Qpid Dispatch Router Book

Contents

1 Introduction

1.1 Overview

The Dispatch router is an AMQP message 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 reciever back across the same path. This means that *responsibility* for the message is transfered 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 immediatly 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 exampe 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 difficult 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
- 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

- Supports arbitrary topology - no restrictions on redundancy
- Automatic route computation - adjusts quickly to changes in topology
- Cost-based route computation
- Rich addressing semantics
- Security

2 Using Qpid Dispatch

2.1 Configuration

The default configuration file is installed in *install-prefix*/etc/qpid/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: o $ qdrouterd [-c my-config-file]

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

2.2 Client Compatibility

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

Client	*Notes*
qpid::messaging	The Qpid messaging clients work with Dispatch Router as long as they are configured to
Proton Messenger	Messenger works with Dispatch Router.

2.3 Tools

2.3.1 qdstat

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

Option	*Description*
-l	Print a list of AMQP links attached to the router. Links are unidirectional. Outgoing links are usua
-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.

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

2.3.2 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 route r $ qdrouterd query –type connection

To enable logging debug and higher level messages by default: $ qdrouter 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: $ qdrouter create type=listener port=5555

Now you can connect to port 5555, for exampple $ qdrouterd 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.

2.4 Features and Examples

2.4.1 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  address      in-proc  local  remote  in  out  thru  to-proc  from-proc
  ================================================================================
  local  $management  Y        0      0       1   0    0     1        0
  local  temp.AY81ga           1      0       0   0    0     0        0
```

Note that there are two known addresses. *$management* is the address of the router's embedded management agent. *temp.AY81ga* 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 two additional addresses which are used for inter-router communication:

```
$ qdstat -a
Router Addresses
  class  address      in-proc  local  remote  in  out  thru  to-proc  from-proc
  ================================================================================
  local  $management  Y        0      0       1   0    0     1        0
  local  qdhello      Y        0      0       0   0    0     0        3
  local  qdrouter     Y        0      0       0   0    0     0        1
  local  temp.khOpGb           1      0       0   0    0     0        0
```

2.4.2 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 *recv.py* example receiver:

```
$ recv.py amqp://0.0.0.0/my-address
```

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

```
$ qdstat -a
Router Addresses
  class   address      in-proc  local  remote  in  out  thru  to-proc  from-proc
  ================================================================================
  local   $management  Y        0      0       1   0    0     1        0
  mobile  my-address           1      0       0   0    0     0        0
  local   temp.fDt8_a          1      0       0   0    0     0        0
```

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

```
$ send.py -a amqp://0.0.0.0/my-address
```

2.4.3 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/$1" % 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 = {u'operation' : u'DISCOVER-MGMT-NODES',
                          u'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()
```

2.5 Known Issues and Limitations

This is an early test release. It is expected that users will find bugs and other various instabilities. The main goal of this release is to prove that the process can be run and that users can demonstrate basic functionality as described in this document. Nevertheless, the following are known issues with the 0.1 release:

- Subscriber addresses are not always cleaned up after a consumer disconnects. See https://issues.apache.org/jira/browse/QPID-4964.
- Dispatch Router does not currently use the target address of a client's sender link to route messages. It only looks at the "to" field in the message's headers. See https://issues.apache.org/jira/browse/QPID-5175.
- All subscription sources are treated as multicast addresses. There is currently no facility for provisioning different types of addresses. Multicast means that if there are multiple subscribers to the same address, they will all receive a copy of each message sent to that address.
- SSL connectors and listeners are supported but very lightly (and not recently) tested.
- SASL authentication is not currently integrated into any authentication framework. Use ANONYMOUS for testing.

3 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
- *Routing mechanism* - message routed, link routed
- *Undeliverable action* - drop, hold and retry, redirect
- *Reliability* - N destinations, etc.

3.1 Routing patterns

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

Pattern	*Description*
Direct	Direct routing allows for only one consumer to use an address at a time. Messages (or links) follo
Multicast	Multicast routing allows multiple consumers to use the same address at the same time. Messages
Balanced	Balanced routing also allows multiple consumers to use the same address. In this case, messages (

3.2 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. Some of the links are designated for message routing, and others are designated for link routing. In both cases, 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.

