

# Qpid Dispatch Router Book

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview	1
1.2	Benefits	2
1.3	Features	2
<b>2</b>	<b>Using Qpid Dispatch</b>	<b>3</b>
2.1	Configuration	3
2.2	Tools	3
2.2.1	qdstat	3
2.2.2	qmanage	3
2.3	Basic Usage and Examples	4
2.3.1	Standalone and Interior Modes	4
2.3.2	Mobile Subscribers	5
2.3.3	Dynamic Reply-To	5
2.4	Link Routing	7
2.4.1	Configuration	8
2.5	Indirect Waypoints and Auto-Links	9
2.5.1	Queue Waypoint Example	9
2.5.2	Sharded Queue Example	11
2.5.3	Dynamically Adding Shards	12
<b>3</b>	<b>Technical Details and Specifications</b>	<b>13</b>
3.1	Client Compatibility	13
3.2	Addressing	13
3.2.1	Routing patterns	14
3.2.2	Routing mechanisms	14
3.2.2.1	Message routing	14
3.3	AMQP Mapping	14
3.3.1	Message Annotations	15
3.3.2	Source/Target Capabilities	15
3.3.3	Dynamic-Node-Properties	15

---

3.3.4	Addresses and Address Formats	15
3.3.4.1	Address Patterns	15
3.3.4.2	Supported Addresses	16
3.3.5	Implementation of the AMQP Management Specification	16
3.4	Configuration Entities	17
3.4.1	container	17
3.4.2	router	17
3.4.3	listener	18
3.4.4	connector	20
3.4.5	log	22
3.4.6	fixedAddress	22
3.4.7	waypoint	23
3.4.8	linkRoutePattern	23
3.4.9	address	23
3.4.10	linkRoute	24
3.4.11	autoLink	24
3.4.12	console	25
3.4.13	policy	25
3.5	Operational Entities	26
3.5.1	org.amqp.management	26
3.5.1.1	Operation GET-TYPES	26
3.5.1.2	Operation GET-ATTRIBUTES	27
3.5.1.3	Operation GET-OPERATIONS	27
3.5.1.4	Operation GET-ANNOTATIONS	27
3.5.1.5	Operation QUERY	27
3.5.1.6	Operation GET-MGMT-NODES	28
3.5.2	management	28
3.5.2.1	Operation GET-SCHEMA-JSON	28
3.5.2.2	Operation GET-LOG	29
3.5.2.3	Operation GET-SCHEMA	29
3.5.3	router.link	29
3.5.4	router.address	30
3.5.5	router.node	30
3.5.6	connection	31
3.5.7	allocator	32
3.5.8	Operations for all entity types	32
3.5.8.1	Operation READ	32
3.5.8.2	Operation CREATE	32
3.5.8.3	Operation UPDATE	33

---

3.5.8.4	Operation DELETE	33
3.5.9	Operations for <i>org.amqp.management</i> entity type	33
3.5.9.1	Operation GET-TYPES	33
3.5.9.2	Operation GET-ATTRIBUTES	34
3.5.9.3	Operation GET-OPERATIONS	34
3.5.9.4	Operation GET-ANNOTATIONS	34
3.5.9.5	Operation QUERY	34
3.5.9.6	Operation GET-MGMT-NODES	35
3.5.10	Operations for <i>management</i> entity type	35
3.5.10.1	Operation GET-SCHEMA-JSON	35
3.5.10.2	Operation GET-LOG	35
3.5.10.3	Operation GET-SCHEMA	36
<b>4</b>	<b>Console</b>	<b>37</b>
4.1	Console overview	37
4.2	Console installation	37
4.2.1	Prerequisites	37
4.2.2	The console files	38
4.3	Console operation	38
4.3.1	Logging in to a router network	38
4.3.2	Overview page	38
4.3.3	Topology page	39
4.3.4	Router entity details page	41
4.3.5	Charts page	42
4.3.6	Schema page	43

---

# Chapter 1

## Introduction

### 1.1 Overview

The Dispatch router is an AMQP message router that provides advanced interconnect capabilities. It allows flexible routing of messages between any AMQP-enabled endpoints, whether they be clients, servers, brokers or any other entity that can send or receive standard AMQP messages.

A messaging client can make a single AMQP connection into a messaging bus built of Dispatch routers and, over that connection, exchange messages with one or more message brokers, and at the same time exchange messages directly with other endpoints without involving a broker at all.

The router is an intermediary for messages but it is *not* a broker. It does not *take responsibility* for messages. It will, however, propagate settlement and disposition across a network such that delivery guarantees are met. In other words: the router network will deliver the message, possibly via several intermediate routers, *and* it will route the acknowledgement of that message by the ultimate receiver back across the same path. This means that *responsibility* for the message is transferred from the original sender to the ultimate receiver *as if they were directly connected*. However this is done via a flexible network that allows highly configurable routing of the message transparent to both sender and receiver.

There are some patterns where this enables "brokerless messaging" approaches that are preferable to brokered approaches. In other cases a broker is essential (in particular where you need the separation of responsibility and/or the buffering provided by store-and-forward) but a dispatch network can still be useful to tie brokers and clients together into patterns that are difficult with a single broker.

For a "brokerless" example, consider the common brokered implementation of the request-response pattern, a client puts a request on a queue and then waits for a reply on another queue. In this case the broker can be a hindrance - the client may want to know immediately if there is nobody to serve the request, but typically it can only wait for a timeout to discover this. With a dispatch network, the client can be informed immediately if its message cannot be delivered because nobody is listening. When the client receives acknowledgement of the request it knows not just that it is sitting on a queue, but that it has actually been received by the server.

For an example of using dispatch to enhance the use of brokers, consider using an array of brokers to implement a scalable distributed work queue. A dispatch network can make this appear as a single queue, with senders publishing to a single address and receivers subscribing to a single address. The dispatch network can distribute work to any broker in the array and collect work from any broker for any receiver. Brokers can be shut down or added without affecting clients. This elegantly solves the common difficulty of "stuck messages" when implementing this pattern with brokers alone. If a receiver is connected to a broker that has no messages, but there are messages on another broker, you have to somehow transfer them or leave them "stuck". With a dispatch network, *all* the receivers are connected to *all* the brokers. If there is a message anywhere it can be delivered to any receiver.

The router is meant to be deployed in topologies of multiple routers, preferably with redundant paths. It uses link-state routing protocols and algorithms (similar to OSPF or IS-IS from the networking world) to calculate the best path from every point to every other point and to recover quickly from failures. It does not need to use clustering for high availability; rather, it relies on redundant paths to provide continued connectivity in the face of system or network failure. Because it never takes responsibility for messages it is effectively stateless. Messages not delivered to their final destination will not be acknowledged to the sender and therefore the sender can re-send such messages if it is disconnected from the network.

---

## 1.2 Benefits

Simplifies connectivity

- An endpoint can do all of its messaging through a single transport connection
- Avoid opening holes in firewalls for incoming connections

Provides messaging connectivity where there is no TCP/IP connectivity

- A server or broker can be in a private IP network (behind a NAT firewall) and be accessible by messaging endpoints in other networks ([learn more](#)).

Simplifies reliability

- Reliability and availability are provided using redundant topology, not server clustering
- Reliable end-to-end messaging without persistent stores
- Use a message broker only when you need store-and-forward semantics

## 1.3 Features

- Can be deployed stand-alone or in a network of routers
    - Supports arbitrary network topology - no restrictions on redundancy
      - \* Automatic route computation - adjusts quickly to changes in topology
  - Provides remote access to brokers or other AMQP servers
  - Security
-

## Chapter 2

# Using Qpid Dispatch

### 2.1 Configuration

The default configuration file is installed in *install-prefix/etc/qpid-dispatch/qdrouterd.conf*. This configuration file will cause the router to run in standalone mode, listening on the standard AMQP port (5672). Dispatch Router looks for the configuration file in the installed location by default. If you wish to use a different path, the "-c" command line option will instruct Dispatch Router as to which configuration to load.

To run the router, invoke the executable: `qdrouterd [-c my-config-file]`

For more details of the configuration file see the *qdrouterd.conf(5)* man page.

### 2.2 Tools

#### 2.2.1 qdstat

*qdstat* is a command line tool that lets you view the status of a Dispatch Router. The following options are useful for seeing what the router is doing:

<i>Option</i>	<i>Description</i>
-l	Print a list of AMQP links attached to the router. Links are unidirectional. Outgoing links are usually associated with a subscription address. The tool distinguishes between <i>endpoint</i> links and <i>router</i> links. Endpoint links are attached to clients using the router. Router links are attached to other routers in a network of routers.
-a	Print a list of addresses known to the router.
-n	Print a list of known routers in the network.
-c	Print a list of connections to the router.
--autolinks	Print a list of configured auto-links.
--linkroutes	Print a list of configured link-routes.

For complete details see the *qdstat(8)* man page and the output of `qdstat --help`.

#### 2.2.2 qdmanage

*qdmanage* is a general-purpose AMQP management client that allows you to not only view but modify the configuration of a running dispatch router.

For example you can query all the connection entities in the router:

---



```
$ qdmanage query --type connection
```

To enable logging debug and higher level messages by default:

```
$ qdmanage update log/DEFAULT enable=debug+
```

In fact, everything that can be configured in the configuration file can also be created in a running router via management. For example to create a new listener in a running router:

```
$ qdmanage create type=listener port=5555
```

Now you can connect to port 5555, for example:

```
$ qdmanage query -b localhost:5555 --type listener
```

For complete details see the *qdmanage(8)* man page and the output of `qdmanage --help`. Also for details of what can be configured see the *qdrouterd.conf(5)* man page.

## 2.3 Basic Usage and Examples

### 2.3.1 Standalone and Interior Modes

The router can operate stand-alone or as a node in a network of routers. The mode is configured in the *router* section of the configuration file. In stand-alone mode, the router does not attempt to collaborate with any other routers and only routes messages among directly connected endpoints.

If your router is running in stand-alone mode, *qdstat -a* will look like the following:

```
$ qdstat -a
Router Addresses
class  addr                phs  distrib  in-proc  local  remote  cntnr  in  out  thru ←
      to-proc  from-proc
=====
local  $_management_internal  closest  1      0      0      0      0  0  0  0 ←
      0          0
local  $displayname          closest  1      0      0      0      0  0  0  0 ←
      0          0
mobile $management        0      closest  1      0      0      0      1  0  0  0 ←
      1          0
local  $management          closest  1      0      0      0      0  0  0  0 ←
      0          0
local  temp.1GThUllfR7N+BDP  closest  0      1      0      0      0  0  0  0 ←
      0          0
```

Note that there are a number of known addresses. *\$management* is the address of the router's embedded management agent. *temp.1GThUllfR7N+BDP* is the temporary reply-to address of the *qdstat* client making requests to the agent.

