Clustering Guide

A RabbitMQ *broker* is a logical grouping of one or several Erlang *nodes*, each running the RabbitMQ *application* and sharing users, virtual hosts, queues, exchanges, etc. Sometimes we refer to the collection of nodes as a *cluster*.

All data/state required for the operation of a RabbitMQ broker is replicated across all nodes, for reliability and scaling, with full ACID properties. An exception to this are message queues, which by default reside on the node that created them, though they are visible and reachable from all nodes. To replicate queues across nodes in a cluster, see the documentation on **high availability** (note that you will need a working cluster first).

RabbitMQ clustering does not tolerate network partitions well, so it should not be used over a WAN. The **shovel** or **federation** plugins are better solutions for connecting brokers across a WAN.

The composition of a cluster can be altered dynamically. All RabbitMQ brokers start out as running on a single node. These nodes can be joined into clusters, and subsequently turned back into individual brokers again.

RabbitMQ brokers tolerate the failure of individual nodes. Nodes can be started and stopped at will.

A node can be a *disk node* or a *RAM node*. (Note: *disk* and *disc* are used interchangeably. Configuration syntax or status messages normally use *disc*.) RAM nodes keep their state only in memory (with the exception of queue contents, which can reside on disc if the queue is persistent or too big to fit in memory). Disk nodes keep state in memory and on disk. As RAM nodes don't have to write to disk as much as disk nodes, they can perform better. However, note that since the queue data is always stored on disc, the performance improvements will affect only resources management (e.g. adding/removing queues, exchanges, or vhosts), but not publishing or consuming speed. Because state is replicated across all nodes in the cluster, it is sufficient (but not recommended) to have just one disk node within a cluster, to store the state of the cluster safely.

Clustering transcript

The following is a transcript of setting up and manipulating a RabbitMQ cluster across three machines - `rabbit1`, `rabbit2`, `rabbit3`, with two of the machines replicating data on ram and disk, and the other replicating data in ram only.

We assume that the user is logged into all three machines, that RabbitMQ has been installed on the machines, and that the rabbitmq-server and rabbitmqctl scripts are in the user's PATH.

Erlang cookie

Erlang nodes use a cookie to determine whether they are allowed to communicate with each other - for two nodes to be able to communicate they must have the same cookie. The cookie is just a string of alphanumeric characters. It can be as long or short as you like.

Erlang will automatically create a random cookie file when the RabbitMQ server starts up. The easiest way to proceed is to allow one node to create the file, and then copy it to all the other nodes in the cluster.

On Unix systems, the cookie will be typically located in `/var/lib/rabbitmq/.erlang.cookie` or `$HOME/.erlang.cookie`.

On Windows, the locations are `C:\Users\`*`Current User`*`\.erlang.cookie` (`%HOMEDRIVE% + %HOMEPATH%\.erlang.cookie`) or `C:\Documents and Settings\`*`Current User`*`\.erlang.cookie`, and `C:\Windows\.erlang.cookie` for RabbitMQ Windows service. If Windows service is used, the cookie should be placed in both places.

As an alternative, you can insert the option "`-setcookie` *`cookie`*" in the `erl` call in the `rabbitmq-server` and `rabbitmqctl` scripts.

Starting independent nodes

Clusters are set up by re-configuring existing RabbitMQ nodes into a cluster configuration. Hence the first step is to start RabbitMQ on all nodes in the normal way:

```
rabbit1$ rabbitmq-server -detached
rabbit2$ rabbitmq-server -detached
rabbit3$ rabbitmq-server -detached
```