3.2.1 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 message-routing 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.

Delivery	*Handling*
pre-settled	If the arriving delivery is pre-settled (i.e., fire and forget), the incoming delivery shall be settled
unsettled	Unsettled delivery is also propagated across the network. Because unsettled delivery records can

3.2.2 Link routing

Link routing occurs when a new link is attached to the router across one of its AMQP connections. It is done based on the `target.address` field of an inbound link and the `source.address` field of an outbound link.

Link routing uses the same routing table that message routing uses. The difference is that the routing occurs during the link-attach operation, and link attaches are propagated along the appropriate path to the destination. What results is a chain of links, connected end-to-end, from source to destination. It is similar to a *virtual circuit* in a telecom system.

Each router in the chain holds pairs of link termini that are tied together. The router then simply exchanges all deliveries, delivery state changes, and link state changes between the two termini.

The endpoints that use the link chain do not see any difference in behavior between a link chain and a single point-to-point link. All of the features available in the link protocol (flow control, transactional delivery, etc.) are available over a routed link-chain.

4 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.

4.1 Message Annotations

The following Message Annotation fields are defined by Dispatch Router:

Field	*Type*	*Description*
x-opt-qd.ingress	string	The identity of the ingress router for a message-routed message. The ingress
x-opt-qd.trace	list of string	The list of routers through which this message-routed message has transited
x-opt-qd.to	string	To-Override for message-routed messages. If this field is present, the address
x-opt-qd.class	string	Message class. This is used to allow the router to provide separate paths for

4.2 Source/Target Capabilities

The following Capability values are used in Sources and Targets.

Capability	*Description*
qd.router	This capability is added to sources and targets that are used for inter-router message exchange.

4.3 Addresses and Address Formats

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

4.3.1 Address Patterns

Pattern	*Description*
`_local/<addr>`	An address that references a locally attached endpoint. Messages using th
`_topo/<area>/<router>/<addr>`	An address that references an endpoint attached to a specific router node
`<addr>`	A mobile address. An address of this format represents an endpoint or a

4.3.2 Supported Addresses

Address	*Description*
`_local/$management`	The management agent on the attached router/container. This address wou

Address	*Description*
`_topo/0/Router.E/agent`	The management agent at Router.E in area 0. This address would be used
`_local/qdhello`	The router entity in each of the connected routers. This address is used to
`_local/qdrouter`	The router entity in each of the connected routers. This address is used by
`_topo/0/Router.E/qdxrouter`	The router entity at the specifically indicated router. This address form is u

4.4 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 operation is not implementd. The router has its own mechansm to discover peers that does not require this operation.
- The DEREGISTER operation is not implementd. The router has its own mechansm to discover peers that does not require this operation.

5 The qdrouter management schema

The schema `qdrouterd.json` is a JSON format file that defines annotations and entity types of the Qpid Dispatch Router management model. The model is based on the AMQP management specification.

The schema is a JSON map with the following keys:

- "description": documentation string for the schema
- "prefix": Prefix prepended to schema names when they are exposed to AMQP management clients.
- "annotations": map of annotation names to definitions (see below)
- "entityTypes": map of entity type names to definitions (see below)

Annotation and entity type definition maps have the following keys:

- "description": documentation string.

- "operations": list of allowed operation names.
- "attributes": map of attribute names to attribute definitions (see below)

Entity type definitions also have these fields:

- "extends": Name of base type. The new type includes operations and attributes from the base type.
- "annotations": List of annotation names. The new type includes operations and attributes from all the annotations.

Attribute definition maps have the following fields:

- "type": one of the following:
- "String": a unicode string value.
- "Integer": an integer value.
- "Boolean": a true/false value.
-
- "default": a default can be a literal value or a reference to another attribute in the form `$attributeName`.

There is the following hierarchy among entity types:

`entity`: The base of all entity types. - `configurationEntity`: base for all types that hold *configuration information*.

Configuration information is supplied in advance and expresse *intent*. For example "I want the router to listen on port N". All the entities that can be used in the configuration file extend `configurationEntity`.

