICA**
00800 7334 2835
www.europe.redhat.com
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
www.apac.redhat.com
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
www.latam.redhat.com
info-latam@redhat.com