If you change the mode to interior and restart the processs, the same command will yield additional addresses which are used for inter-router communication:

```
$ qdstat -a
Router Addresses
class  addr                phs  distrib  in-proc  local  remote  cntnr  in  out  ←
      thru  to-proc  from-proc
=====
local  $_management_internal  closest  1      0      0      0      0  0  0  0 ←
      0          0
```

local	\$displayname		closest	1	0	0	0	0	0	0	↵
	0	0									
mobile	\$management	0	closest	1	0	0	0	1	0	0	↵
	1	0									
local	\$management		closest	1	0	0	0	0	0	0	↵
	0	0									
local	qdhello		flood	1	0	0	0	0	0	0	↵
	0	10									
local	qdrouter		flood	1	0	0	0	0	0	0	↵
	0	0									
topo	qdrouter		flood	1	0	0	0	0	0	0	↵
	0	1									
local	qdrouter.ma		multicast	1	0	0	0	0	0	0	↵
	0	0									
topo	qdrouter.ma		multicast	1	0	0	0	0	0	0	↵
	0	0									
local	temp.wfx54+zf+YWQF3T		closest	0	1	0	0	0	0	0	↵
	0	0									

### 2.3.2 Mobile Subscribers

The term "mobile subscriber" simply refers to the fact that a client may connect to the router and subscribe to an address to receive messages sent to that address. No matter where in the network the subscriber attaches, the messages will be routed to the appropriate destination.

To illustrate a subscription on a stand-alone router, you can use the examples that are provided with Qpid Proton. Using the *simple\_recv.py* example receiver:

```
$ python ./simple_recv.py -a 127.0.0.1/my-address
```

This command creates a receiving link subscribed to the specified address. To verify the subscription:

```
$ qdstat -a
```

Router Addresses											
class	addr		phs	distrib	in-proc	local	remote	cntnr	in	out	thru ↵
	to-proc	from-proc									
=====											
local	\$_management_internal		closest	1	0	0	0	0	0	0	↵
	0	0									
local	\$displayname		closest	1	0	0	0	0	0	0	↵
	0	0									
mobile	\$management	0	closest	1	0	0	0	2	0	0	↵
	2	0									
local	\$management		closest	1	0	0	0	0	0	0	↵
	0	0									
mobile	my-address	0	closest	0	1	0	0	0	0	0	↵
	0	0									
local	temp.75_d2X23x_KOT51		closest	0	1	0	0	0	0	0	↵
	0	0									

You can then, in a separate command window, run a sender to produce messages to that address:

```
$ python ./simple_send.py -a 127.0.0.1/my-address
```

### 2.3.3 Dynamic Reply-To

Dynamic reply-to can be used to obtain a reply-to address that routes back to a client's receiving link regardless of how many hops it has to take to get there. To illustrate this feature, see below a simple program (written in C++ against the `qpid::messaging` API) that queries the management agent of the attached router for a list of other known routers' management addresses.

```

#include <qpid/messaging/Address.h>
#include <qpid/messaging/Connection.h>
#include <qpid/messaging/Message.h>
#include <qpid/messaging/Receiver.h>
#include <qpid/messaging/Sender.h>
#include <qpid/messaging/Session.h>

using namespace qpid::messaging;
using namespace qpid::types;

using std::stringstream;
using std::string;

int main() {
    const char* url = "amqp:tcp:127.0.0.1:5672";
    std::string connectionOptions = "{protocol:amqp1.0}";

    Connection connection(url, connectionOptions);
    connection.open();
    Session session = connection.createSession();
    Sender sender = session.createSender("mgmt");

    // create reply receiver and get the reply-to address
    Receiver receiver = session.createReceiver("#");
    Address responseAddress = receiver.getAddress();

    Message request;
    request.setReplyTo(responseAddress);
    request.setProperty("x-amqp-to", "amqp://_local/$management");
    request.setProperty("operation", "DISCOVER-MGMT-NODES");
    request.setProperty("type", "org.amqp.management");
    request.setProperty("name", "self");

    sender.send(request);
    Message response = receiver.fetch();
    Variant content(response.getContentObject());
    std::cout << "Response: " << content << std::endl << std::endl;

    connection.close();
}

```

The equivalent program written in Python against the Proton Messenger API:

```

from proton import Messenger, Message

def main():
    host = "0.0.0.0:5672"

    messenger = Messenger()
    messenger.start()
    messenger.route("amqp:/*", "amqp://%s/%s" % host)
    reply_subscription = messenger.subscribe("amqp:/#")
    reply_address = reply_subscription.address

    request = Message()
    response = Message()

    request.address = "amqp://_local/$management"
    request.reply_to = reply_address
    request.properties = {'operation' : u'DISCOVER-MGMT-NODES',
                          'type'      : u'org.amqp.management',

```

```

        u'name'      : u'self' }

    messenger.put(request)
    messenger.send()
    messenger.recv()
    messenger.get(response)

    print "Response: %r" % response.body

    messenger.stop()

main()

```

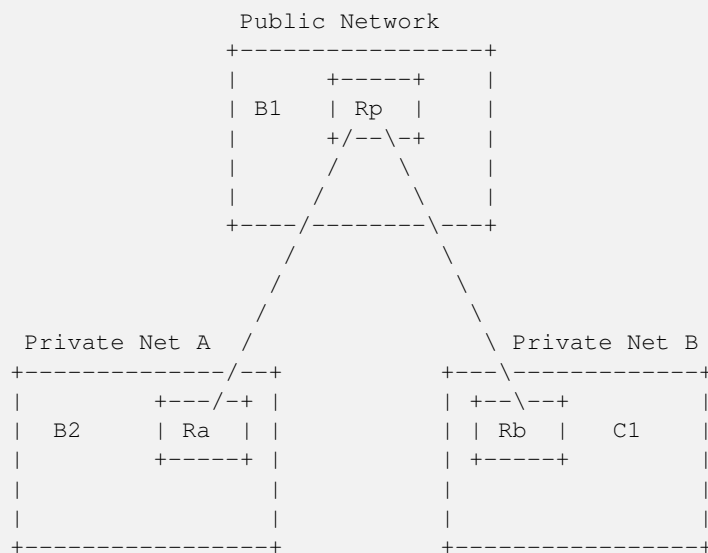
## 2.4 Link Routing

This feature was introduced in Qpid Dispatch 0.4. This feature was significantly updated in Qpid Dispatch 0.6.

Link-routing is an alternative strategy for routing messages across a network of routers. With the existing message-routing strategy, each router makes a routing decision on a per-message basis when the message is delivered. Link-routing is different because it makes routing decisions when link-attach frames arrive. A link is effectively chained across the network of routers from the establishing node to the destination node. Once the link is established, the transfer of message deliveries, flow frames, and dispositions is performed across the routed link.

The main benefit to link-routing is that endpoints can use the full link protocol to interact with other endpoints in far-flung parts of the network. For example, a client can establish a receiver across the network to a queue on a remote broker and use link credit to control the flow of messages from the broker. Similarly, a receiver can establish a link to a topic on a remote broker using a server-side filter.

Why would one want to do this? One reason is to provide client isolation. A network like the following can be deployed:



The clients in Private Net B can be constrained (by firewall policy) to only connect to the Router in their own network. Using link-routing, these clients can access queues, topics, and other AMQP services that are in the Public Network or even in Private Net A.

For example, The router Ra can be configured to expose queues in broker B2 to the network. Client C1 can then establish a connection to Rb, the local router, open a subscribing link to "b2.event-queue", and receive messages stored on that queue in broker B2.

C1 is unable to create a TCP/IP connection to B1 because of its isolation (and because B2 is itself in a private network). However, with link routing, C1 can interact with B2 using the AMQP link protocol.

Note that in this case, neither C1 nor B2 have been modified in any way and neither need be aware of the fact that there is a message-router network between them.

### 2.4.1 Configuration

Starting with the configured topology shown above, how is link-routing configured to support the example described above?

First, router Ra needs to be told how to make a connection to the broker B2:

```
connector {
  name: broker
  role: route-container
  host: <B2-url>
  port: <B2-port>
  sasl-mechanisms: ANONYMOUS
}
```

This *route-container* connector tells the router how to connect to an external AMQP container when it is needed. The name "broker" will be used later to refer to this connection.

Now, the router must be configured to route certain addresses to B2:

```
linkRoute {
  prefix: b2
  dir: in
  connection: broker
}

linkRoute {
  prefix: b2
  dir: out
  connection: broker
}
```

The linkRoute tells router Ra that any sender or receiver that is attached with a target or source (respectively) whos address begins with "b2", should be routed to the broker B2 (via the route-container connector).

Note that receiving and sending links are configured and routed separately. This allows configuration of link routes for listeners only or senders only. A direction of "in" matches client senders (i.e. links that carry messages inbound to the router network). Direction "out" matches client receivers.

Examples of addresses that "begin with b2" include:

- b2
- b2.queues
- b2.queues.app1

When the route-container connector is configured, router Ra establishes a connection to the broker. Once the connection is open, Ra tells the other routers (Rp and Rb) that it is a valid destination for link-routes to the "b2" prefix. This means that sender or receiver links attached to Rb or Rp will be routed via the shortest path to Ra where they are then routed outbound to the broker B2.

On Rp and Rb, it is advisable to add the identical configuration. It is permissible for a linkRoute configuration to reference a connection that does not exist.

This configuration tells the routers that link-routing is intended to be available for targets and sources starting with "b2". This is important because it is possible that B2 might be unavailable or shut off. If B2 is unreachable, Ra will not advertize itself as a destination for "b2" and the other routers might never know that "b2" was intended for link-routing.

The above configuration allows Rb and Rp to reject attaches that should be routed to B2 with an error message that indicates that there is no route available to the destination.

## 2.5 Indirect Waypoints and Auto-Links

This feature was introduced in Qpid Dispatch 0.6. It is a significant improvement on an earlier somewhat experimental feature called Waypoints.

Auto-link is a feature of Qpid Dispatch Router that enables a router to actively attach a link to a node on an external AMQP container. The obvious application for this feature is to route messages through a queue on a broker, but other applications are possible as well.

An auto-link manages the lifecycle of one AMQP link. If messages are to be routed to and from a queue on a broker, then two auto-links are needed: one for sending messages to the queue and another for receiving messages from the queue. The container to which an auto-link attempts to attach may be identified in one of two ways:

- The name of the connector/listener that resulted in the connection of the container, or
- The AMQP container-id of the remote container.

### 2.5.1 Queue Waypoint Example

Here is an example configuration for routing messages deliveries through a pair of queues on a broker:

```
connector {
  name: broker
  role: route-container
  host: <hostname>
  port: <port>
  sasl-mechanisms: ANONYMOUS
}

address {
  prefix: queue
  waypoint: yes
}

autoLink {
  addr: queue.first
  dir: in
  connection: broker
}

autoLink {
  addr: queue.first
  dir: out
  connection: broker
}

autoLink {
  addr: queue.second
  dir: in
  connection: broker
}

autoLink {
  addr: queue.second
  dir: out
  connection: broker
}
```

The *address* entity identifies a namespace (queue.) that will be used for routing messages through queues via autolinks. The *four* *autoLink*\* entities identify the head and tail of two queues on the broker that will be connected via auto-links.