This creates three *independent* RabbitMQ brokers, one on each node, as confirmed by the *cluster_status* command:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1]}]},{running_nodes,[rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit2]}]},{running_nodes,[rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit3]}]},{running_nodes,[rabbit@rabbit3]}]
...done.
```

The node name of a RabbitMQ broker started from the `rabbitmq-server` shell script is `rabbit@`*`shorthostname`*, where the short node name is lower-case (as in `rabbit@rabbit1`, above). If you use the `rabbitmq-server.bat` batch file on Windows, the short node name is upper-case (as in `rabbit@RABBIT1`). When you type node names, case matters, and these strings must match exactly.

Creating the cluster

In order to link up our three nodes in a cluster, we tell two of the nodes, say `rabbit@rabbit2` and `rabbit@rabbit3`, to join the cluster of the third, say `rabbit@rabbit1`.

We first join `rabbit@rabbit2` as a ram node in a cluster with `rabbit@rabbit1` in a cluster. To do that, on `rabbit@rabbit2` we stop the RabbitMQ application and join the `rabbit@rabbit1` cluster enabling the `--ram` flag, and restart the RabbitMQ application. Note that joining a cluster implicitly resets the node, thus removing all resources and data that were previously present on that node.

```
rabbit2$ rabbitmqctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl join_cluster --ram rabbit@rabbit1
Clustering node rabbit@rabbit2 with [rabbit@rabbit1] ...done.
rabbit2$ rabbitmqctl start_app
Starting node rabbit@rabbit2 ...done.
```

We can see that the two nodes are joined in a cluster by running the *cluster_status* command on either of the nodes:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1]},{ram,[rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1]},{ram,[rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2]}]
...done.
```

Now we join `rabbit@rabbit3` as a disk node to the same cluster. The steps are identical to the ones above, except that we omit the `--ram` flag in order to turn it into a disk rather than ram node. This time we'll cluster to `rabbit2` to demonstrate that the node chosen to cluster to does not matter - it is enough to provide one online node and the node will be clustered to the cluster that the specified node belongs to.

```
rabbit3$ rabbitmqctl stop_app
Stopping node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl join_cluster rabbit@rabbit2
Clustering node rabbit@rabbit3 with rabbit@rabbit2 ...done.
rabbit3$ rabbitmqctl start_app
Starting node rabbit@rabbit3 ...done.
```

We can see that the three nodes are joined in a cluster by running the *cluster_status* command on any of the nodes:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit3]},{ram,[rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit3,rabbit@rabbit2,rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit3]},{ram,[rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit3,rabbit@rabbit1,rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit3,rabbit@rabbit1]},{ram,[rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1,rabbit@rabbit3]}]
...done.
```

By following the above steps we can add new nodes to the cluster at any time, while the cluster is running.

Changing node types

We can change the type of a node from ram to disk and vice versa. Say we wanted to reverse the types of `rabbit@rabbit2` and `rabbit@rabbit3`, turning the former from a ram node into a disk node and the latter from a disk node into a ram node. To do that we can use the `change_cluster_node_type` command. The node must be stopped first.

```
rabbit2$ rabbitmqctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl change_cluster_node_type disc
Turning rabbit@rabbit2 into a disc node ...
...done.
Starting node rabbit@rabbit2 ...done.
rabbit3$ rabbitmqctl stop_app
Stopping node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl change_cluster_node_type ram
Turning rabbit@rabbit3 into a ram node ...
rabbit3$ rabbitmqctl start_app
Starting node rabbit@rabbit3 ...done.
```

Restarting cluster nodes

Nodes that have been joined to a cluster can be stopped at any time. It is also ok for them to crash. In both cases the rest of the cluster continues operating unaffected, and the nodes automatically "catch up" with the other cluster nodes when they start up again.

We shut down the nodes `rabbit@rabbit1` and `rabbit@rabbit3` and check on the cluster status at each step:

```
rabbit1$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit1 ...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit3,rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit2,rabbit@rabbit1]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit3]}]
...done.
rabbit3$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit3 ...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2]}]
...done.
```

Now we start the nodes again, checking on the cluster status as we go along:

```
rabbit1$ rabbitmq-server -detached
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmq-server -detached
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1,rabbit@rabbit3]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit2,rabbit@rabbit1]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1,rabbit@rabbit3]}]
...done.
```

There are some important caveats:

- At least one disk node should be running at all times to prevent data loss. RabbitMQ will prevent the creation of a RAM-only cluster in many situations, but it still won't stop you from stopping and forcefully resetting all the disc nodes, which will lead to a RAM-only cluster. Doing this is not advisable and makes losing data very easy.
- When the entire cluster is brought down, the last node to go down must be the first node to be brought online. If this doesn't happen, the nodes will wait 30 seconds for the last disc node to come back online, and fail afterwards. If the last node to go offline cannot be brought back up, it can be removed from the cluster using the `forget_cluster_node` command - consult the `rabbitmqctl` manpage for more information.

Breaking up a cluster

Nodes need to be removed explicitly from a cluster when they are no longer meant to be part of it. We first remove `rabbit@rabbit3` from the cluster, returning it to independent operation. To do that, on `rabbit@rabbit3` we stop the RabbitMQ application, reset the node, and restart the RabbitMQ application.

```
rabbit3$ rabbitmqctl stop_app
Stopping node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl reset
Resetting node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl start_app
Starting node rabbit@rabbit3 ...done.
```

Note that it would have been equally valid to list `rabbit@rabbit3` as a node.

Running the *cluster_status* command on the nodes confirms that `rabbit@rabbit3` now is no longer part of the cluster and operates independently:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit2,rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
```

```
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit3]}]},{running_nodes,[rabbit@rabbit3]}]
...done.
```

We can also remove nodes remotely. This is useful, for example, when having to deal with an unresponsive node. We can for example remove `rabbit@rabbi1` from `rabbit@rabbit2`.

```
rabbit1$ rabbitmqctl stop_app
Stopping node rabbit@rabbit1 ...done.
rabbit2$ rabbitmqctl forget_cluster_node rabbit@rabbit1
Removing node rabbit@rabbit1 from cluster ...
...done.
```

Note that `rabbit1` still thinks its clustered with `rabbit2`, and trying to start it will result in an error. We will need to reset it to be able to start it again.

```
rabbit1$ rabbitmqctl start_app
Starting node rabbit@rabbit1 ...
Error: inconsistent_cluster: Node rabbit@rabbit1 thinks it's clustered with node rabbit@rabbit2, but rabbit@rabbit2 disagrees
rabbit1$ rabbitmqctl reset
Resetting node rabbit@rabbit1 ...done.
rabbit1$ rabbitmqctl start_app
Starting node rabbit@mcnulty ...
...done.
```

The *cluster_status* command now shows all three nodes operating as independent RabbitMQ brokers:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1]}]},{running_nodes,[rabbit@rabbit1]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit2]}]},{running_nodes,[rabbit@rabbit2]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit3]}]},{running_nodes,[rabbit@rabbit3]}]
...done.
```

Note that `rabbit@rabbit2` retains the residual state of the cluster, whereas `rabbit@rabbit1` and `rabbit@rabbit3` are freshly initialised RabbitMQ brokers. If we want to re-initialise `rabbit@rabbit2` we follow the same steps as for the other nodes:

```
rabbit2$ rabbitmqctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl reset
Resetting node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl start_app
Starting node rabbit@rabbit2 ...done.
```

Auto-configuration of a cluster

Instead of configuring clusters "on the fly" using the `cluster` command, clusters can also be set up via the **RabbitMQ configuration file**. The file should set the `cluster_nodes` field in the rabbit application to a tuple containining a list of rabbit nodes, and an atom - either `disc` or `ram` - indicating whether the node should join them as a disc node or not.

If `cluster_nodes` is specified, RabbitMQ will try to cluster to each node provided, and stop after it can cluster with one of them. RabbitMQ will try cluster to any node which is online that has the same version of Erlang and RabbitMQ. If no suitable nodes are found, the node is left unclustered.

Note that the cluster configuration is applied only to fresh nodes. A fresh nodes is a node which has just been reset or is being start for the first time. Thus, the automatic clustering won't take place after restarts of nodes. This means that any change to the clustering via `rabbitmqctl` will take precedence over the automatic clustering configuration.

A common use of cluster configuration via the RabbitMQ config file is to automatically configure nodes to join a common cluster. For this purpose the same cluster nodes can be specified on all cluster, plus the boolean to determine disc nodes.

Say we want to join our three separate nodes of our running example back into a single cluster, with `rabbit@rabbit1` and `rabbit@rabbit2` being the disk nodes of the cluster. First we reset and stop all nodes, to make sure that we're working with fresh nodes:

```
rabbit1$ rabbitmqctl stop_app
Stopping node rabbit@rabbit1 ...done.
rabbit1$ rabbitmqctl reset
Resetting node rabbit@rabbit1 ...done.
rabbit1$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit1 ...done.
rabbit2$ rabbitmqctl stop_app
Stopping node rabbit@rabbit2 ...done.
rabbit2$ rabbitmqctl reset
Resetting node rabbit@rabbit2 ...done.
```

```
rabbit2$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit2 ...done.
rabbit3$ rabbitmqctl stop_app
Stopping node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl reset
Resetting node rabbit@rabbit3 ...done.
rabbit3$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit3 ...done.
```

Now we set the relevant field in the config file:

```
[
  ...
  {rabbit, [
        ...
        {cluster_nodes, {['rabbit@rabbit1', 'rabbit@rabbit2', 'rabbit@rabbit3'], disc}},
        ...
  ]},
  ...
].
```

For instance, if this were the only field we needed to set, we would simply create the RabbitMQ config file with the contents:

```
[{rabbit,
  [{cluster_nodes, {['rabbit@rabbit1', 'rabbit@rabbit2', 'rabbit@rabbit3'], disc}}]}].
```

Since we want `rabbit@rabbit3` to be a ram node, we need to specify that in its configuration file:

```
[{rabbit,
  [{cluster_nodes, {['rabbit@rabbit1', 'rabbit@rabbit2', 'rabbit@rabbit3'], ram}}]}].
```

(Note for Erlang programmers and the curious: this is a standard Erlang configuration file. For more details, see the **configuration guide** and the **Erlang Config Man Page**.)

Once we have the configuration files in place, we simply start the nodes:

```
rabbit1$ rabbitmq-server -detached
rabbit2$ rabbitmq-server -detached
rabbit3$ rabbitmq-server -detached
```

We can see that the three nodes are joined in a cluster by running the *cluster_status* command on any of the nodes:

```
rabbit1$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit1 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]
...done.
rabbit2$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit2 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]
...done.
rabbit3$ rabbitmqctl cluster_status
Cluster status of node rabbit@rabbit3 ...
[{nodes,[{disc,[rabbit@rabbit1,rabbit@rabbit2]},{ram,[rabbit@rabbit3]}]},
 {running_nodes,[rabbit@rabbit1,rabbit@rabbit2,rabbit@rabbit3]}]
...done.
```

Note that, in order to remove a node from an auto-configured cluster, it must first be removed from the **RabbitMQ configuration file** files of the other nodes in the cluster. Only then, can it be reset safely.

Upgrading clusters

When upgrading from one major or minor version of RabbitMQ to another (i.e. from 3.0.x to 3.1.x, or from 2.x.x to 3.x.x), or when upgrading Erlang, the whole cluster must be taken down for the upgrade (since clusters cannot run mixed versions like this). This will not be the case when upgrading from one patch version to another (i.e. from 3.0.x to 3.0.y); these versions can be mixed in a cluster (with the exception that 3.0.0 cannot be mixed with later versions from the 3.0.x series).

RabbitMQ will automatically update its persistent data structures if necessary when upgrading between major / minor versions. In a cluster, this task is performed by the first disc node to be started (the "upgrader" node). Therefore when upgrading a RabbitMQ cluster, you should not attempt to start any RAM nodes first; any RAM nodes started will emit an error message and fail to start up.

While not strictly necessary, it is a good idea to decide ahead of time which disc node will be the upgrader, stop that node last, and start it first. Otherwise changes to the cluster configuration that were made between the upgrader node stopping and the last node stopping will be lost.

Automatic upgrades are only possible from RabbitMQ versions 2.1.1 and later. If you have an earlier cluster, you will need to rebuild it to upgrade.

A cluster on a single machine

Under some circumstances it can be useful to run a cluster of RabbitMQ nodes on a single machine. This would typically be useful for experimenting with clustering on a desktop or laptop without the overhead of starting several virtual machines for the cluster. The two main requirements for running more than one node on a single machine are that each node should have a unique name and bind to a unique port / IP address combination for each protocol in use.

You can start multiple nodes on the same host manually by repeated invocation of `rabbitmq-server` (

`rabbitmq-server.bat` on Windows). You must ensure that for each invocation you set the environment variables `RABBITMQ_NODENAME` and `RABBITMQ_NODE_PORT` to suitable values.

For example:

```
$ RABBITMQ_NODE_PORT=5672 RABBITMQ_NODENAME=rabbit rabbitmq-server -detached
$ RABBITMQ_NODE_PORT=5673 RABBITMQ_NODENAME=hare rabbitmq-server -detached
$ rabbitmqctl -n hare stop_app
$ rabbitmqctl -n hare join_cluster rabbit@`hostname -s`
$ rabbitmqctl -n hare start_app
```

will set up a two node cluster, both nodes as disc nodes. Note that if you have RabbitMQ opening any ports other than AMQP, you'll need to configure those not to clash as well - for example:

```
$ RABBITMQ_NODE_PORT=5672 RABBITMQ_SERVER_START_ARGS="-rabbitmq_management listener [{port,15672}]" RABBITMQ_NODENAME=rabbit ra
$ RABBITMQ_NODE_PORT=5673 RABBITMQ_SERVER_START_ARGS="-rabbitmq_management listener [{port,15673}]" RABBITMQ_NODENAME=hare rabb
```

will start two nodes (which can then be clustered) when the management plugin is installed.

Firewalled nodes

The case for firewalled clustered nodes exists when nodes are in a data center or on a reliable network, but separated by firewalls. Again, clustering is not recommended over a WAN or when network links between nodes are unreliable.

In the most common configuration you will need to open ports 4369 and 25672 for clustering to work.

Erlang makes use of a Port Mapper Daemon (epmd) for resolution of node names in a cluster. The default epmd port is 4369, but this can be changed using the ERL_EPMD_PORT environment variable. All nodes must use the same port. For further details see the **Erlang epmd manpage**.

Once a distributed Erlang node address has been resolved via epmd, other nodes will attempt to communicate directly with that address using the Erlang distributed node protocol. The default port for this traffic in RabbitMQ is 20000 higher than RABBITMQ_NODE_PORT (i.e. 25672 by default). This can be explicitly configured using the RABBITMQ_DIST_PORT variable - see **the configuration guide**.

Connecting to Clusters from Clients

A client can connect as normal to any node within a cluster. If that node should fail, and the rest of the cluster survives, then the client should notice the closed connection, and should be able to reconnect to some surviving member of the cluster. Generally, it's not advisable to bake in node hostnames or IP addresses into client applications: this introduces inflexibility and will require client applications to be edited, recompiled and redeployed should the configuration of the cluster change or the number of nodes in the cluster change. Instead, we recommend a more abstracted approach: this could be a dynamic DNS service which has a very short TTL configuration, or a plain TCP load balancer, or some sort of mobile IP achieved with pacemaker or similar technologies. In general, this aspect of managing the connection to nodes within a cluster is beyond the scope of RabbitMQ itself, and we recommend the use of other technologies designed specifically to solve these problems.

Sitemap | Contact