Red Hat Reference Architecture Series

Getting Started with MRG-M

Version 1.0
December 2010

Red Hat Cloud Foundations Reference Architecture
Edition One: Getting Started with MRG

Red Hat Reference Architecture Series

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

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

Table of Contents

1 Executive Summary

Even 30 years into the digital age, ever-growing demand is being placed on systems. New methods of delivery and sustenance are being tested on an almost weekly basis. CPU clock time is now almost too slow, and business, even in times of downturn, marches on with a steady pressure to achieve and out-perform the competition. Automation of business process, our chosen profession, is ripe with new and challenging opportunities to solve repetitive and daunting problems.

In this ecosystem, stabilizing the inter-operation of business systems with different agendas has been the focus of not a few CIO's and operations folks over the last several decades. As systems become more "cloudy", real-time connections between service providers and subscribers are destined to become the norm, rather than the occasional exception. When you combine the cloud with SOA, the requirement that sequestered systems find a way to reach out to each other becomes part of the very fabric of corporate operations.

Although clouds and service-oriented methodology consume the real estate of most IT white boards of every large organization in the country, some older concepts are still being deployed as new code across all sectors of the marketplace. One of those concepts is the idea of using event-based messaging as a platform to achieve parity of data across systems, both inside the firewall and outside it. In today's digital-biz world, messaging is still a business-critical function, and the middleware to support it still consumes respectable slices of the budget.

Short transactions are the lifeblood of integrity for a host of business systems.

As a social phenomenon, newly-formed digital cultures are twittering their way through everyday life, sharing short messages that keep the connection-hungry apprised of every move each other makes. In this same vein, short transactions are the lifeblood of integrity for a host of business systems, from financial to manufacturing, to entertainment, government and most others. Adoption of this form of loose-but-dependent connectivity between systems will only increase as new and improved software architectural designs become part of daily life.

1.1 The history of messaging

In the mid-90's, It had become apparent to system implementers that reliable integration of business systems was best achieved by capturing relevant business events, and broadcasting those events to appropriate systems that would consume the event and respond according to that system's business rules. Asynchronous messaging architecture was found to be effective in providing reliable delivery of events wrapped in messages between digital stakeholders.

The software that supports the paradigm, often called middleware, has and continues to be important due to its natural support for loosely coupled asynchronous event driven architectures. Support of multi-tiered complex transactions makes middleware a good choice for Financial, Stocks, Telecomm, Manufacturing, Aerospace and Defense, Medicine and Transportation. Traditional commercial middleware vendors such as IBM, Oracle (BEA), Sonic and TIBCO continue to sell and improve the prospect of interoperation with somewhat proprietary influences.

Over the last two decades, asynchronous messaging has solidified itself as the standard for application integration. However, there are several nagging issues with messaging that eventually made a new standard necessary for the industry:

- Integration is often 10-30% of every IT effort
- Proprietary middleware can be a source of vendor lock-in
- Existing middleware is too language-specific
- Wire-level interoperability is missing
- Interoperability remains more difficult than it should be
- Change is still the enemy

For example, in financial services, . Banks must connect with an expanding network of customers and third parties, and as the sheer numbers of relationships and transactions increases, complexity has become a way of life. In today's ecosystem, the names change about as fast as the Hang Seng Index.

In many businesses, achieving 100% reliable conversational clarity is job one.

On method of dealing with the need to interact with ever-changing partners and customers is to develop proprietary messaging formats, driving costs upward. Banks individually shoulder the cost of supporting proprietary architectures, and their clients support discrete communication tools across multiple relationships.

With the advent of new regulations, pressure mounts to simplify payments infrastructures. Large companies are particularly keen to find smarter ways to navigate the financial supply chain, including making it cheaper to move money to and from bank accounts. Electronic invoice presentment and payment makes streamlining connectivity with multiple service providers an issue.

Companies now need a more ubiquitous and secure means of transmitting business messages without using proprietary formats and technologies. In short, they want to process financial transactions with their partners as simply as they might tweet the babysitter, but with security and higher predictability of delivery. We need a new standard, one that will allow us to transmit authenticated financial events without resorting to multiple layers of bit manipulation.