If there is no broker connected, the auto-links shall remain *inactive*. This can be observed by using the *qdstat* tool:

```
$ qdstat --autolinks
AutoLinks
  addr          dir  phase  link  status  lastErr
=====
queue.first    in   1      queue.first  inactive
queue.first    out  0      queue.first  inactive
queue.second   in   1      queue.second  inactive
queue.second   out  0      queue.second  inactive
```

If a broker comes online with a queue called *queue.first*, the auto-links will attempt to activate:

```
$ qdstat --autolinks
AutoLinks
  addr          dir  phase  link  status  lastErr
=====
queue.first    in   1      6      active
queue.first    out  0      7      active
queue.second   in   1      failed  Node not found: queue.second
queue.second   out  0      failed  Node not found: queue.second
```

Note that two of the auto-links are in *failed* state because the queue does not exist on the broker.

If we now use the Qpid Proton example application *simple\_send* to send three messages to *queue.first* via the router:

```
$ python simple_send.py -a 127.0.0.1/queue.first -m3
all messages confirmed
```

and then look at the address statistics on the router:

```
$ qdstat -a
Router Addresses
  class  addr          phs  distrib  in-proc  local  remote  cntnr  in  out  thru  to-  ←
      proc  from-proc
=====
mobile  queue.first  1    balanced  0         0      0        0      0  0   0    0    0  ←
      0
mobile  queue.first  0    balanced  0         1      0        0      3  3   0    0    0  ←
      0
```

we see that *queue.first* appears twice in the list of addresses. The *phs*, or phase column shows that there are two phases for the address. Phase 0 is for routing message deliveries from producers to the tail of the queue (the *out* auto-link associated with the queue). Phase 1 is for routing deliveries from the head of the queue to subscribed consumers.

Note that three deliveries have been counted in the "in" and "out" columns for phase 0. The "in" column represents the three messages that arrived from *simple\_send* and the "out" column represents the three deliveries to the queue on the broker.

If we now use *simple\_recv* to receive three messages from this address:

```
$ python simple_recv_noignore.py -a 127.0.0.1:5672/queue.first -m3
{u'sequence': int32(1)}
{u'sequence': int32(2)}
{u'sequence': int32(3)}
```

We receive the three queued messages. Looking at the addresses again, we see that phase 1 was used to deliver those messages from the queue to the consumer.

```
$ qdstat -a
Router Addresses
  class  addr          phs  distrib  in-proc  local  remote  cntnr  in  out  thru  to-  ←
      proc  from-proc
=====
```

```
=====
mobile  queue.first    1    balanced  0      0      0      0      3    3    0    0  ↔
         0
mobile  queue.first    0    balanced  0      1      0      0      3    3    0    0  ↔
         0
```

Note that even in a multi-router network, and with multiple producers and consumers for *queue.first*, all deliveries will be routed through the queue on the connected broker.

## 2.5.2 Sharded Queue Example

Here is an extension of the above example to illustrate how Qpid Dispatch Router can be used to create a distributed queue in which multiple brokers share the message-queueing load.

```
connector {
  name: broker1
  role: route-container
  host: <hostname>
  port: <port>
  sasl-mechanisms: ANONYMOUS
}

connector {
  name: broker2
  role: route-container
  host: <hostname>
  port: <port>
  sasl-mechanisms: ANONYMOUS
}

address {
  prefix: queue
  waypoint: yes
}

autoLink {
  addr: queue.first
  dir: in
  connection: broker1
}

autoLink {
  addr: queue.first
  dir: out
  connection: broker1
}

autoLink {
  addr: queue.first
  dir: in
  connection: broker2
}

autoLink {
  addr: queue.first
  dir: out
  connection: broker2
}
```



In the above configuration, there are two instances of *queue.first* on brokers 1 and 2. Message traffic from producers to address *queue.first* shall be balanced between the two instance and messages from the queues shall be balanced across the collection of subscribers to the same address.

### 2.5.3 Dynamically Adding Shards

Since configurable entities in the router can also be accessed via the management protocol, we can remotely add a shard to the above example using *qdmanage*:

```
qdmanage create --type org.apache.qpid.dispatch.connector host=<host> port=<port> name= ↵  
broker3  
qdmanage create --type org.apache.qpid.dispatch.router.config.autoLink addr=queue.first dir ↵  
=in connection=broker3  
qdmanage create --type org.apache.qpid.dispatch.router.config.autoLink addr=queue.first dir ↵  
=out connection=broker3
```

## Chapter 3

# Technical Details and Specifications

### 3.1 Client Compatibility

Dispatch Router should, in theory, work with any client that is compatible with AMQP 1.0. The following clients have been tested:

<i>Client</i>	<i>Notes</i>
qpid::messaging	The Qpid messaging clients work with Dispatch Router as long as they are configured to use the 1.0 version of the protocol. To enable AMQP 1.0 in the C++ client, use the {protocol:amqp1.0} connection option.
Proton Reactor	The Proton Reactor API is compatible with Dispatch Router.
Proton Messenger	Messenger works with Dispatch Router.

### 3.2 Addressing

AMQP addresses are used to control the flow of messages across a network of routers. Addresses are used in a number of different places in the AMQP 1.0 protocol. They can be used in a specific message in the `to` and `reply-to` fields of a message's properties. They are also used during the creation of links in the `address` field of a `source` or a `target`.

Addresses designate various kinds of entities in a messaging network:

- Endpoint processes that consume data or offer a service
- Topics that match multiple consumers to multiple producers
- Entities within a messaging broker:
  - Queues
  - Durable Topics
  - Exchanges

The syntax of an AMQP address is opaque as far as the router network is concerned. A syntactical structure may be used by the administrator that creates addresses, but the router treats them as opaque strings. Routers consider addresses to be mobile such that any address may be directly connected to any router in a network and may move around the topology. In cases where messages are broadcast to or balanced across multiple consumers, an address may be connected to multiple routers in the network.

Addresses have semantics associated with them. When an address is created in the network, it is assigned a set of semantics (and access rules) during a process called provisioning. The semantics of an address control how routers behave when they see the address being used.

Address semantics include the following considerations:

- *Routing pattern* - direct, multicast, balanced
- *Undeliverable action* - drop, hold and retry, redirect
- *Reliability* - N destinations, etc.

### 3.2.1 Routing patterns

Routing patterns constrain the paths that a message can take across a network.

<i>Pattern</i>	<i>Description</i>
<i>Direct</i>	Direct routing allows for only one consumer to use an address at a time. Messages (or links) follow the lowest cost path across the network from the sender to the one receiver.
<i>Multicast</i>	Multicast routing allows multiple consumers to use the same address at the same time. Messages are routed such that each consumer receives a copy of the message.
<i>Balanced</i>	Balanced routing also allows multiple consumers to use the same address. In this case, messages are routed to exactly one of the consumers, and the network attempts to balance the traffic load across the set of consumers using the same address.

### 3.2.2 Routing mechanisms

The fact that addresses can be used in different ways suggests that message routing can be accomplished in different ways. Before going into the specifics of the different routing mechanisms, it would be good to first define what is meant by the term *routing*:

In a network built of multiple routers connected by connections (i.e., nodes and edges in a graph), *routing* determines which connection to use to send a message directly to its destination or one step closer to its destination.

Each router serves as the terminus of a collection of incoming and outgoing links. The links either connect directly to endpoints that produce and consume messages, or they connect to other routers in the network along previously established connections.

#### 3.2.2.1 Message routing

Message routing occurs upon delivery of a message and is done based on the address in the message's `to` field.

When a delivery arrives on an incoming link, the router extracts the address from the delivered message's `to` field and looks the address up in its routing table. The lookup results in zero or more outgoing links onto which the message shall be resent.

<i>Delivery</i>	<i>Handling</i>
<i>pre-settled</i>	If the arriving delivery is pre-settled (i.e., fire and forget), the incoming delivery shall be settled by the router, and the outgoing deliveries shall also be pre-settled. In other words, the pre-settled nature of the message delivery is propagated across the network to the message's destination.
<i>unsettled</i>	Unsettled delivery is also propagated across the network. Because unsettled delivery records cannot be discarded, the router tracks the incoming deliveries and keeps the association of the incoming deliveries to the resulting outgoing deliveries. This kept association allows the router to continue to propagate changes in delivery state (settlement and disposition) back and forth along the path which the message traveled.

## 3.3 AMQP Mapping

Dispatch Router is an AMQP router and as such, it provides extensions, code-points, and semantics for routing over AMQP. This page documents the details of Dispatch Router's use of AMQP.

### 3.3.1 Message Annotations

The following Message Annotation fields are defined by Dispatch Router:

<i>Field</i>	<i>Type</i>	<i>Description</i>
x-opt-qd.ingress	string	The identity of the ingress router for a message-routed message. The ingress router is the first router encountered by a transiting message. The router will, if this field is present, leave it unaltered. If the field is not present, the router shall insert the field with its own identity.
x-opt-qd.trace	list of string	The list of routers through which this message-routed message has transited. If this field is not present, the router shall do nothing. If the field is present, the router shall append its own identity to the end of the list.
x-opt-qd.to	string	To-Override for message-routed messages. If this field is present, the address in this field shall be used for routing in lieu of the <i>to</i> field in the message properties. A router may append, remove, or modify this annotation field depending on the policy in place for routing the message.
x-opt-qd.phase	integer	The address-phase, if not zero, for messages flowing between routers.

### 3.3.2 Source/Target Capabilities

The following Capability values are used in Sources and Targets.

<i>Capability</i>	<i>Description</i>
qd.router	This capability is added to sources and targets that are used for inter-router message exchange. This capability denotes a link used for router-control messages flowing between routers.
qd.router-data	This capability is added to sources and targets that are used for inter-router message exchange. This capability denotes a link used for user messages being message-routed across an inter-router connection.

### 3.3.3 Dynamic-Node-Properties

The following dynamic-node-properties are used by Dispatch in Sources.

<i>Property</i>	<i>Description</i>
x-opt-qd.address	The node address describing the destination desired for a dynamic source. If this is absent, the router will terminate any dynamic receivers. If this address is present, the router will use the address to route the dynamic link attach to the proper destination container.

### 3.3.4 Addresses and Address Formats

The following AMQP addresses and address patterns are used within Dispatch Router.

#### 3.3.4.1 Address Patterns

<i>Pattern</i>	<i>Description</i>
<code>_local/&lt;addr&gt;</code>	An address that references a locally attached endpoint. Messages using this address pattern shall not be routed over more than one link.