- `operationalEntity`: base for all types that hold *operational information*.

Operational information reflects the actual current state of the router. For example, "how many addresses are presently active?" All the entities queried by the `qdstat` tool extend `operationalEntity`.

The two types are often related. For example `listener` and `connector` extend `configurationEntity`, they express the intent to make or receive connections. `connection` extends `operationalEntity`, it holds information the actual connection status.

The rest of this section provides the schema documentation in readable format.

5.1 Annotations

5.1.1 `addrPort`

Attributes for internet address and port.

Used by `listener`, `connector`.

`addr` (String, default=0.0.0.0) Host address: ipv4 or ipv6 literal or a host name.

`port` (String, default=amqp) Port number or symbolic service name.

5.1.2 `saslMechanisms`

Attribute for a list of SASL mechanisms.

Used by `listener`, `connector`.

`saslMechanisms` (String, required) Comma separated list of accepted SASL authentication mechanisms.

5.1.3 `connectionRole`

Attribute for the role of a connection.

Used by `listener`, `connector`.

`role` (One of [normal, inter-router, on-demand], default=normal) 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 interRouter connections.

5.1.4 `sslProfile`

Attributes for setting TLS/SSL configuration for connections.

Used by `listener`, `connector`.

`certDb` (String) The path to the database that contains the public certificates of trusted certificate authorities (CAs).

`certFile` (String) The path to the file containing the PEM-formatted public certificate to be used on the local end of any connections using this profile.

`keyFile` **(String)** The path to the file containing the PEM-formatted private key for the above certificate.

`passwordFile` **(String)** If the above private key is password protected, this is the path to a file containing the password that unlocks the certificate key.

`password` **(String)** 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.

5.2 Base Entity Type

5.2.1 `entity`

Base entity type for all entities.

`name` **(String, unique)** Unique name, can be changed.

`identity` **(String, unique)** Unique identity, will not change.

`type` **(String, required)** Management entity type.

5.3 Configuration Entities

5.3.1 `configurationEntity`

Base type for entities containing configuration information.

Extends: `entity` (`name`, `identity`, `type`).

5.3.2 `container`

Attributes related to the AMQP container.

Extends: `configurationEntity` (`name`, `identity`, `type`).

`containerName` **(String)** 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=1)** The number of threads that will be created to process message traffic and other application work (timers, non-amqp file descriptors, etc.) .

`debugDump` **(String)** A file to dump debugging information that can't be logged normally.

5.3.3 `router`

Tracks peer routers and computes routes to destinations.

Extends: `configurationEntity` (`name`, `identity`, `type`).

`routerId` **(String)** Router's unique identity.

`mode` **(One of [standalone, interior, edge, endpoint], default=standalone)** In standalone mode, the router operates as a single component. It does not participate in the routing protocol and therefore will not coorperate with other routers. In interior mode, the router operates in cooreration with other interior routers in an interconnected network. In edge mode, the router operates with an uplink into an interior router network. Edge routers are typically used as connection concentrators or as security firewalls for access into the interior network.

`area` **(String)** Unused placeholder.

`helloInterval` **(Integer, default=1)** Interval in seconds between HELLO messages sent to neighbor routers.

`helloMaxAge` **(Integer, default=3)** Time in seconds after which a neighbor is declared lost if no HELLO is received.

`raInterval` **(Integer, default=30)** Interval in seconds between Router-Advertisements sent to all routers.

`remoteLsMaxAge` **(Integer, default=60)** Time in seconds after which link state is declared stale if no RA is received.

`mobileAddrMaxAge` **(Integer, default=60)** Time in seconds after which mobile addresses are declared stale if no RA is received.

`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.

5.3.4 `listener`

Listens for incoming connections to the router.

Extends: `configurationEntity` (`name`, `identity`, `type`).

Annotations: `sslProfile` (`certDb`, `certFile`, `keyFile`, `passwordFile`, `password`), `addrPort` (`addr`, `port`), `saslMechanisms` (`saslMechanisms`), `connectionRole` (`role`).