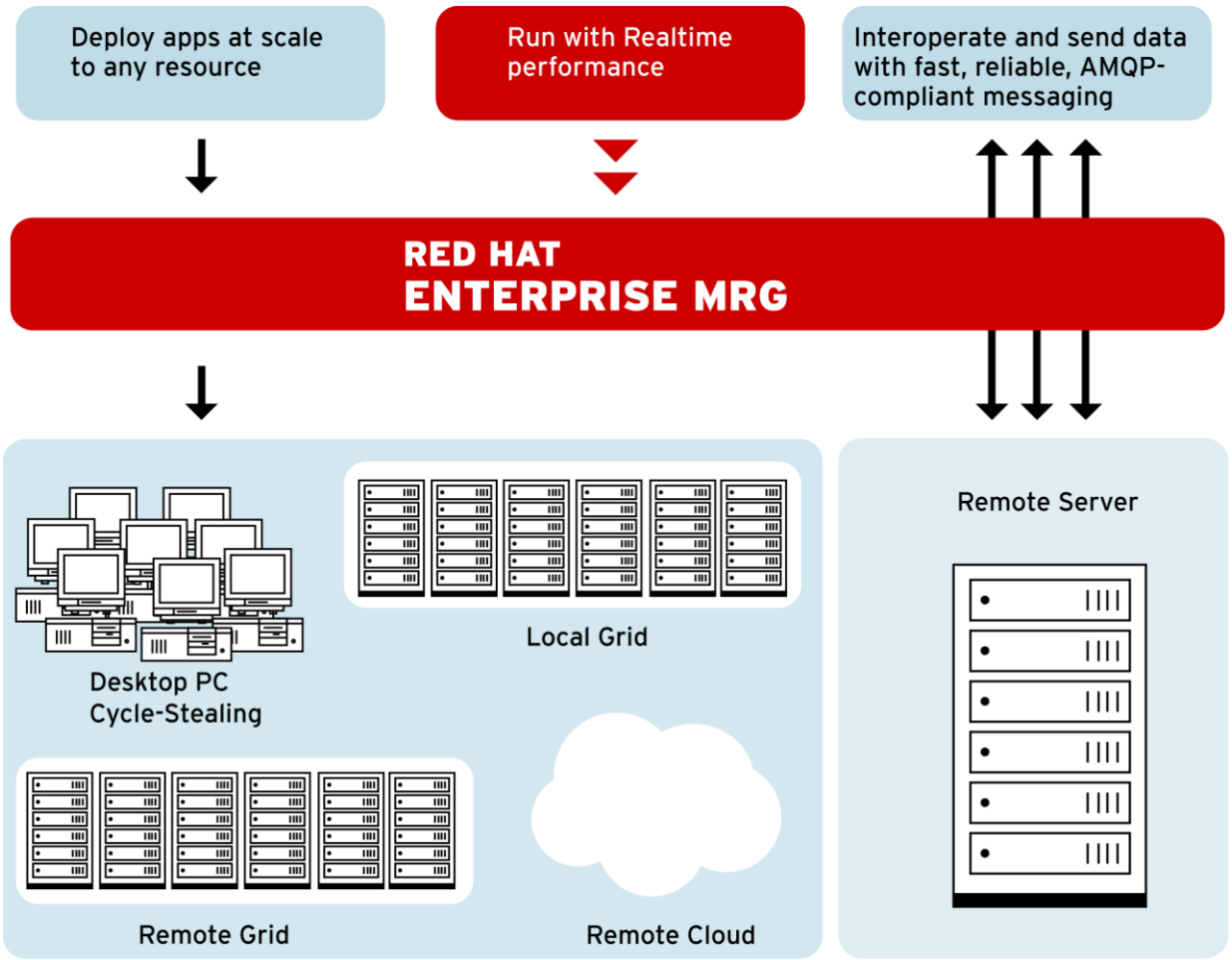
1.2 The next generation

Now arrives the **Advanced Message Queuing Protocol** (AMQP) which is intended to help resolve some of these issues, and used as the basis for new set of products, integrated into existing systems, with enhanced interoperability for common APIs, including JMS.

AMQP can be used with most of the current messaging and Web Service Specifications, such as JMS, SOAP, WS-Security and WS-Transactions, and provides specified routing to and from multicast for subnet optimizations or grid deployments.

By complying with the AMQP standard, middleware products written in several languages for different platforms can share messages with one another. AMQP addresses the challenge of reliable transport of valuable messages across and between business partners in near real-time.

As part of Red Hat's Enterprise MRG suite, MRG-M messaging is our implementation of the AMQP standard.



2 What is MRG-M?

JPMorgan Chase was a financial industry leader that decided to find a better way. Rather than implementing yet another proprietary messaging system, the company decided to sponsor an approach that could be replicated throughout the industry, and accepted as the benchmark. Eventually AMQP was created as the network protocol for the solution.

The new protocol had to be simple and language neutral. To achieve this, JPM decided that it must be ubiquitous, and easily adopted everywhere. AMQP is an open Internet protocol for business messaging which 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. MRG-M is Red Hat's reference implementation of AMQP.

2.1 The MRG-M Model

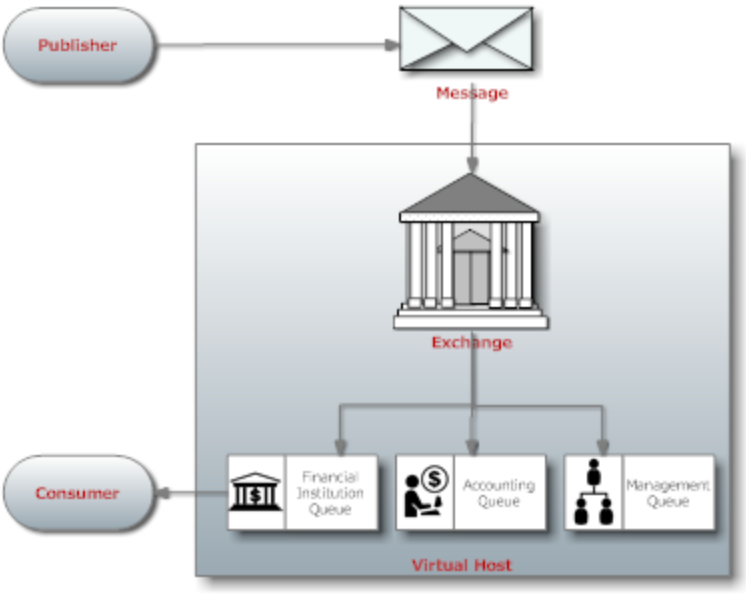
The MRG-M model explicitly defines a server's semantics because interoperability demands the same semantics for any server implementation. The model specifies a modular set of components and standard rules for connecting these components. It emulates the classic messaging concepts of store-and-forward 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 which 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 provides the message routing criteria

You can think of an MRG-M 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 MRG-M Model 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

MRG-M supports a variety of message queues, including private or shared, durable or transient, 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 these 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 these to a single subscriber. Subscription queues are typically temporary, and private to one subscriber.

There is no formal definition of these queues in MRG-M: they are simply examples of how message queues can be defined. According to the specification, it should be trivial to create new entities such as durable, shared subscription queues, and those with persistence.

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 MRG-M was to include explicitly-defined semantics supporting multiple routing models. Therefore, complex routing is well-supported in MRG-M.

Part of the lure of MRG-M 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.

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

2.2 The MRG-M Wire-level Format

The MRG-M 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 MRG-M 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 that carries these methods from application to server, and back, and which handles channel multiplexing, framing, content encoding, heart-beating, data representation, and error handling. Both the transport layer & high-level layers are pluggable, which allows evolution of the protocol and the adoption of emerging technologies.

According to the specification, the MRG-M 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 extended 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

2.3 Messages in MRG-M

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 MRG-M 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

MRG-M supports both of these message types.

MRG-M 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

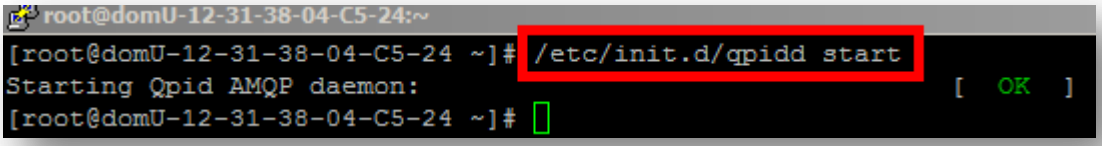
3 Hands-On: Single Broker

The examples below are designed to demonstrate three basic uses of the messaging server: point-to-point messaging, publisher/subscriber messaging, and persistent messaging.

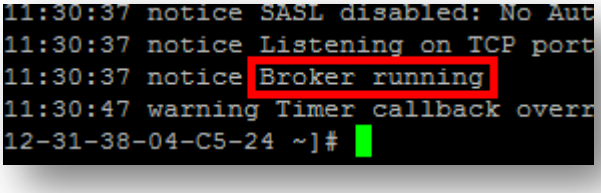
3.1 Starting the Message Server

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

1. Open a Putty terminal.
2. Run the command: /etc/init.d/qpidd start



3. This will start the messaging server. If the server started successfully, one of the last lines in the log file (/var/lib/qpidd/daemon.log) should read ‘Broker running’.



4. You are ready to try one of the example programs. Minimize the Qpid broker window. Make sure not to close it or none of the examples will work.