<i>Pattern</i>	<i>Description</i>
<code>_topo/0/&lt;router&gt;/&lt;addr&gt;</code>	An address that references an endpoint attached to a specific router node in the network topology. Messages with addresses that follow this pattern shall be routed along the shortest path to the specified router. Note that addresses of this form are a-priori routable in that the address itself contains enough information to route the message to its destination. The <i>0</i> component immediately preceding the router-id is a placeholder for an <i>area</i> which may be used in the future if area routing is implemented.
<code>&lt;addr&gt;</code>	A mobile address. An address of this format represents an endpoint or a set of distinct endpoints that are attached to the network in arbitrary locations. It is the responsibility of the router network to determine which router nodes are valid destinations for mobile addresses.

### 3.3.4.2 Supported Addresses

<i>Address</i>	<i>Description</i>
<code>\$management</code>	The management agent on the attached router/container. This address would be used by an endpoint that is a management client/console/tool wishing to access management data from the attached container.
<code>_topo/0/Router.E/\$management</code>	The management agent at Router.E in area 0. This address would be used by a management client wishing to access management data from a specific container that is reachable within the network.
<code>_local/qdhello</code>	The router entity in each of the connected routers. This address is used to communicate with neighbor routers and is exclusively for the HELLO discovery protocol.
<code>_local/qdrouter</code>	The router entity in each of the connected routers. This address is used by a router to communicate with other routers in the network.
<code>_topo/0/Router.E/qdrouter</code>	The router entity at the specifically indicated router. This address form is used by a router to communicate with a specific router that may or may not be a neighbor.

### 3.3.5 Implementation of the AMQP Management Specification

Qpid Dispatch is manageable remotely via AMQP. It is compliant with the emerging AMQP Management specification (draft 9).

Differences from the specification:

- The `name` attribute is not required when an entity is created. If not supplied it will be set to the same value as the system-generated "identity" attribute. Otherwise it is treated as per the standard.
- The `REGISTER` and `DEREGISTER` operations are not implemented. The router automatically discovers peer routers via the router network and makes their management addresses available via the standard `GET-MGMT-NODES` operation. = Management Schema

This chapter documents the set of **management entity types** that define configuration and management of a Dispatch Router. A management entity type has a set of **attributes** that can be read, some attributes can also be updated. Some entity types also support **operations** that can be called.

All management entity types have the following standard attributes:

#### type

The fully qualified type of the entity, e.g. `org.apache.qpid.dispatch.router`. This document uses the short name without the `org.apache.qpid.dispatch` prefix e.g. `router`. The dispatch tools will accept the short or long name.

#### name

A user-generated identity for the entity. This can be used in other entities that need to refer to the named entity.

**identity**

A system-generated identity of the entity. It includes the short type name and some identifying information. E.g. `log/AGENT` or `listener/localhost:amqp`

There are two main categories of management entity type.

**Configuration Entities**

Parameters that can be set in the configuration file (see `qdrouterd.conf(5)` man page) or set at run-time with the `qdmanage(8)` tool.

**Operational Entities**

Run-time status values that can be queried using `qdstat(8)` or `qdmanage(8)` tools.

## 3.4 Configuration Entities

Configuration entities define the attributes allowed in the configuration file (see `qdrouterd.conf(5)`) but you can also create entities once the router is running using the `qdrouterd(8)` tool's `create` operation. Some entities can also be modified using the `update` operation, see the entity descriptions below.

### 3.4.1 container

(DEPRECATED) Attributes related to the AMQP container. This entity has been deprecated. Use the router entity instead.

Operations allowed: `READ`

***containerName* (string, CREATE)**

The name of the AMQP container. If not specified, the container name will be set to a value of the container's choosing. The automatically assigned container name is not guaranteed to be persistent across restarts of the container.

***workerThreads* (integer, default=4, CREATE)**

The number of threads that will be created to process message traffic and other application work (timers, non-amqp file descriptors, etc.) .

***debugDump* (path, CREATE)**

A file to dump debugging information that can't be logged normally.

***saslConfigPath* (path, CREATE)**

Absolute path to the SASL configuration file.

***saslConfigName* (string, CREATE)**

Name of the SASL configuration. This string + `.conf` is the name of the configuration file.

### 3.4.2 router

Tracks peer routers and computes routes to destinations.

Operations allowed: `READ`

***routerId* (string, CREATE)**

(DEPRECATED) Router's unique identity. This attribute has been deprecated. Use `id` instead

***id* (string, CREATE)**

Router's unique identity. One of `id` or `routerId` is required. The router will fail to start without `id` or `routerId`

***mode* (One of [standalone, interior], default=standalone, CREATE)**

In standalone mode, the router operates as a single component. It does not participate in the routing protocol and therefore will not cooperate with other routers. In interior mode, the router operates in cooperation with other interior routers in an interconnected network.

**area (string)**

Unused placeholder.

**helloInterval (integer, default=1, CREATE)**

Interval in seconds between HELLO messages sent to neighbor routers.

**helloMaxAge (integer, default=3, CREATE)**

Time in seconds after which a neighbor is declared lost if no HELLO is received.

**raInterval (integer, default=30, CREATE)**

Interval in seconds between Router-Advertisements sent to all routers in a stable network.

**raIntervalFlux (integer, default=4, CREATE)**

Interval in seconds between Router-Advertisements sent to all routers during topology fluctuations.

**remoteLsMaxAge (integer, default=60, CREATE)**

Time in seconds after which link state is declared stale if no RA is received.

**mobileAddrMaxAge (integer, default=60, CREATE)**

(DEPRECATED) This value is no longer used in the router.

**addrCount (integer)**

Number of addresses known to the router.

**linkCount (integer)**

Number of links attached to the router node.

**nodeCount (integer)**

Number of known peer router nodes.

**workerThreads (integer, default=4, CREATE)**

The number of threads that will be created to process message traffic and other application work (timers, non-amqp file descriptors, etc.) .

**debugDump (path, CREATE)**

A file to dump debugging information that can't be logged normally.

**saslConfigPath (path, CREATE)**

Absolute path to the SASL configuration file.

**saslConfigName (string, default=qdrouterd, CREATE)**

Name of the SASL configuration. This string + *.conf* is the name of the configuration file.

### 3.4.3 listener

Listens for incoming connections to the router.

Operations allowed: CREATE, DELETE, READ

**addr (string, default=127.0.0.1, CREATE)**

(DEPRECATED)IP address: ipv4 or ipv6 literal or a host name. This attribute has been deprecated. Use host instead

**host (string, default=127.0.0.1, CREATE)**

IP address: ipv4 or ipv6 literal or a host name

**port (string, default=amqp, CREATE)**

Port number or symbolic service name.

**protocolFamily (One of [IPv4, IPv6], CREATE)**

[IPv4, IPv6] IPv4: Internet Protocol version 4; IPv6: Internet Protocol version 6. If not specified, the protocol family will be automatically determined from the address.

**role (One of [normal, inter-router, route-container, on-demand], default=normal, CREATE)**

The role of an established connection. In the normal role, the connection is assumed to be used for AMQP clients that are doing normal message delivery over the connection. In the inter-router role, the connection is assumed to be to another router in the network. Inter-router discovery and routing protocols can only be used over inter-router connections. route-container role can be used for router-container connections, for example, a router-broker connection. on-demand role has been deprecated.

**cost (integer, default=1, CREATE)**

For the *inter-router* role only. This value assigns a cost metric to the inter-router connection. The default (and minimum) value is one. Higher values represent higher costs. The cost is used to influence the routing algorithm as it attempts to use the path with the lowest total cost from ingress to egress.

**certDb (path, CREATE)**

The absolute path to the database that contains the public certificates of trusted certificate authorities (CA).

**certFile (path, CREATE)**

The absolute path to the file containing the PEM-formatted public certificate to be used on the local end of any connections using this profile.

**keyFile (path, CREATE)**

The absolute path to the file containing the PEM-formatted private key for the above certificate.

**passwordFile (path, CREATE)**

If the above private key is password protected, this is the absolute path to a file containing the password that unlocks the certificate key.

**password (string, CREATE)**

An alternative to storing the password in a file referenced by passwordFile is to supply the password right here in the configuration file. This option can be used by supplying the password in the *password* option. Don't use both password and passwordFile in the same profile.

**uidFormat (string, CREATE)**

A list of x509 client certificate fields that will be used to build a string that will uniquely identify the client certificate owner. For e.g. a value of *cou* indicates that the uid will consist of c - common name concatenated with o - organization-company name concatenated with u - organization unit; or a value of *o2* indicates that the uid will consist of o (organization name) concatenated with 2 (the sha256 fingerprint of the entire certificate) . Allowed values can be any combination of *c* (ISO3166 two character country code), *s* (state or province), *l* (Locality; generally - city), *o* (Organization - Company Name), *u* (Organization Unit - typically certificate type or brand), *n* (CommonName - typically a user name for client certificates) and *l* (sha1 certificate fingerprint, as displayed in the fingerprints section when looking at a certificate with say a web browser is the hash of the entire certificate) and 2 (sha256 certificate fingerprint) and 5 (sha512 certificate fingerprint).

**displayNameFile (string, CREATE)**

The absolute path to the file containing the unique id to display name mapping

**sslProfileName (string)**

The name of the ssl profile. This is for internal use only. Use the *name* attribute to assign a name to an sslProfile section

**saslMechanisms (string, CREATE)**

Space separated list of accepted SASL authentication mechanisms.

**authenticatePeer (boolean, CREATE)**

yes: Require the peer's identity to be authenticated; no: Do not require any authentication.

**requireEncryption (boolean, CREATE)**

yes: Require the connection to the peer to be encrypted; no: Permit non-encrypted communication with the peer

**requireSsl (boolean, CREATE)**

yes: Require the use of SSL or TLS on the connection; no: Allow clients to connect without SSL or TLS.

**trustedCerts (path, CREATE)**

This optional setting can be used to reduce the set of available CAs for client authentication. If used, this setting must provide the absolute path to a PEM file that contains the trusted certificates.



***maxFrameSize* (integer, default=16384, CREATE)**

Defaults to 16384. If specified, it is the maximum frame size in octets that will be used in the connection-open negotiation with a connected peer. The frame size is the largest contiguous set of uninterrupted data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

***idleTimeoutSeconds* (integer, default=16, CREATE)**

The idle timeout, in seconds, for connections through this listener. If no frames are received on the connection for this time interval, the connection shall be closed.

***requirePeerAuth* (boolean, CREATE)**

(DEPRECATED) This attribute is now controlled by the `authenticatePeer` attribute.

***allowUnsecured* (boolean, CREATE)**

(DEPRECATED) This attribute is now controlled by the `requireEncryption` attribute.

***allowNoSasl* (boolean, CREATE)**

(DEPRECATED) This attribute is now controlled by the `authenticatePeer` attribute.

***stripAnnotations* (One of [in, out, both, no], default=both, CREATE)**

[in, out, both, no] in: Strip the dispatch router specific annotations only on ingress; out: Strip the dispatch router specific annotations only on egress; both: Strip the dispatch router specific annotations on both ingress and egress; no - do not strip dispatch router specific annotations

***linkCapacity* (integer, CREATE)**

The capacity of links within this connection, in terms of message deliveries. The capacity is the number of messages that can be in-flight concurrently for each link.