`requirePeerAuth` **(Boolean, default=True)** Only for listeners using SSL. If set to 'yes', attached clients will be required to supply a certificate. If the certificate is not traceable to a CA in the ssl profile's cert-db, authentication fails for the connection.

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

`allowUnsecured` **(Boolean)** For listeners using SSL only. If set to 'yes', this option causes the listener to watch the initial network traffic to determine if the client is using SSL or is running in-the-clear. The listener will enable SSL only if the client uis using SSL.

`allowNoSasl` **(Boolean)** If set to 'yes', this option causes the listener to allow clients to connect even if they skip the SASL authentication protocol.

`maxFrameSize` **(Integer, default=65536)** Defaults to 65536. 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 uniterruptible data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

5.3.5 `connector`

Establishes an outgoing connections from the router.

Extends: `configurationEntity` (`name`, `identity`, `type`).

Annotations: `sslProfile` (`certDb`, `certFile`, `keyFile`, `passwordFile`, `password`), `addrPort` (`addr`, `port`), `saslMechanisms` (`saslMechanisms`), `connectionRole` (`role`).

`allowRedirect` **(Boolean, default=True)** Allow the peer to redirect this connection to another address.

`maxFrameSize` **(Integer, default=65536)** 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 uniterruptible data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

5.3.6 `log`

Configure logging for a particular module.

Extends: `configurationEntity` (`name`, `identity`, `type`).

`module` **(One of [ROUTER, MESSAGE, SERVER, AGENT, CONTAINER, CONFIG, ERROR,** Module to configure. The special module 'DEFAULT' specifies defaults for all modules.

`enable` **(String, required, default=default)** 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)** Include timestamp in log messages.

`source` **(Boolean)** Include source file and line number in log messages.

`output` **(String)** Where to send log messages. Can be 'stderr', 'syslog' or a file name.

5.3.7 `fixedAddress`

Establishes semantics for addresses starting with a prefix.

Extends: `configurationEntity` (`name`, `identity`, `type`).

`prefix` **(String, required)** The address prefix (always starting with '/').

`phase` **(Integer)** The phase of a multi-hop address passing through one or more waypoints.

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

5.3.8 `waypoint`

A remote node that messages for an address pass through.

Extends: `configurationEntity` (`name`, `identity`, `type`).

`address` **(String, required)** The AMQP address of the waypoint.

`connector` **(String, required)** The name of the on-demand connector used to reach the waypoint's container.

`inPhase` **(Integer, default=-1)** The phase of the address as it is routed *to* the waypoint.

`outPhase` **(Integer, default=-1)** The phase of the address as it is routed *from* the waypoint.

5.4 Operational Entities

5.4.1 `operationalEntity`

Base type for entities containing current operational information.

Extends: `entity` (`name`, `identity`, `type`).

5.4.2 `router.link`

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

Extends: `operationalEntity` (`name`, `identity`, `type`).

`linkName` (String)

`linkType` (One of [endpoint, waypoint, inter-router, inter-area])

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

`owningAddr` (String)

`eventFifoDepth` (Integer)

`msgFifoDepth` (Integer)

`remoteContainer` (String)

5.4.3 `router.address`

AMQP address managed by the router.

Extends: `operationalEntity` (`name`, `identity`, `type`).

`inProcess` (Boolean)

`subscriberCount` (Integer)

`remoteCount` (Integer)

`deliveriesIngress` (Integer)

`deliveriesEgress` (Integer)

`deliveriesTransit` (Integer)

`deliveriesToContainer` (Integer)

`deliveriesFromContainer` (Integer)

5.4.4 `router.node`

AMQP node managed by the router.

Extends: `operationalEntity` (`name`, `identity`, `type`).

`addr` (String)

`nextHop` (Integer)

`routerLink` (Integer)

`validOrigins` (List)

5.4.5 `connection`

Connections to the router's container.

Extends: `operationalEntity` (`name`, `identity`, `type`).

`container` (String)

`state` (One of [connecting, opening, operational, failed, user])

`host` (String)

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

`role` (String)

`sasl` (String)

5.4.6 `allocator`

Memory allocation pool.

Extends: `operationalEntity` (`name`, `identity`, `type`).