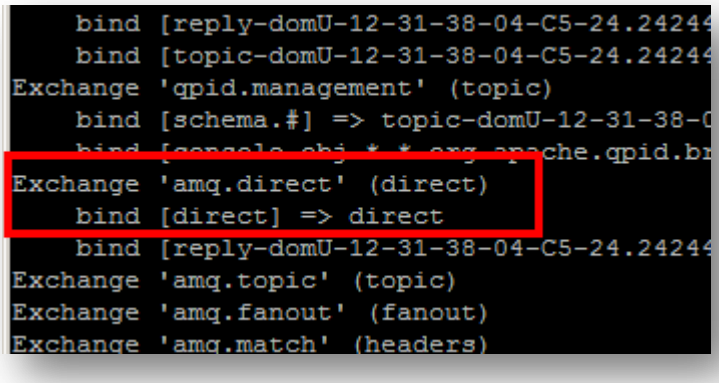
3.2 Point-to-Point Messages with Java

This example illustrates point-to-point functionality, or the queue destination type. In these steps you will first run a command that populates a queue on the server with 5 messages. Then you will run a command that reads all available messages on the queue and prints them to the screen.

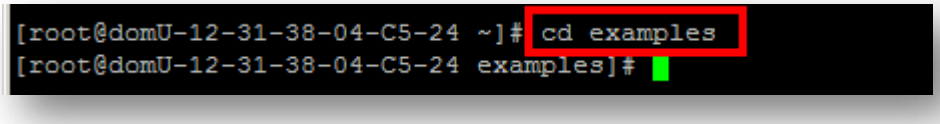
1. Open a Putty session to the server, as described in ‘Connect to a Host using Putty’.
2. First we need a queue to use to send messages. We will create a queue for the default direct exchange ‘amq.direct’. Run the following commands:
   a. `qpid-config add queue direct`

b. `qpid-config bind amq.direct direct direct`

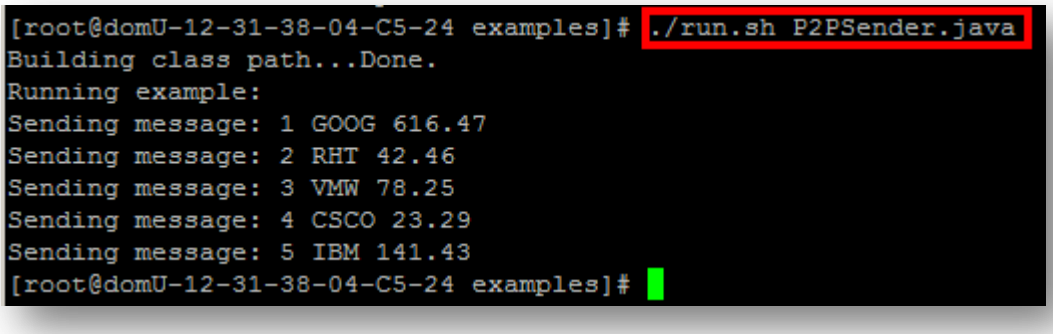
3. Run the following command to verify that the queue has been successfully created: qpid-config -b exchanges
4. You should see a binding from the 'amq.direct' exchange to the 'direct' queue.



5. Run the command: `cd examples`



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



7. The `P2PSender.java` code added 5 messages through the `amq.direct` exchange. If you're familiar with the JMS API, the following code examples will be second nature.

    a. First, Java uses the JNDI API to locate and configure the messaging interface at runtime. This can be seen on lines 26-36. JNDI is used to get a connection factory, then to generate the actual connection handle to the messaging server, and it gets a handle to the specific queue on the messaging server to which messages will be sent.

```
/* START JNDI configuration */

Properties props = new Properties();
props.setProperty("java.naming.factory.initial", "org.apache
props.setProperty("connectionfactory.host", "amqp://guest:gu
props.setProperty("queue.name", "queue");

ctx = new InitialContext(props);
factory = (QueueConnectionFactory) ctx.lookup("host");
queue = (Queue) ctx.lookup("name");

/* END JNDI configuration */
```

b. Once the factory and queue handles are established, the connection can be started.

```
/* START messaging code */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
sender = session.createSender(queue);
message = session.createTextMessage();
```

c. From the connection, a session must be obtained. All message transmission must occur within the context of a session.

```
/* START messaging code */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
sender = session.createSender(queue);
message = session.createTextMessage();
```

d. From the session a sender or a reciever handle is obtained. Since we will only send messages with this particular command, we need only create a sender object.

```
/* START messaging code */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
sender = session.createSender(queue);
message = session.createTextMessage();
```

e. The last initialization task is creating a message object to transmit. We can use the same object to send multiple messages, by changing the contents of the object, so we need only create one.

```
/* START messaging code */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
sender = session.createSender(queue);
message = session.createTextMessage();
```

f. Finally, we can transmit messages. Here, we set the contents of the message object and transmit the message once for each string we wish to transmit.

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

/* send a control message to signal termination */
message.setText("END");
sender.send(message);
```

g. To signal the end of a sequence of messages, we send a control message. This control message is nothing special from the messaging server's perspective. The receiving client is simply programmed to terminate when a text message of value "END" is received.

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

/* send a control message to signal termination */
message.setText("END");
sender.send(message);
```

8. Now that messages are waiting on the server, we can run the command to retrieve messages. 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 connected to the `amq.direct` exchange on the server and retrieved messages until the control message “END” was received.
    a. The sender and receiver commands are almost identical until this point. Here in the sender, we created a sender object. In the receiver, as one would think, we are creating a receiver object instead.

```
/* receive messages */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
receiver = session.createReceiver(queue);
conn.start();
```

    b. Then, as required by the API, the connection is started in order to signal that messages are about to be read.

```
/* receive messages */
conn = factory.createQueueConnection();
session = conn.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
receiver = session.createReceiver(queue);
conn.start();
```

    c. Finally, the command loops indefinately until the control message “END” is received.