### 3.4.4 connector

Establishes an outgoing connection from the router.

Operations allowed: CREATE, DELETE, READ

***addr* (string, default=127.0.0.1, CREATE)**

(DEPRECATED) IP address: ipv4 or ipv6 literal or a host name. This attribute has been deprecated. Use `host` instead

***host* (string, default=127.0.0.1, CREATE)**

IP address: ipv4 or ipv6 literal or a host name

***port* (string, default=amqp, CREATE)**

Port number or symbolic service name.

***protocolFamily* (One of [IPv4, IPv6], CREATE)**

[IPv4, IPv6] IPv4: Internet Protocol version 4; IPv6: Internet Protocol version 6. If not specified, the protocol family will be automatically determined from the address.

***role* (One of [normal, inter-router, route-container, on-demand], default=normal, CREATE)**

The role of an established connection. In the normal role, the connection is assumed to be used for AMQP clients that are doing normal message delivery over the connection. In the inter-router role, the connection is assumed to be to another router in the network. Inter-router discovery and routing protocols can only be used over inter-router connections. route-container role can be used for router-container connections, for example, a router-broker connection. on-demand role has been deprecated.

***cost* (integer, default=1, CREATE)**

For the *inter-router* role only. This value assigns a cost metric to the inter-router connection. The default (and minimum) value is one. Higher values represent higher costs. The cost is used to influence the routing algorithm as it attempts to use the path with the lowest total cost from ingress to egress.

***certDb* (path, CREATE)**

The absolute path to the database that contains the public certificates of trusted certificate authorities (CA).

***certFile (path, CREATE)***

The absolute path to the file containing the PEM-formatted public certificate to be used on the local end of any connections using this profile.

***keyFile (path, CREATE)***

The absolute path to the file containing the PEM-formatted private key for the above certificate.

***passwordFile (path, CREATE)***

If the above private key is password protected, this is the absolute path to a file containing the password that unlocks the certificate key.

***password (string, CREATE)***

An alternative to storing the password in a file referenced by *passwordFile* is to supply the password right here in the configuration file. This option can be used by supplying the password in the *password* option. Don't use both *password* and *passwordFile* in the same profile.

***uidFormat (string, CREATE)***

A list of x509 client certificate fields that will be used to build a string that will uniquely identify the client certificate owner. For e.g. a value of *cou* indicates that the uid will consist of c - common name concatenated with o - organization-company name concatenated with u - organization unit; or a value of *o2* indicates that the uid will consist of o (organization name) concatenated with 2 (the sha256 fingerprint of the entire certificate) . Allowed values can be any combination of *c* (ISO3166 two character country code), *s* (state or province), *l* (Locality; generally - city), *o* (Organization - Company Name), *u* (Organization Unit - typically certificate type or brand), *n* (CommonName - typically a user name for client certificates) and *l* (sha1 certificate fingerprint, as displayed in the fingerprints section when looking at a certificate with say a web browser is the hash of the entire certificate) and 2 (sha256 certificate fingerprint) and 5 (sha512 certificate fingerprint).

***displayNameFile (string, CREATE)***

The absolute path to the file containing the unique id to display name mapping

***sslProfileName (string)***

The name of the ssl profile. This is for internal use only. Use the *name* attribute to assign a name to an sslProfile section

***saslMechanisms (string, CREATE)***

Space separated list of accepted SASL authentication mechanisms.

***allowRedirect (boolean, default=True, CREATE)***

Allow the peer to redirect this connection to another address.

***maxFrameSize (integer, default=65536, CREATE)***

Maximum frame size in octets that will be used in the connection-open negotiation with a connected peer. The frame size is the largest contiguous set of uninterrupted data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

***idleTimeoutSeconds (integer, default=16, CREATE)***

The idle timeout, in seconds, for connections through this connector. If no frames are received on the connection for this time interval, the connection shall be closed.

***stripAnnotations (One of [in, out, both, no], default=both, CREATE)***

[*in, out, both, no*] *in*: Strip the dispatch router specific annotations only on ingress; *out*: Strip the dispatch router specific annotations only on egress; *both*: Strip the dispatch router specific annotations on both ingress and egress; *no* - do not strip dispatch router specific annotations

***linkCapacity (integer, CREATE)***

The capacity of links within this connection, in terms of message deliveries. The capacity is the number of messages that can be in-flight concurrently for each link.

***verifyHostName (boolean, default=True, CREATE)***

*yes*: Ensures that when initiating a connection (as a client) the host name in the URL to which this connector connects to matches the host name in the digital certificate that the peer sends back as part of the SSL connection; *no*: Does not perform host name verification

***saslUsername* (string, CREATE)**

The user name that the connector is using to connect to a peer.

***saslPassword* (string, CREATE)**

The password that the connector is using to connect to a peer.

### 3.4.5 log

Configure logging for a particular module. You can use the `UPDATE` operation to change log settings while the router is running.

Operations allowed: `UPDATE`, `READ`

***module* (One of [`ROUTER`, `ROUTER_CORE`, `ROUTER_HELLO`, `ROUTER_LS`, `ROUTER_MA`, `MESSAGE`, `SERVER`, `AGENT`,**

Module to configure. The special module `DEFAULT` specifies defaults for all modules.

***enable* (string, default=default, required, UPDATE)**

Levels are: trace, debug, info, notice, warning, error, critical. The enable string is a comma-separated list of levels. A level may have a trailing + to enable that level and above. For example `trace,debug,warning+` means enable trace, debug, warning, error and critical. The value `none` means disable logging for the module. The value `default` means use the value from the `DEFAULT` module.

***timestamp* (boolean, UPDATE)**

Include timestamp in log messages.

***source* (boolean, UPDATE)**

Include source file and line number in log messages.

***output* (string, UPDATE)**

Where to send log messages. Can be `stderr`, `syslog` or a file name.

### 3.4.6 fixedAddress

(DEPRECATED) Establishes treatment for addresses starting with a prefix. This entity has been deprecated. Use `address` instead

Operations allowed: `CREATE`, `READ`

***prefix* (string, required, CREATE)**

The address prefix (always starting with /).

***phase* (integer, CREATE)**

The phase of a multi-hop address passing through one or more waypoints.

***fanout* (One of [`multiple`, `single`], default=multiple, CREATE)**

One of `multiple` or `single`. Multiple fanout is a non-competing pattern. If there are multiple consumers using the same address, each consumer will receive its own copy of every message sent to the address. Single fanout is a competing pattern where each message is sent to only one consumer.

***bias* (One of [`closest`, `spread`], default=closest, CREATE)**

Only if fanout is single. One of `closest` or `spread`. Closest bias means that messages to an address will always be delivered to the closest (lowest cost) subscribed consumer. Spread bias will distribute the messages across subscribers in an approximately even manner.

### 3.4.7 waypoint

(DEPRECATED) A remote node that messages for an address pass through. This entity has been deprecated. Use `autoLink` instead

Operations allowed: `CREATE`, `DELETE`, `READ`

***address* (string, required, CREATE)**

The AMQP address of the waypoint.

***connector* (string, required, CREATE)**

The name of the on-demand connector used to reach the waypoint's container.

***inPhase* (integer, default=-1, CREATE)**

The phase of the address as it is routed *to* the waypoint.

***outPhase* (integer, default=-1, CREATE)**

The phase of the address as it is routed *from* the waypoint.

### 3.4.8 linkRoutePattern

(DEPRECATED) An address pattern to match against link sources and targets to cause the router to link-route the attach across the network to a remote node. This entity has been deprecated. Use `linkRoute` instead

Operations allowed: `CREATE`, `READ`

***prefix* (string, required, CREATE)**

An address prefix to match against target and source addresses. This pattern must be of the form `<text>.<textI>.<textN>` or `<text>` or `<text>.` and matches any address that contains that prefix. For example, if the prefix is set to `org.apache` (or `org.apache.`), any address that has the prefix `org.apache` (like `org.apache.dev`) will match. Note that a prefix must not start with a `(.)`, can end in a `(.)` and can contain zero or more dots `(.)`. Any characters between the dots are simply treated as part of the address

***dir* (One of [*in*, *out*, *both*], default=*both*, CREATE)**

Link direction for match: *in* matches only links inbound to the client; *out* matches only links outbound from the client; *both* matches any link.

***connector* (string, CREATE)**

The name of the on-demand connector used to reach the target node's container. If this value is not provided, it means that the target container is expected to be connected to a different router in the network. This prevents links to a link-routable address from being misinterpreted as message-routing links when there is no route to a valid destination available.

### 3.4.9 address

Entity type for address configuration. This is used to configure the treatment of message-routed deliveries within a particular address-space. The configuration controls distribution and address phasing.

Operations allowed: `CREATE`, `DELETE`, `READ`

***prefix* (string, required, CREATE)**

The address prefix for the configured settings

***distribution* (One of [*multicast*, *closest*, *balanced*], default=*balanced*, CREATE)**

Treatment of traffic associated with the address

***waypoint* (boolean, CREATE)**

Designates this address space as being used for waypoints. This will cause the proper address-phasing to be used.

***ingressPhase* (integer, CREATE)**

Advanced - Override the ingress phase for this address

***egressPhase* (integer, CREATE)**

Advanced - Override the egress phase for this address

### 3.4.10 linkRoute

Entity type for link-route configuration. This is used to identify remote containers that shall be destinations for routed link-attaches. The link-routing configuration applies to an addressing space defined by a prefix.

Operations allowed: CREATE, DELETE, READ

***prefix* (string, required, CREATE)**

The address prefix for the configured settings

***containerId* (string, CREATE)**

ContainerID for the target container

***connection* (string, CREATE)**

The name from a connector or listener

***distribution* (One of [*linkBalanced*], default=*linkBalanced*, CREATE)**

Treatment of traffic associated with the address

***dir* (One of [*in*, *out*], required, CREATE)**

The permitted direction of links: *in* means client senders; *out* means client receivers

***operStatus* (One of [*inactive*, *active*])**

The operational status of this linkRoute: *inactive* - The remote container is not connected; *active* - the remote container is connected and ready to accept link routed attachments.

### 3.4.11 autoLink

Entity type for configuring auto-links. Auto-links are links whose lifecycle is managed by the router. These are typically used to attach to waypoints on remote containers (brokers, etc.).

Operations allowed: CREATE, DELETE, READ

***addr* (string, required, CREATE)**

The address of the provisioned object

***dir* (One of [*in*, *out*], required, CREATE)**

The direction of the link to be created. *In* means into the router, *out* means out of the router.

***phase* (integer, CREATE)**

The address phase for this link. Defaults to *0* for *out* links and *1* for *in* links.

***containerId* (string, CREATE)**

ContainerID for the target container

***connection* (string, CREATE)**

The name from a connector or listener

***linkRef* (string)**

Reference to the org.apache.qpid.dispatch.router.link if the link exists

***operStatus* (One of [*inactive*, *attaching*, *failed*, *active*, *quiescing*, *idle*])**

The operational status of this autoLink: *inactive* - The remote container is not connected; *attaching* - the link is attaching to the remote node; *failed* - the link attach failed; *active* - the link is attached and operational; *quiescing* - the link is transitioning to idle state; *idle* - the link is attached but there are no deliveries flowing and no unsettled deliveries.