`typeSize` (Integer)

`transferBatchSize` (Integer)

`localFreeListMax` (Integer)

`globalFreeListMax` (Integer)

`totalAllocFromHeap` (Integer)

`totalFreeToHeap` (Integer)

`heldByThreads` (Integer)

`batchesRebalancedToThreads` (Integer)

`batchesRebalancedToGlobal` (Integer)

6 Manual page qdrouterd.8

6.1 Name

qdrouterd - AMQP message router.

6.2 Synopsis

qdrouterd [*options*]

6.3 Description

The Qpid Dispatch router (qdrouterd) is a network daemon that directs AMQP 1.0 messages between endpoints, such as messaging clients and servers. ## Options

-c, –config=PATH (/etc/qpid-dispatch/qdrouterd.conf) Load configuration from file at PATH

-I, –include=PATH (/usr/lib/qpid-dispatch/python) Location of Dispatch's Python library

-d, –daemon Run process as a SysV-style daemon

-P, –pidfile If daemon, the file for the stored daemon pid

-U, –user If daemon, the username to run as

-h, –help Print this help

Run `qdrouterd --help` to see options.

6.4 Files

/etc/qpid-dispatch/qdrouterd.conf Configuration flie.

6.5 See Also

qdrouterd.conf(5), *qdstat*(8), *qdstat.conf*(5),

http://qpid.apache.org/components/dispatch-router

7 Manual page qdstat.8

7.1 Name

qdstat - A tool to inspect Dispatch router networks

7.2 Synopsis

qdrouterd [*options*]

7.3 Description

qdstat shows status information about networks of Dispatch routers. It can display connections, network nodes and links, and router stats such as memory use. ## Options

-h, –**help** show this help message and exit

–**version** Print version and exit.

-g, –**general** Show General Router Stats

-c, –**connections** Show Connections

-l, –**links** Show Router Links

-n, –**nodes** Show Router Nodes

-a, –**address** Show Router Addresses

-m, –**memory** Show Broker Memory Stats

7.3.1 Connection Options

-b URL, –**bus=URL** URL of the messaging bus to connect to (default

-r ROUTER-ID, –**router=ROUTER-ID** Router to be queried

-t SECS, –timeout=SECS Maximum time to wait for connection in seconds (default 5)

–sasl-mechanism=MECH Force SASL mechanism (e.g. EXTERNAL, ANONYMOUS, PLAIN, CRAM-MD5, DIGEST-MD5, GSSAPI).

–ssl-certificate=CERT Client SSL certificate (PEM Format)

–ssl-key=KEY Client SSL private key (PEM Format)

Run `qdstat --help` to see options.

7.4 See Also

qdrouterd (8), *qdmanage* (8), *qdrouterd.conf* (5)

http://qpid.apache.org/components/dispatch-router

8 Manual page qdmanage.8

8.1 Name

qdmanage - A management tool for Dispatch routers.

8.2 Synopsis

qdmanage *operation* [*options...*] [*arguments...*]

8.3 Description

An AMQP management client for use with qdrouterd. Sends AMQP management operations requests and prints the response in JSON format. This is a generic AMQP management tool and can be used with any standard AMQP managed endpoint, not just with qdrouter.

8.4 Operations

`query` [***ATTR...***] Prints the named attributes of all entities. With no arguments prints all attributes. The –type option restricts the result to entities extending the type.

`create` [***ATTR=VALUE...***] Create a new entity with the specified attributes. With the –stdin option, read attributes from stdin. This can be a JSON map of attributes to create a single entity, or a JSON list of maps to create multiple entitiees.

`read` Print the attributes of an entity specified by the –name or –identity options. With the –stdin option, create entities based on data from stdin. This can be a JSON map of attributes to create a single entity, or a JSON list of maps to create multiple entitiees.

`update` [***ATTR=VALUE...***] Update the attributes of an existing entity. With the –stdin option, read attributes from stdin. This can be a JSON map of attributes to update a single entity, or a JSON list of maps to update multiple entitiees. If an ATTR name is listed with no =VALUE, that attribute will be deleted from the entity.

`delete` Delete an entity specified by the –name or –identity options.