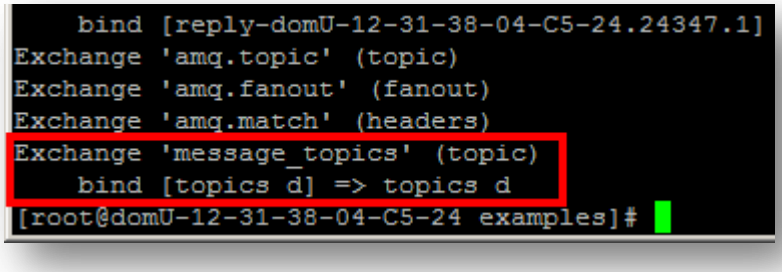
```
while (true) {
      Message m = receiver.receive(1);

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

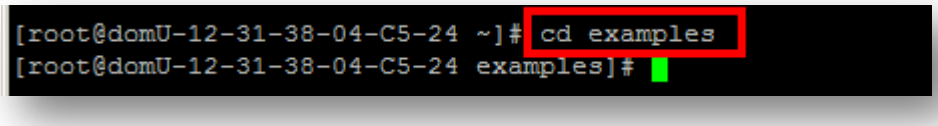
3.3 Persistent Point-to-Point Messages with Java

This example illustrates point-to-point functionality with persistence. These steps are identical to the precious Point-to-Point Messaging example except that here the code has been pointed to a durable (persistent) queue, and we will restart the broker between sending and receiving messages.

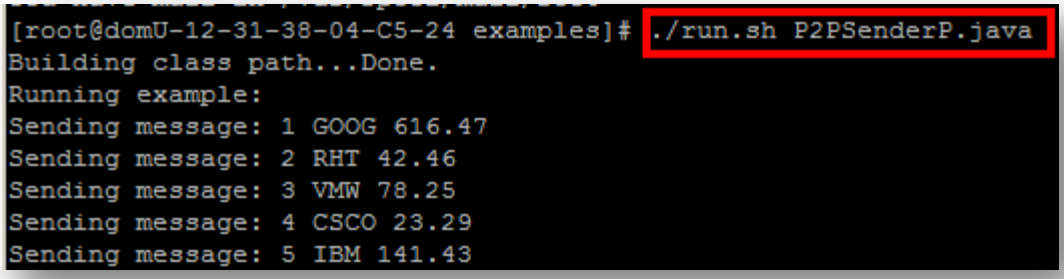
1. Open a Putty session to the server, as described in 'Connect to a Host using Putty'.
2. Run the following command to verify that the persistent queue exists: qpid-config -b exchanges
3. You should see a binding from the 'message_topics' exchange to the 'topics_d' queue.



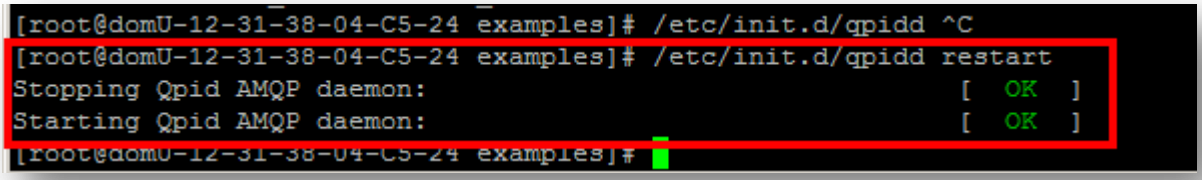
4. Run the command: `cd examples`



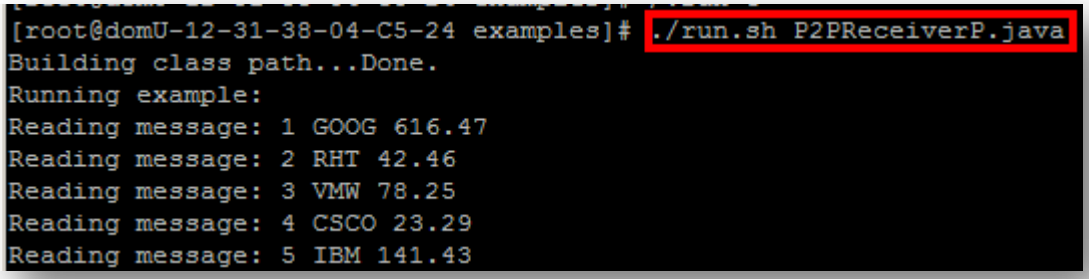
5. Run the command: `./run.sh P2PSenderP.java`



6. Restart the broker: `/etc/init.d/qpidd restart`



7. To verify that the messages still exist, run the command: `./run.sh P2PReceiverP.java:`



3.4 Publisher/Subscriber Messages with Python

This project illustrates Publisher/Subscriber functionality, or the topic destination type. We will not need to create a queue for this example because one will be created automatically and bound to the `amq.topic` exchange. In these steps you will create subscriber process which listens indefinitely to a topic queue. Then you will generate messages and send them to the topic queue.

1. Open a Putty session to the server, as described in ‘Connect to a Host using Putty’.

2. Change to the `examples` directory.

```
[root@domU-12-31-38-04-C5-24 ~]# cd examples
[root@domU-12-31-38-04-C5-24 examples]#
```

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

```
[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
```

4. Note that the subscriber will appear to hang as it waits for messages to be published.
    a. First we configure a connection to the broker, and then we start the connection.

```
connection = Connection(broker);

try:
  connection.connect()
  session = connection.session()
  receiver = session.receiver(address)
```

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

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

print "To end program, press Ctrl-c"

while True:
  try:
    message = receiver.fetch()
```

c. Finally, we loop indefinately while we receive 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 message.content
    session.acknowledge(message)
  except BaseException:
    break
```

5. Open a second Putty session to the server.
6. Again, go to the `examples` directory.

```
[root@domU-12-31-38-04-C5-24 ~]# cd examples
[root@domU-12-31-38-04-C5-24 examples]#
```

7. 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]#
```

8. You will notice that the subscriber screen now has output:

```
[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
```

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

```
[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
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
```

10. `Publisher.py` and `Subscriber.py` mainly differ after the session handle is created. The only differences is that a ‘sender’ handle is created from the session, instead of a ‘receiver’ handle, and messages are sent, instead of 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))
```

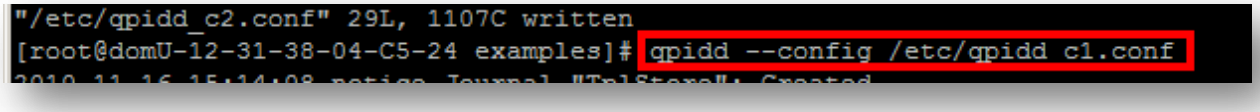
4 Hands-On: Clustered Brokers

Let's now demonstrate publisher/subscriber messaging over clustered brokers.

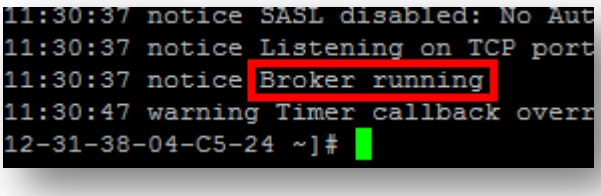
4.1 Starting Clustered Message Servers

Clustered broker examples require two messaging services to be running. This section will show you how to get the message servers running.

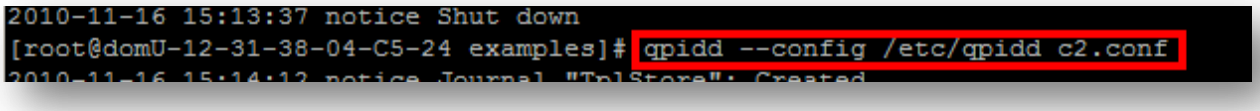
1. Open a Putty terminal.
2. Run the command: `qpidd --config /etc/qpidd_c1.conf`



3. This will start one messaging server. If the server started successfully, one of the last lines printed to the screen should tell you that it's running.



4. Minimize the Qpid broker window. Make sure not to close it or none of the example will work.
5. Open another Putty terminal.
6. Run the command: `qpidd --config /etc/qpidd_c2.conf`



7. This will start a second messaging server. If the server started successfully, one of the last lines printed to the screen should read ‘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 ~]#
```

8. Minimize the second Qpid broker window. Make sure not to close it or none of the example will work.
9. You are now ready to run the clustering example.