***lastError* (string)**

The error description from the last attach failure

---

### 3.4.12 console

Start a websocket/tcp proxy and http file server to serve the web console

Operations allowed: READ

***listener (string)***

The name of the listener to send the proxied tcp traffic to.

***wsport (integer, default=5673)***

port on which to listen for websocket traffic

***proxy (string)***

The full path to the proxy program to run.

***home (string)***

The full path to the html/css/js files for the console.

***args (string)***

Optional args to pass the proxy program for logging, authentication, etc.

### 3.4.13 policy

Defines global connection limit

Operations allowed: READ

***maximumConnections (integer, CREATE)***

Global maximum number of concurrent client connections allowed. Zero implies no limit. This limit is always enforced even if no other policy settings have been defined.

***enableAccessRules (boolean, CREATE)***

Enable user rule set processing and connection denial.

***policyFolder (path, CREATE)***

The absolute path to a folder that holds policyRuleset definition .json files. For a small system the rulesets may all be defined in this file. At a larger scale it is better to have the policy files in their own folder and to have none of the rulesets defined here. All rulesets in all .json files in this folder are processed.

***defaultApplication (string, CREATE)***

Application policyRuleset to use for connections with no open.hostname or a hostname that does not match any existing policy. For users that don't wish to use open.hostname or any multi-tenancy feature, this default policy can be the only policy in effect for the network.

***defaultApplicationEnabled (boolean, CREATE)***

Enable defaultApplication policy fallback logic.

***connectionsProcessed (integer) , connectionsDenied (integer) , connectionsCurrent (integer)***

=== policyRuleset

Per application definition of the locations from which users may connect and the groups to which users belong.

Operations allowed: CREATE, READ

***applicationName (string, required)***

The application name.

***maxConnections (integer, CREATE)***

Maximum number of concurrent client connections allowed. Zero implies no limit.

***maxConnPerUser* (integer, CREATE)**

Maximum number of concurrent client connections allowed for any single user. Zero implies no limit.

***maxConnPerHost* (integer, CREATE)**

Maximum number of concurrent client connections allowed for any remote host. Zero implies no limit.

***userGroups* (map, CREATE)**

A map where each key is a user group name and the corresponding value is a CSV string naming the users in that group. Users who are assigned to one or more groups are deemed *restricted*. Restricted users are subject to connection ingress policy and are assigned policy settings based on the assigned user groups. Unrestricted users may be allowed or denied. If unrestricted users are allowed to connect then they are assigned to user group default.

***ingressHostGroups* (map, CREATE)**

A map where each key is an ingress host group name and the corresponding value is a CSV string naming the IP addresses or address ranges in that group. IP addresses may be FQDN strings or numeric IPv4 or IPv6 host addresses. A host range is two host addresses of the same address family separated with a hyphen. The wildcard host address *\** represents any host address.

***ingressPolicies* (map, CREATE)**

A map where each key is a user group name and the corresponding value is a CSV string naming the ingress host group names that restrict the ingress host for the user group. Users who are members of the user group are allowed to connect only from a host in one of the named ingress host groups.

***connectionAllowDefault* (boolean, CREATE)**

Unrestricted users, those who are not members of a defined user group, are allowed to connect to this application. Unrestricted users are assigned to the *default* user group and receive *default* settings.

***settings* (map, CREATE)**

A map where each key is a user group name and the value is a map of the corresponding settings for that group.

## 3.5 Operational Entities

Operational entities provide statistics and other run-time attributes of the router. The `qdstat(8)` tool provides a convenient way to query run-time statistics. You can also use the general-purpose management tool `qdmanage(8)` to query operational attributes.

### 3.5.1 org.amqp.management

The standard AMQP management node interface.

Operations allowed: QUERY, GET-TYPES, GET-ANNOTATIONS, GET-OPERATIONS, GET-ATTRIBUTES, GET-MGMT-NODES, READ

#### 3.5.1.1 Operation GET-TYPES

Get the set of entity types and their inheritance relationships

REQUEST PROPERTIES

***entityType* (string)**

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity* (string)**

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of the entity types (strings) that it extends.

### 3.5.1.2 Operation GET-ATTRIBUTES

Get the set of entity types and the annotations they implement

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is a list (of strings) of attributes on that entity type.

### 3.5.1.3 Operation GET-OPERATIONS

Get the set of entity types and the operations they support

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of operation names (strings) that it supports.

### 3.5.1.4 Operation GET-ANNOTATIONS

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of annotations (strings) that it implements.

### 3.5.1.5 Operation QUERY

Query for attribute values of multiple entities.

**Request body (map)** A map containing the key `attributeNames` with value a list of (string) attribute names to return. If the list or the map is empty or the body is missing all attributes are returned.

REQUEST PROPERTIES

***count*** (integer)

If set, specifies the number of entries from the result set to return. If not set return all from `offset`

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

---



***identity (string)***

Set to the value `self`

***offset (integer)***

If set, specifies the number of the first element of the result set to be returned.

**Response body (map)** A map with two entries. `attributeNames` is a list of the attribute names returned. `results` is a list of lists each containing the attribute values for a single entity in the same order as the names in the `attributeNames` entry. If an attribute name is not applicable for an entity then the corresponding value is `null`

**RESPONSE PROPERTIES*****count (integer)***

Number of results returned

***identity (string)***

Set to the value `self`

**3.5.1.6 Operation GET-MGMT-NODES**

Get the addresses of all management nodes known to this router

**REQUEST PROPERTIES*****identity (string)***

Set to the value `self`

**Response body (list)** A list of addresses (strings) of management nodes known to this management node.

**3.5.2 management**

Qpid dispatch router extensions to the standard `org.amqp.management` interface.

Operations allowed: `GET-SCHEMA`, `GET-JSON-SCHEMA`, `GET-LOG`, `PROFILE`, `QUERY`, `GET-TYPES`, `GET-ANNOTATIONS`, `GET-OPERATIONS`, `GET-ATTRIBUTES`, `GET-MGMT-NODES`, `READ`

**3.5.2.1 Operation GET-SCHEMA-JSON**

Get the `qdrouterd` schema for this router in JSON format

**REQUEST PROPERTIES*****indent (integer)***

Number of spaces to indent the formatted result. If not specified, the result is in minimal format, no unnecessary spaces or newlines.

***identity (string)***

Set to the value `self`

**Response body (string)** The `qdrouter` schema as a JSON string.

---

### 3.5.2.2 Operation GET-LOG

Get recent log entries from the router.

REQUEST PROPERTIES

**limit (integer)**

Maximum number of log entries to get.

**identity (string)**

Set to the value `self`

**Response body (string)** A list of log entries where each entry is a list of: module name(string), level name(string), message text(string), file name(string or None), line number(integer or None) , timestamp(integer)

### 3.5.2.3 Operation GET-SCHEMA

Get the qdrouterd schema for this router in AMQP map format

REQUEST PROPERTIES

**identity (string)**

Set to the value `self`

**Response body (map)** The qdrouter schema as a map.

## 3.5.3 router.link

Link to another AMQP endpoint: router node, client or other AMQP process.

Operations allowed: `UPDATE`, `READ`

**adminStatus** (One of [*enabled*, *disabled*], **default=enabled**, **UPDATE**) , **operStatus** (One of [*up*, *down*, *quiescing*, *idle*] ) , **linkName** (

Name assigned to the link in the Attach.

**linkType** (One of [*endpoint*, *router-control*, *inter-router*])

Type of link: endpoint: a link to a normally connected endpoint; inter-router: a link to another router in the network.

**linkDir** (One of [*in*, *out*])

Direction of delivery flow over the link, inbound or outbound to or from the router.

**owningAddr** (string)

Address assigned to this link during attach: The target for inbound links or the source for outbound links.

**capacity** (integer)

The capacity, in deliveries, for the link. The number of undelivered plus unsettled deliveries shall not exceed the capacity. This is enforced by link flow control.

**peer** (string)

Identifier of the paired link if this is an attach-routed link.

**undeliveredCount** (integer)

The number of undelivered messages pending for the link.

**unsettledCount** (integer)

The number of unsettled deliveries awaiting settlement on the link

**deliveryCount** (integer)

The total number of deliveries that have traversed this link.

### 3.5.4 router.address

AMQP address managed by the router.

Operations allowed: READ

***distribution* (One of [flood, multicast, closest, balanced, linkBalanced])**

Forwarding treatment for the address: flood - messages delivered to all subscribers along all available paths (this will cause duplicate deliveries if there are redundant paths); multi - one copy of each message delivered to all subscribers; anyClosest - messages delivered to only the closest subscriber; anyBalanced - messages delivered to one subscriber with load balanced across subscribers; linkBalanced - for link-routing, link attaches balanced across destinations.

***inProcess* (integer)**

The number of in-process subscribers for this address

***subscriberCount* (integer)**

The number of local subscribers for this address (i.e. attached to this router)

***remoteCount* (integer)**

The number of remote routers that have at least one subscriber to this address

***containerCount* (integer)**

The number of attached containers that serve this route address

***deliveriesIngress* (integer)**

The number of deliveries to this address that entered the router network on this router

***deliveriesEgress* (integer)**

The number of deliveries to this address that exited the router network on this router

***deliveriesTransit* (integer)**

The number of deliveries to this address that transited this router to another router

***deliveriesToContainer* (integer)**

The number of deliveries to this address that were given to an in-process subscriber

***deliveriesFromContainer* (integer)**

The number of deliveries to this address that were originated from an in-process entity

***key* (string)**

Internal unique (to this router) key to identify the address

***remoteHostRouters* (list)**

List of remote routers on which there is a destination for this address.

***transitOutstanding* (list)**

List of numbers of outstanding deliveries across a transit (inter-router) link for this address. This is for balanced distribution only.

***trackedDeliveries* (integer)**

Number of transit deliveries being tracked for this address (for balanced distribution).

### 3.5.5 router.node

Remote router node connected to this router.

Operations allowed: READ

***id* (string)**

Remote node identifier.

---

***instance (integer)***

Remote node boot number.

***linkState (list)***

List of remote node's neighbours.

***nextHop (string)***

Neighbour ID of next hop to remote node from here.

***validOrigins (list)***

List of valid origin nodes for messages arriving via the remote node, used for duplicate elimination in redundant networks.

***address (string)***

Address of the remote node

***routerLink (entityId)***

Local link to remote node

***cost (integer)***

Reachability cost

### 3.5.6 connection

Connections to the router's container.

Operations allowed: `READ`

***container (string)***

The container for this connection

***opened (boolean)***

The connection has been opened (i.e. AMQP OPEN)

***host (string)***

IP address and port number in the form `addr:port`.