`get-types` [***TYPE***] List entity types with their base types. With no arguments list all types.

`get-operations` [***TYPE***] List entity types with their operations. With no arguments list all types.

`get-attributes` [***TYPE***] List entity types with their attributes. With no arguments list all types.

`get-annotations` [***TYPE***] List entity types with their annotations. With no arguments list all types.

`get-mgmt-nodes` List all other known management nodes connected to this one.

8.5 Options

-h, –help show this help message and exit

–version Print version and exit.

–type=TYPE Type of entity to operate on.

–name=NAME Name of entity to operate on.

–identity=ID Identity of entity to operate on.

–indent=INDENT Pretty-printing indent. -1 means don't pretty-print

–stdin Read attributes as JSON map or list of maps from

8.5.1 Connection Options

-b URL, –bus=URL URL of the messaging bus to connect to (default

-r ROUTER-ID, –router=ROUTER-ID Router to be queried

-t SECS, –timeout=SECS Maximum time to wait for connection in seconds (default 5)

–sasl-mechanism=MECH Force SASL mechanism (e.g. EXTERNAL, ANONYMOUS, PLAIN, CRAM-MD5, DIGEST-MD5, GSSAPI).

–ssl-certificate=CERT Client SSL certificate (PEM Format)

–ssl-key=KEY Client SSL private key (PEM Format)

Run `qdmanage --help` to see options.

8.6 Files

/usr//usr/share/doc/qpid-dispatch/qdrouter.json Management schema for qdrouterd.

/usr//usr/share/doc/qpid-dispatch/qdrouter.json.readme.txt Explanation of the management schema.

8.7 Examples

Show the logging configuration

```
qdmanage query --type=log
```

Enable debug and higher log messages by default:

```
qdmanage udpdate name=log/DEFAULT enable=debug+
```

Enable trace log messages only for the MESSAGE moudle and direct MESSAGE logs to the file “test.log”

```
qdmanage udpdate name=log/MESSAGE enable=trace output=test.log
```

Set MESSAGE logging back to the default:

```
qdmanage udpdate name=log/MESSAGE enable=default
```

Disable MESSAGE logging:

```
qdmanage udpdate name=log/MESSAGE enable=none
```

8.8 See Also

qdrouterd(8), *qdstat*(8), *qdrouterd.conf*(5)

http://qpid.apache.org/components/dispatch-router

9 Manual page qdrouterd.conf.5

9.1 Name

qdrouterd.conf - Configuration file for the Qpid Dispatch router

9.2 Description

The configuration file is made up of sections with this syntax:

```
SECTION-NAME {
    ATTRIBUTE-NAME: ATTRIBUTE-VALUE
    ATTRIBUTE-NAME: ATTRIBUTE-VALUE
    ...
}
```

There are two types of sections:

Configuration sections correspond to configuration entities. They can be queried and configured via management tools as well as via the configuration file.

Annotation sections define a group of attribute values that can be included in one or more entity sections.

For example you can define an "ssl-profile" annotation section with SSL credentials that can be included in multiple "listener" entities. Here's an example, note how the 'ssl-profile' attribute of 'listener' sections references the 'name' attribute of 'ssl-profile' sections.

```
ssl-profile {
    name: ssl-profile-one
    cert-db: ca-certificate-1.pem
    cert-file: server-certificate-1.pem
    key-file: server-private-key.pem
}

listener {
    ssl-profile: ssl-profile-one
```

```
    addr: 0.0.0.0
    port: 20102
    sasl-mechanisms: ANONYMOUS
}
```

9.3 Annotation Sections

9.3.1 Addrport

Attributes for internet address and port.

addr (String, default=0.0.0.0) Host address: ipv4 or ipv6 literal or a host name.

port (String, default=amqp) Port number or symbolic service name.

Used by listener, connector.

9.3.2 Saslmechanisms

Attribute for a list of SASL mechanisms.

saslMechanisms (String, required) Comma separated list of accepted SASL authentication mechanisms.

Used by listener, connector.

9.3.3 Connectionrole

Attribute for the role of a connection.

role (One of [normal, inter-router, on-demand], default=normal) 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 interRouter connections.