4.2 Clustered Publisher/Subscriber Messages with Python

This example illustrates Publisher/Subscriber functionality or the topic destination type over clustered brokers. We will not need to create a queue for this example because one will be created automatically and bound to the `amq.topic` exchange. In these steps you will create subscriber process which listens indefinitely to a topic queue via a connection to broker 2 (`/etc/qpidd_c2.conf`). Then you will generate messages and send them to the topic queue via a connection to broker 1 (`/etc/qpidd_c1.conf`).

1. Open a Putty session to the server, as described in ‘Connect to a Host using Putty’.
2. Change to the `examples` directory.

```
[root@domU-12-31-38-04-C5-24 ~]# cd examples
[root@domU-12-31-38-04-C5-24 examples]#
```

3. 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
```

4. Note that the subscriber will appear to hang as it waits for messages to be published.
5. Open a second Putty session to the server.

6. Change to the `examples` directory.

```
[root@domU-12-31-38-04-C5-24 ~]# cd examples
[root@domU-12-31-38-04-C5-24 examples]#
```

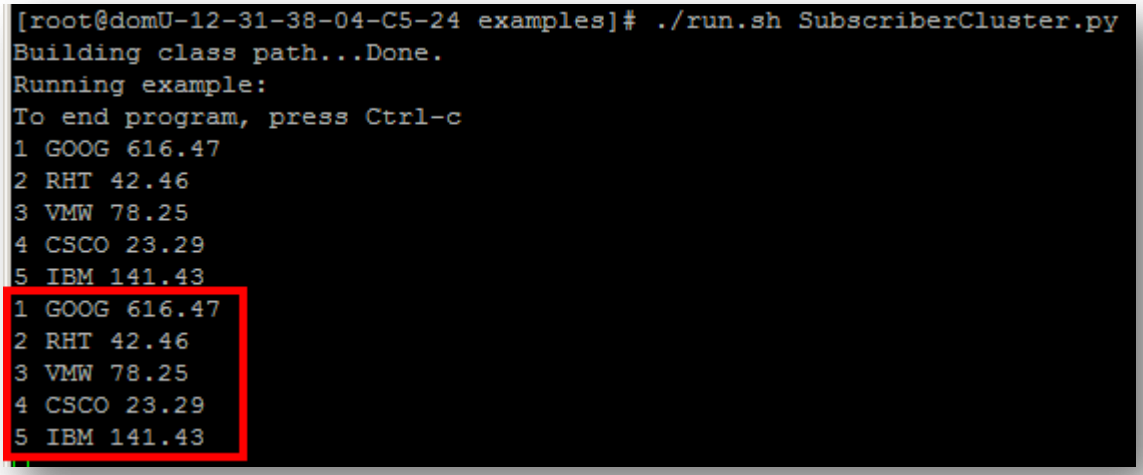
7. Run the command: `./run.sh PublisherCluster.py`

```
[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
```

8. You will notice that the subscriber screen now has output.