***dir (One of [in, out])***

Direction of connection establishment in or out of the router.

***role (string) , isAuthenticated (boolean)***

Indicates whether the identity of the connection's user is authentic.

***isEncrypted (boolean)***

Indicates whether the connection content is encrypted.

***sasl (string)***

SASL mechanism in effect for authentication.

***user (string)***

Identity of the authenticated user.

***ssl (boolean)***

True iff SSL/TLS is in effect for this connection.

***sslProto (string)***

SSL protocol name

***sslCipher (string)***

SSL cipher name

***sslSsf (integer)***

SSL strength factor in effect

***properties (map)***

Connection properties supplied by the peer.

---

### 3.5.7 allocator

Memory allocation pool.

Operations allowed: `READ`

*typeName* (string) , *typeSize* (integer) , *transferBatchSize* (integer) , *localFreeListMax* (integer) , *globalFreeListMax* (integer) , *totalFreeListMax* (integer)

=== policyStats

Per application connection and access statistics.

Operations allowed: `READ`

*applicationName* (string)

The application name.

*connectionsApproved* (integer) , *connectionsDenied* (integer) , *connectionsCurrent* (integer) , *perUserState* (map)

A map where the key is the authenticated user name and the value is a list of the user's connections.

*perHostState* (map)

A map where the key is the host name and the value is a list of the host's connections.

*sessionDenied* (integer) , *senderDenied* (integer) , *receiverDenied* (integer)

== Management Operations

The *qdstat(8)* and *qdmange(8)* tools allow you to view or modify management entity attributes. They work by invoking **management operations**. You can invoke these operations from any AMQP client by sending a message with the appropriate properties and body to the *\$management* address. The message should have a *reply-to* address indicating where the response should be sent.

### 3.5.8 Operations for all entity types

#### 3.5.8.1 Operation READ

Read attributes of an entity

REQUEST PROPERTIES

*type* (string)

Type of desired entity.

*name* (string)

Name of desired entity. Must supply name or identity.

*identity* (string)

Identity of desired entity. Must supply name or identity.

**Response body (map)** Attributes of the entity

#### 3.5.8.2 Operation CREATE

Create a new entity.

**Request body (map, required)** Attributes for the new entity. Can include name and/or type.

REQUEST PROPERTIES

*type* (string, required)

Type of new entity.

*name* (string)

Name of new entity. Optional, defaults to identity.

**Response body (map)** Attributes of the entity

### 3.5.8.3 Operation UPDATE

Update attributes of an entity

**Request body (map)** Attributes to update for the entity. Can include name or identity.

REQUEST PROPERTIES

***type* (string)**

Type of desired entity.

***name* (string)**

Name of desired entity. Must supply name or identity.

***identity* (string)**

Identity of desired entity. Must supply name or identity.

**Response body (map)** Updated attributes of the entity

### 3.5.8.4 Operation DELETE

Delete an entity

REQUEST PROPERTIES

***type* (string)**

Type of desired entity.

***name* (string)**

Name of desired entity. Must supply name or identity.

***identity* (string)**

Identity of desired entity. Must supply name or identity.

## 3.5.9 Operations for *org.amqp.management* entity type

### 3.5.9.1 Operation GET-TYPES

Get the set of entity types and their inheritance relationships

REQUEST PROPERTIES

***entityType* (string)**

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity* (string)**

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of the entity types (strings) that it extends.

---

### 3.5.9.2 Operation GET-ATTRIBUTES

Get the set of entity types and the annotations they implement

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is a list (of strings) of attributes on that entity type.

### 3.5.9.3 Operation GET-OPERATIONS

Get the set of entity types and the operations they support

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of operation names (strings) that it supports.

### 3.5.9.4 Operation GET-ANNOTATIONS

REQUEST PROPERTIES

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

***identity*** (string)

Set to the value `self`

**Response body (map)** A map where each key is an entity type name (string) and the corresponding value is the list of annotations (strings) that it implements.

### 3.5.9.5 Operation QUERY

Query for attribute values of multiple entities.

**Request body (map)** A map containing the key `attributeNames` with value a list of (string) attribute names to return. If the list or the map is empty or the body is missing all attributes are returned.

REQUEST PROPERTIES

***count*** (integer)

If set, specifies the number of entries from the result set to return. If not set return all from `offset`

***entityType*** (string)

If set, restrict query results to entities that extend (directly or indirectly) this type

---

***identity* (string)**

Set to the value `self`

***offset* (integer)**

If set, specifies the number of the first element of the result set to be returned.

**Response body (map)** A map with two entries. `attributeNames` is a list of the attribute names returned. `results` is a list of lists each containing the attribute values for a single entity in the same order as the names in the `attributeNames` entry. If an attribute name is not applicable for an entity then the corresponding value is `null`

## RESPONSE PROPERTIES

***count* (integer)**

Number of results returned

***identity* (string)**

Set to the value `self`

**3.5.9.6 Operation GET-MGMT-NODES**

Get the addresses of all management nodes known to this router

## REQUEST PROPERTIES

***identity* (string)**

Set to the value `self`

**Response body (list)** A list of addresses (strings) of management nodes known to this management node.

**3.5.10 Operations for *management* entity type****3.5.10.1 Operation GET-SCHEMA-JSON**

Get the qdrouterd schema for this router in JSON format

## REQUEST PROPERTIES

***indent* (integer)**

Number of spaces to indent the formatted result. If not specified, the result is in minimal format, no unnecessary spaces or newlines.

***identity* (string)**

Set to the value `self`

**Response body (string)** The qdrouter schema as a JSON string.

**3.5.10.2 Operation GET-LOG**

Get recent log entries from the router.

## REQUEST PROPERTIES

***limit* (integer)**

Maximum number of log entries to get.

***identity* (string)**

Set to the value `self`

**Response body (string)** A list of log entries where each entry is a list of: module name(string), level name(string), message text(string), file name(string or None), line number(integer or None) , timestamp(integer)



### 3.5.10.3 Operation GET-SCHEMA

Get the qdrouterd schema for this router in AMQP map format

REQUEST PROPERTIES

*identity* (string)

Set to the value `self`

**Response body (map)** The qdrouter schema as a map.

## Chapter 4

# Console

### 4.1 Console overview

The console is an HTML based web site that displays information about a qpid dispatch router network.

The console requires an HTML web server that can serve static html, javascript, style sheets, and images.

The current version of the is read-only. The ability to call management methods that change the running of the router network is planned for a future version of console.

The console only provides limited information about the clients that are attached to the router network and is therefore more appropriate for administrators needing to know the layout and health of the router network.

### 4.2 Console installation

#### 4.2.1 Prerequisites

The following need to be installed before running a console:

- One or more dispatch routers. See the documentation for the dispatch router for help in starting a router network.
- node.js This is needed to provide a proxy between the console's websocket traffic and tcp.
- A web server. This can be any server capable of serving static html/js/css/image files.

A nodejs proxy is distributed with proton. To start the proton's nodejs proxy:

```
cd ~/rh-qpid-proton/examples/javascript/messenger
node proxy.js &
```

This will start the proxy listening to ws traffic on port 5673 and translating it to tcp on port 5672. One of the routers in the network needs to have a listener configured on port 5672. That listener's role should be *normal*. For example:

```
listener {
  host: 0.0.0.0
  role: normal
  port: amqp
  saslMechanisms: ANONYMOUS
}
```

### 4.2.2 The console files

The files for the console are located under the console directory in the source tree: `app/ bower_components/ css/ img/ index.html lib/ plugin/ vendor.js`

Copy these files to a directory under the the html or webapps directory of your web server. For example, for apache tomcat the files should be under `webapps/dispatch`. Then the console is available as `http://localhost:8080/dispatch`

## 4.3 Console operation

### 4.3.1 Logging in to a router network

The console communicates to the router network using the proton javascript bindings. When run from a web page, the proton bindings use web sockets to send and receive commands. However, the dispatch router requires tcp. Therefore a web-sockets to tcp proxy is used.

### Qpid Dispatch Router Console

⚙️ Connect

Enter the address and port of a **Qpid Dispatch Router** to connect..

The port should be a websockets <==> tcp proxy.

When Autostart is checked, you will be automatically logged in to the router the next time you start the console.

**Address:**

**Port:**

**Autostart:** ☐

Enter the address of a proxy that is connected to a router in the network.

User name and password are not used at this time.

The Autostart checkbox, when checked, will automatically log in with the previous host:port the next time you start the console.

### 4.3.2 Overview page

On the overview page, aggregate information about routers, addresses, and connections is displayed.

Qpid Dispatch Router Console

Connect

Overview

Topology

List

Charts

Schema

Routers

QDR.A

QDR.B

QDR.C

QDR.D

QDR.X

QDR.Y

Addresses

\$ \_management\_internal

\$displayname

\$management (mobile)

\$management (local)

QDR.A

QDR.B

QDR.C

QDR.D

QDR.X

QDR.Y

qdhello

qdrouter (local)

qdrouter (unknown: T)

qdrouter.ma (unknown: T)

qdrouter.ma (local)

temp.HuS\_uNRntbwi\_38

Connections

Addresses

address	class	phase	in-proc	local	remote	in	out
qdhello	local		1	20	0	0	0
qdrouter	local		1	0	0	0	0
qdrouter.ma	local		1	0	0	0	0
qdrouter	unknown: T		1	0	30	0	0
qdrouter.ma	unknown: T		1	0	30	0	0
\$managem...	mobile	0	1	0	0	196	0
\$managem...	local		1	0	0	1,201	0
\$ _manage...	local		1	0	0	0	0
\$displayna...	local		1	0	0	0	0
QDR.B	router		0	0	5	1,201	0
QDR.C	router		0	0	5	1,201	0
QDR.X	router		0	0	5	0	0
QDR.D	router		0	0	5	1,201	0
QDR.Y	router		0	0	5	1,200	0
QDR.A	router		0	0	5	1,222	0
temp.HuS_...	local		0	1	0	0	89

### 4.3.3 Topology page

This page displays the router network in a graphical form showing how the routers are connected and information about the individual routers and links.