Used by listener, connector.

9.3.4 Sslprofile

Attributes for setting TLS/SSL configuration for connections.

certDb (String) The path to the database that contains the public certificates of trusted certificate authorities (CAs).

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

keyFile (String) The path to the file containing the PEM-formatted private key for the above certificate.

passwordFile (String) If the above private key is password protected, this is the path to a file containing the password that unlocks the certificate key.

password (String) 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.

Used by listener, connector.

9.4 Configuration Sections

9.4.1 Container

Attributes related to the AMQP container.

containerName (String) 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=1) The number of threads that will be created to process message traffic and other application work (timers, non-amqp file descriptors, etc.) .

debugDump (String) A file to dump debugging information that can't be logged normally.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

9.4.2 Router

Tracks peer routers and computes routes to destinations.

routerId (String) Router's unique identity.

mode (One of [standalone, interior, edge, endpoint], default=standalone) In standalone mode, the router operates as a single component. It does not participate in the routing protocol and therefore will not coorperate with other routers. In interior mode, the router operates in cooreration with other interior routers in an interconnected network. In edge mode, the router operates with an uplink into an interior router network. Edge routers are typically used as connection concentrators or as security firewalls for access into the interior network.

area (String) Unused placeholder.

helloInterval (Integer, default=1) Interval in seconds between HELLO messages sent to neighbor routers.

helloMaxAge (Integer, default=3) Time in seconds after which a neighbor is declared lost if no HELLO is received.

raInterval (Integer, default=30) Interval in seconds between Router-Advertisements sent to all routers.

remoteLsMaxAge (Integer, default=60) Time in seconds after which link state is declared stale if no RA is received.

mobileAddrMaxAge (Integer, default=60) Time in seconds after which mobile addresses are declared stale if no RA is received.

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.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

9.4.3 Listener

Listens for incoming connections to the router.

requirePeerAuth (Boolean, default=True) Only for listeners using SSL. If set to 'yes', attached clients will be required to supply a certificate. If the certificate is not traceable to a CA in the ssl profile's cert-db, authentication fails for the connection.

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

allowUnsecured (Boolean) For listeners using SSL only. If set to 'yes', this option causes the listener to watch the initial network traffic to determine if the client is using SSL or is running in-the-clear. The listener will enable SSL only if the client uis using SSL.

allowNoSasl (Boolean) If set to 'yes', this option causes the listener to allow clients to connect even if they skip the SASL authentication protocol.

maxFrameSize (Integer, default=65536) Defaults to 65536. 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 uniterruptible data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

Annotations sslProfile, addrPort, saslMechanisms, connectionRole.

9.4.4 Connector

Establishes an outgoing connections from the router.

allowRedirect (Boolean, default=True) Allow the peer to redirect this connection to another address.

maxFrameSize (Integer, default=65536) 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 uniterruptible data that can be sent for a message delivery over the connection. Interleaving of messages on different links is done at frame granularity.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

Annotations sslProfile, addrPort, saslMechanisms, connectionRole.

9.4.5 Log

Configure logging for a particular module.

module (One of [ROUTER, MESSAGE, SERVER, AGENT, CONTAINER, CONFIG, ERROR Module to configure. The special module 'DEFAULT' specifies defaults for all modules.

enable (String, required, default=default) 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) Include timestamp in log messages.

source (Boolean) Include source file and line number in log messages.

output (String) Where to send log messages. Can be 'stderr', 'syslog' or a file name.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

9.4.6 Fixedaddress

Establishes semantics for addresses starting with a prefix.

prefix (String, required) The address prefix (always starting with '/').

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

fanout (One of [multiple, single], default=multiple) 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) 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.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.

9.4.7 Waypoint

A remote node that messages for an address pass through.

address (String, required) The AMQP address of the waypoint.

connector (String, required) The name of the on-demand connector used to reach the waypoint's container.

inPhase (Integer, default=-1) The phase of the address as it is routed *to* the waypoint.

outPhase (Integer, default=-1) The phase of the address as it is routed *from* the waypoint.

name (String, unique) Unique name, can be changed.

identity (String, unique) Unique identity, will not change.