```
[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
```

9. Execute `./run.sh PublisherCluster.py` again. You will notice that the subscriber continues to receive messages!



10. `PublisherCluster.py` and `SubscriberCluster.py` differ from `Publisher.py` and `Subscriber.py` respectively 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 5670 and 5671 respectively, which are the ports on which the two clustered message server instances are listening.

5 Appendices

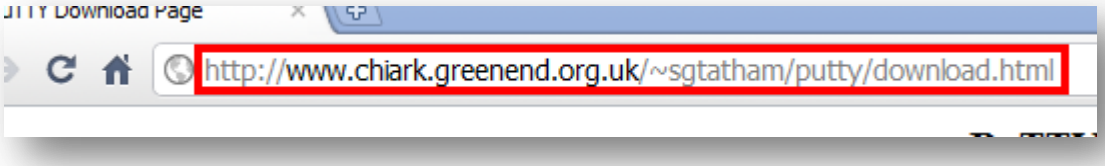
Appendix A: Installing Putty

Before you can try any of the examples, you must be able to connect to a terminal on the server. This is accomplished by making a connection over SSH using Putty—a remote terminal client. This demonstration shows how to set up a connection to Amazon's EC2 cloud.

5.2 Putty for SSH

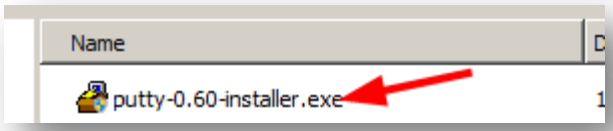
This section will guide you through the process of downloading and installing Putty.

1. Goto http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

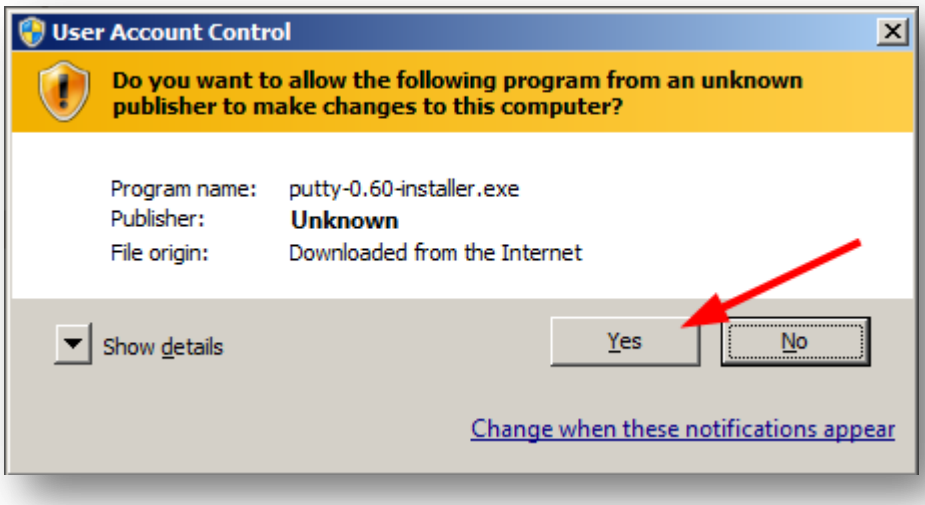


2. Download the putty installer. (At the time this guide was written the latest installer was putty-0.60-installer.exe)

3. Run the installer.

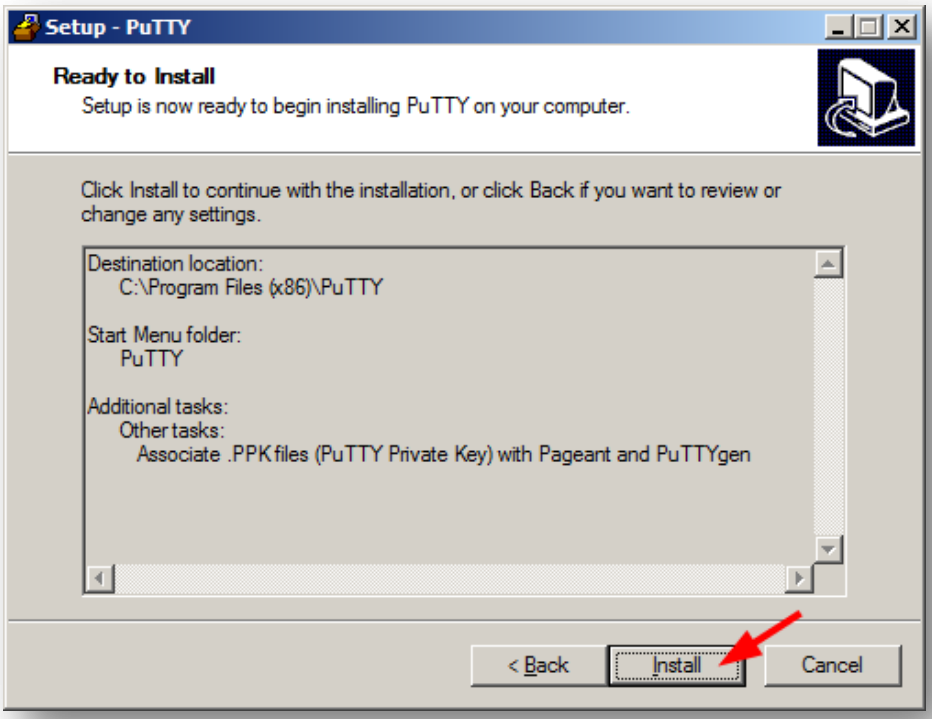