Qpid Dispatch Router Console

Connect

Overview

☆ Topology

List

Charts

Schema

Connection Info

Attribute	Value
name	connection/localhost.local...
container	516aa3a8-0432-406e-b5f...
dir	in
host	localhost.localdomain:57130
identity	9
isAuthenticated	false
isEncrypted	false
opened	true
properties	
role	normal
sasl	
ssl	false
sslCipher	
sslProto	
sslSsf	
type	org.apache.qpid.dispatch....
user	anonymous

Router

Client

```
graph LR; Client((Client)) --> QDR.X((QDR.X)); QDR.X --> QDR.A((QDR.A)); QDR.X --> QDR.B((QDR.B)); QDR.A --> QDR.C((QDR.C)); QDR.A --> QDR.D((QDR.D)); QDR.B --> QDR.C((QDR.C)); QDR.B --> QDR.D((QDR.D)); QDR.C --> QDR.Y((QDR.Y)); QDR.D --> QDR.Y((QDR.Y));
```

4.3.4 Router entity details page

Qpid Dispatch Router Console

⚙️ Connect

Overview

☆ Topology

**☰ List**

📊 Charts

☰ Schema

QDR.A

QDR.B

QDR.C

QDR.D

QDR.X

QDR.Y

Policy Stats

Allocator

Container

Log

Router config.auto Link

Connector

Policy Ruleset

Waypoint

Fixed Address

Link Route Pattern

Listener

Policy

Router config.address

**Router node**

Router address

Router link

Connection

Router config.link Route

Router

router.node/QDR.A

router.node/QDR.B

router.node/QDR.C

router.node/QDR.X

router.node/QDR.D

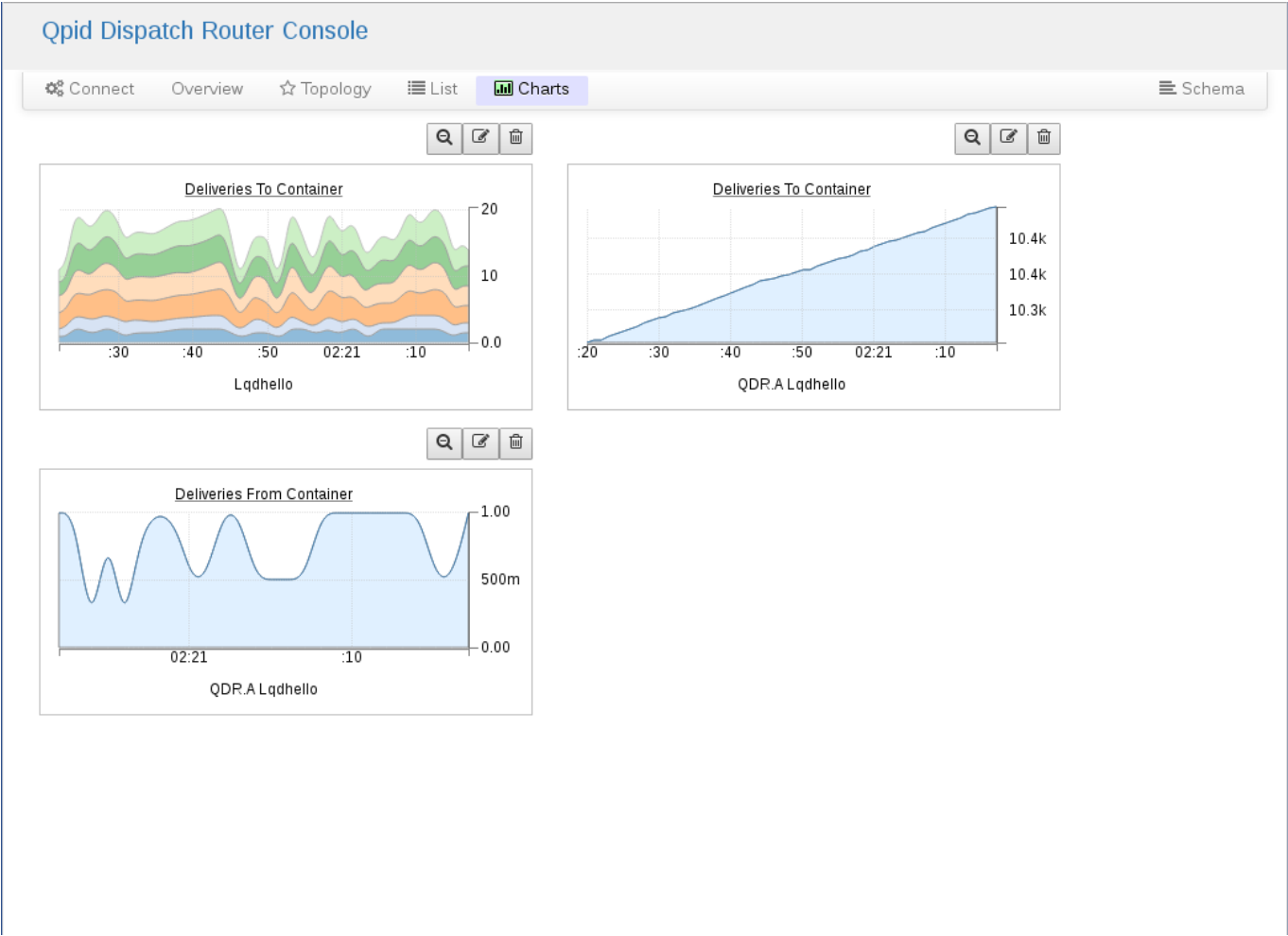
router.node/QDR.Y

Attribute	Value
routerLink	
nextHop	(self)
name	router.node/QDR.A
validOrigins	[]
linkState	["QDR.D","QDR.X","QDR.B","QDR.C"]
instance	1,463,679,202
cost	
address	amqp:/_topo/0/QDR.A
type	org.apache.qpid.dispatch.router.node
id	QDR.A
identity	router.node/QDR.A

Displays detailed information about entities such as routers, links, addresses, memory.

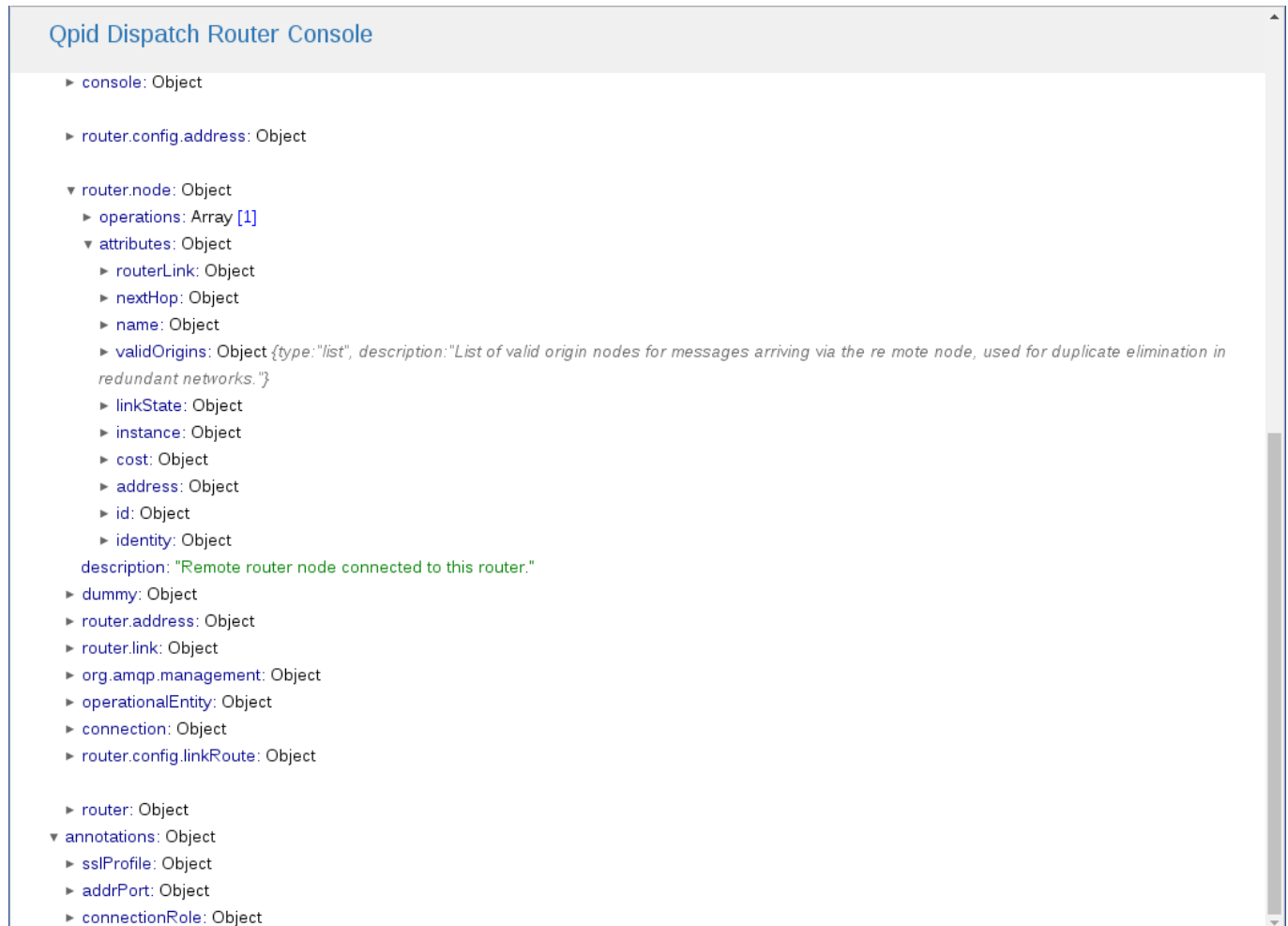
Numeric attributes can be graphed by clicking on the graph icon.

4.3.5 Charts page



This page displays graphs of numeric values that are on the entity details page.

### 4.3.6 Schema page



The screenshot displays the 'Qpid Dispatch Router Console' interface. It shows a hierarchical JSON schema for the router configuration. The schema is organized into a tree structure with expandable/collapsible nodes. The root node is 'console: Object'. Under it, there are several other objects: 'router.config.address: Object', 'router.node: Object', 'dummy: Object', 'router.address: Object', 'router.link: Object', 'org.amqp.management: Object', 'operationalEntity: Object', 'connection: Object', 'router.config.linkRoute: Object', 'router: Object', 'annotations: Object', 'sslProfile: Object', 'addrPort: Object', and 'connectionRole: Object'. The 'router.node: Object' node is expanded, showing its sub-objects: 'operations: Array [1]', 'attributes: Object', 'validOrigins: Object', 'linkState: Object', 'instance: Object', 'cost: Object', 'address: Object', 'id: Object', and 'identity: Object'. The 'validOrigins' object has a description: 'List of valid origin nodes for messages arriving via the remote node, used for duplicate elimination in redundant networks.' The 'description' field for the 'router.node' object is highlighted in green and reads: 'Remote router node connected to this router.'

```
console: Object
  router.config.address: Object
  router.node: Object
    operations: Array [1]
    attributes: Object
      routerLink: Object
      nextHop: Object
      name: Object
      validOrigins: Object {type:"list", description:"List of valid origin nodes for messages arriving via the remote node, used for duplicate elimination in redundant networks."}
      linkState: Object
      instance: Object
      cost: Object
      address: Object
      id: Object
      identity: Object
    description: "Remote router node connected to this router."
  dummy: Object
  router.address: Object
  router.link: Object
  org.amqp.management: Object
  operationalEntity: Object
  connection: Object
  router.config.linkRoute: Object
  router: Object
  annotations: Object
    sslProfile: Object
    addrPort: Object
    connectionRole: Object
```

This page displays the json schema that is used to manage the router network. :leveloffset: 0