4. If prompted by Windows Vista or Windows 7 to approve the installation, click 'Yes'.

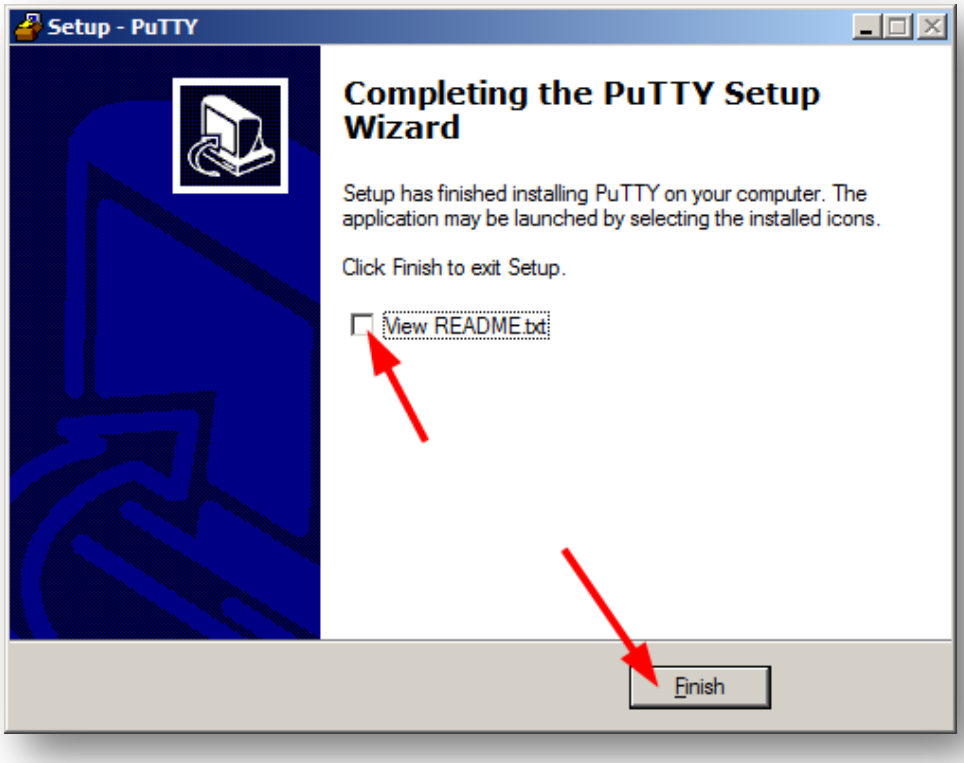


5. Proceed approving all the default settings by clicking ‘Next’ on each of the installation screens.

6. Finally, click ‘Install’ to begin the install process.



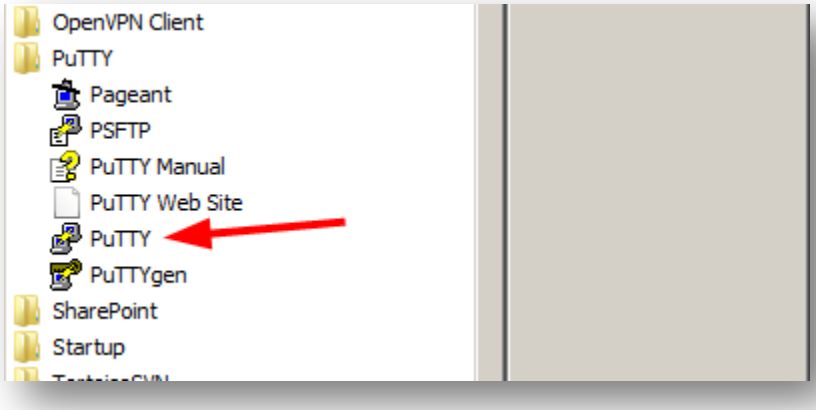
7. Once the installation completes, remove the check mark next to ‘View README.txt’ and then click ‘Finish’.



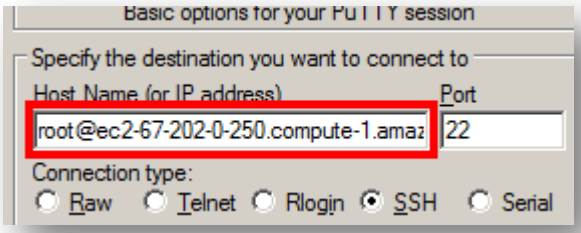
5.3 Configuring a Connection in Putty

This section will guide you through the process of configuring a connection (which Putty calls a session). This includes referencing a private key file used for authentication.

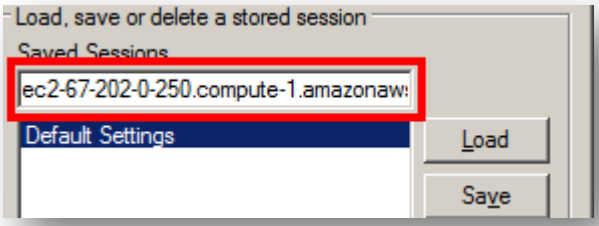
1. Open Putty from your start menu.



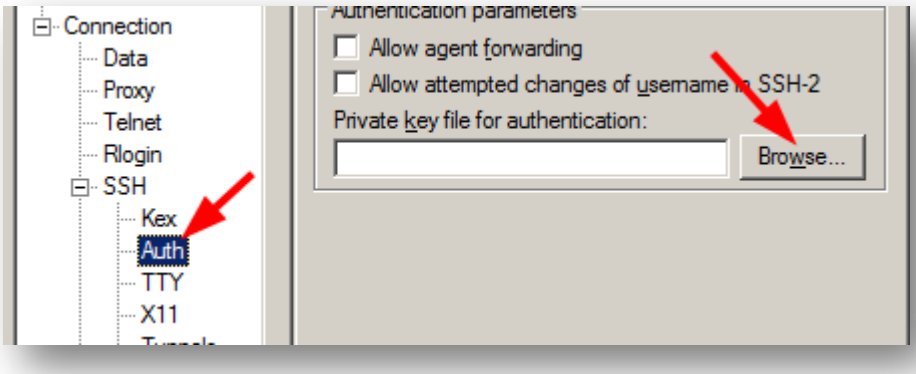
2. Enter your username and host name in the ‘Host Name’ box. Below, we have entered a username of ‘root’ and a hostname of ‘ec2-67-202-0-250.compute-1.amazonaws.com’. Notice the ‘@’ sign between the username and host name.



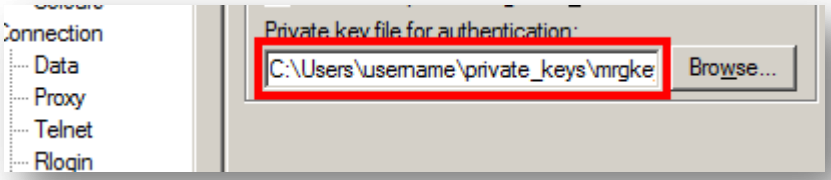
3. Name your connection. A common practice is to simply use the name of the host name here.



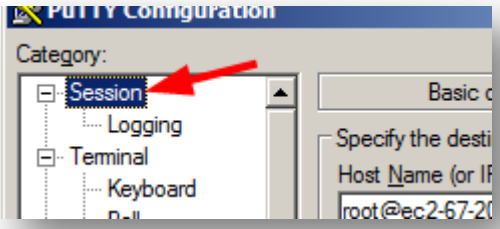
4. If you have a private key to use for authentication (file ends with .ppk extension), then do the following.
    a. In the configuration navigation tree on the left select Connection->SSH->Auth, then click ‘Browse’.



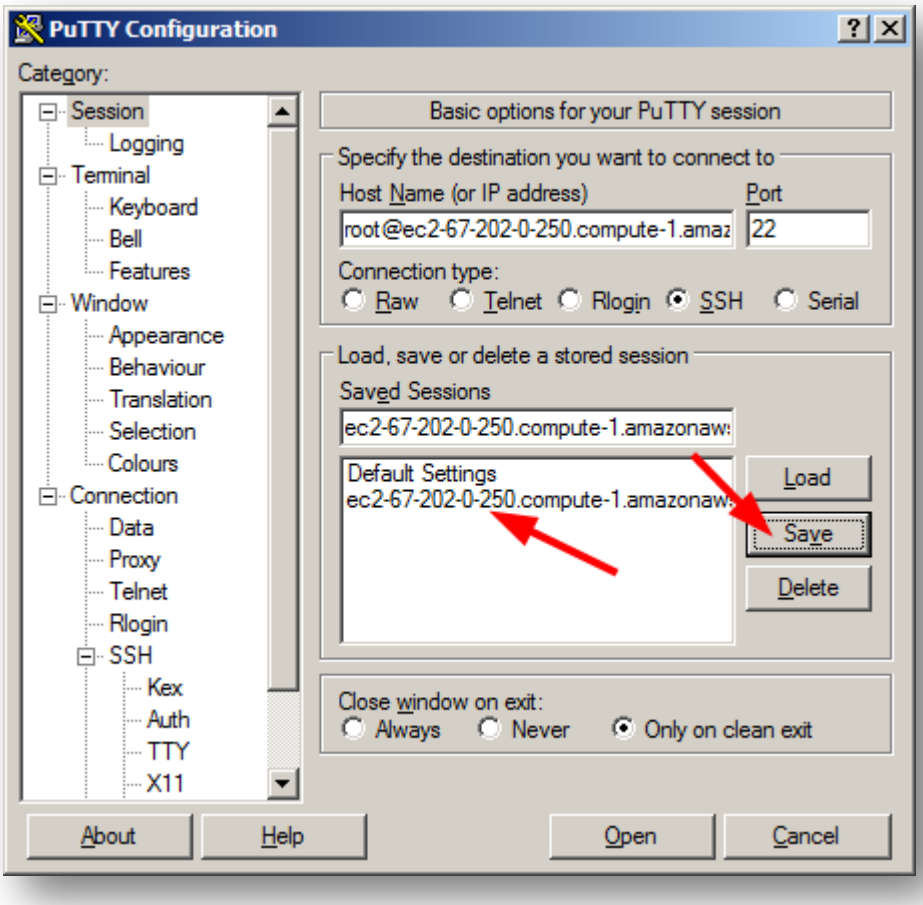
    b. Once you have selected the correct private key file, it will be listed in the box.



c. Return to ‘Session’, at the top of the configuration navigation tree on the left.



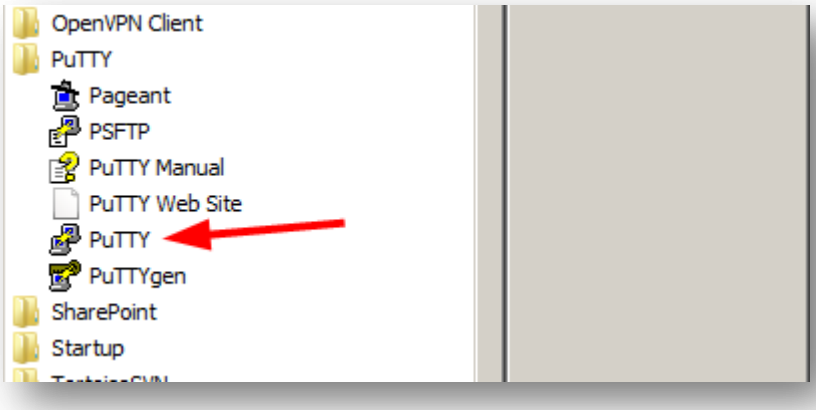
5. Click ‘Save’ to store your configuration in the ‘Saved Sessions’ list.



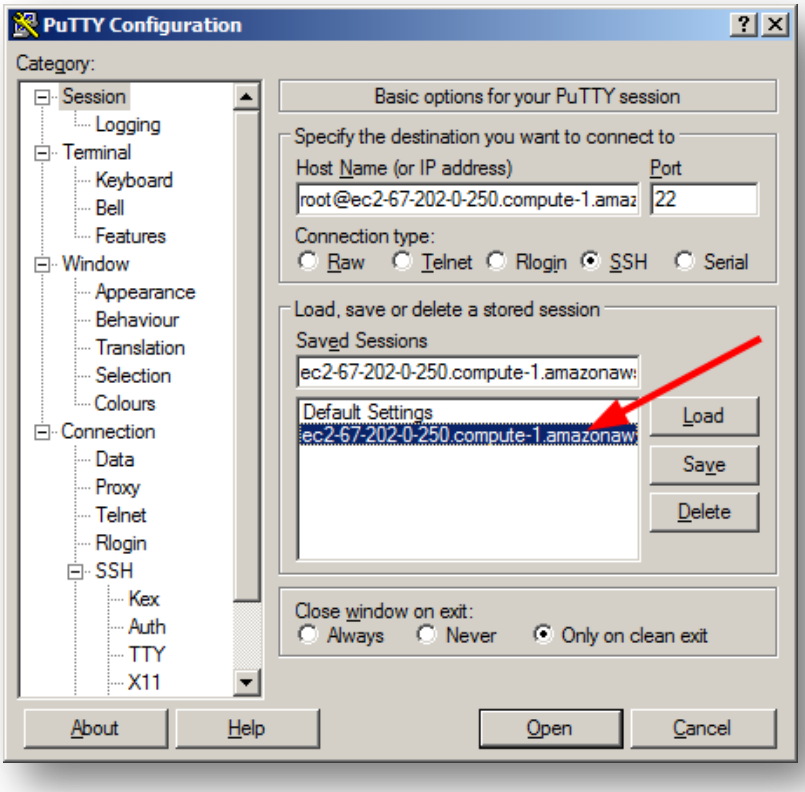
5.4 Connect to a Host using Putty

This section will guide you through the process of connecting to a server using a previously saved session (connection configuration).

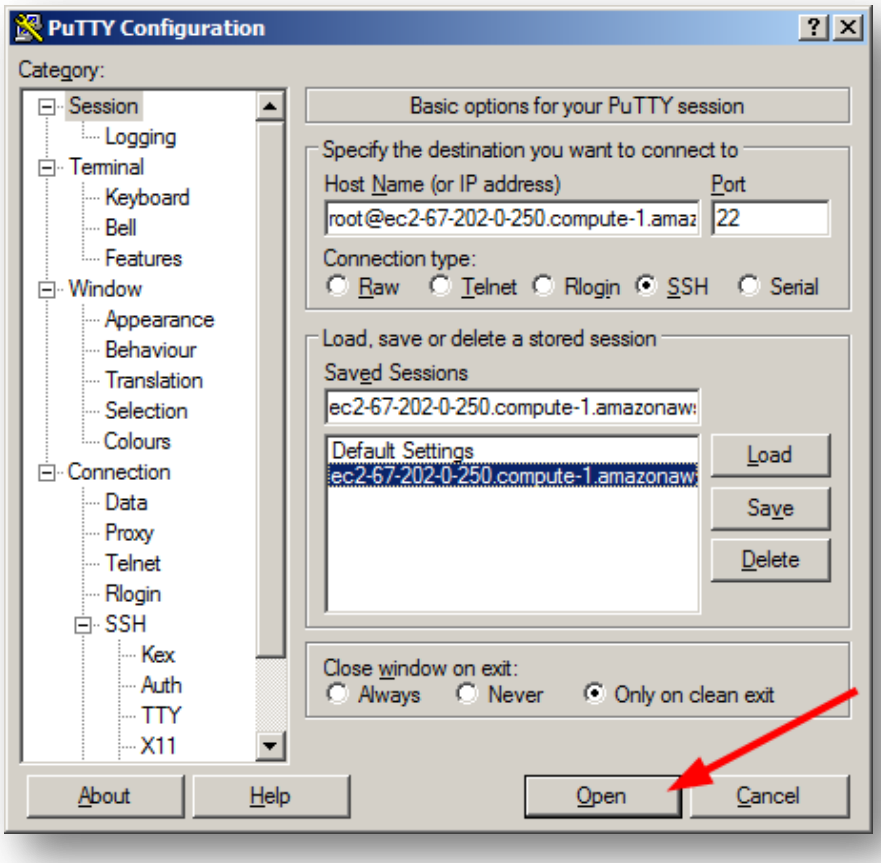
1. Open Putty from your start menu.



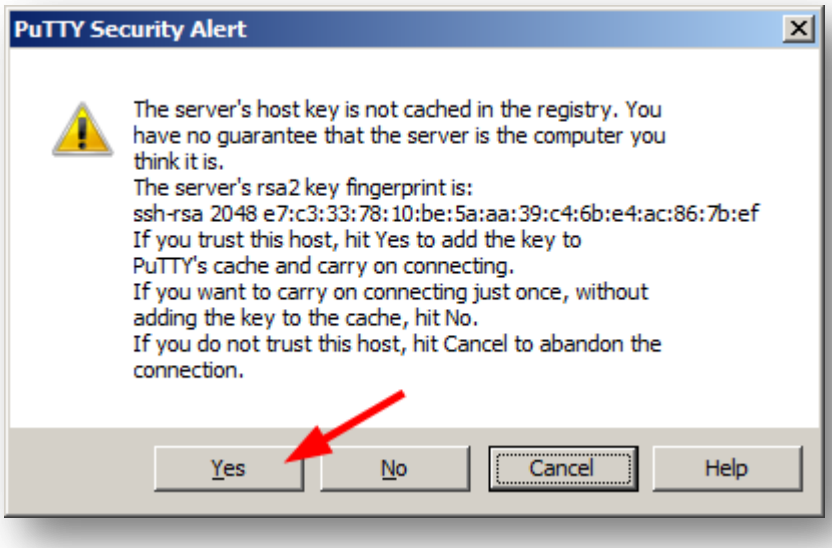
2. Click on the session you want to open.



3. Click ‘Open’.



4. The first time you connect to the server, you will be notified that the server’s host key is unknown. Click ‘Yes’ to acknowledge you are connecting to a new host.



If you provided a private key, and it is valid, you will be taken directly to a shell prompt, like

the one below. If you did not provide a private key, then you will be prompted for your password first.