OOI-CI Prototype Exchange-Space API

Tony Garnock-Jones <tonyg@lshift.net>

28th December 2009

Abstract

Description of the prototype OOI-CI AMQP API for managing and making use of exchange-spaces and exchange-points alongside existing AMQP 0-9-1 resources, and relation of this API to other ongoing work.

1 Introduction

A fully general AMQP model would not only cover AMQP 1.0 and 0-9-1, but would also be an instance of a NIPCA-style DIF architecture, and would permit uniform naming and use of exchanges, resources, gateways and so forth. While we're working on such a generalisation of the AMQP model, an interim API will be useful not only for a first generation of prototype applications but also for clarifying our thinking around enrolment, DIF architecture, and generalised AMQP.

This document discusses the programming and wire-protocol interfaces to the new exchange-space and exchange-point functionality, but does not discuss details of potential implementations of the new interfaces. It also touches on the relationship of this prototyping work to the generalisation of AMQP and to the ongoing NIPCA development work.

2 Behavioural Definitions

The new entities have particular behaviours that need to be defined independent of the protocols used to manipulate them.

2.1 Exchange Space

Exchange spaces are "virtual" or "cloud" analogues of AMQP 0-9-1's virtual-hosts. Some aspects of "real" AMQP virtual-hosts are not available in exchange-spaces.[1] An exchange space logically

- contains exchange-points, and
- mediates access to the contained exchange-points.

[1] Notably, queues are not available within an exchange-space.

In effect, an exchange-space is an administrative scope for the exchange-point resources it contains. The lifecycle of the exchange-space bounds the lifecycle of the contained exchange-points: when the space is deleted, so are its contained points.

2.2 Exchange Point

Exchange points are "virtual" or "cloud" analogues of AMQP 0-9-1's exchanges.[2] An exchange point

- can be of any AMQP 0-9-1 type
- can be published to
- can have queues bound to it

Exchange points, just like regular AMQP exchanges, have lifetimes that are independent of the lifetimes of the client connections that created them. In AMQP, an exchange that is declared by a connected client persists until either it is explicitly deleted, the containing broker is restarted (in the case of non-durable exchanges), or the containing broker's configuration database or the containing broker itself is deleted (in the case of durable exchanges). Exchange-points behave similarly: they persist until explicitly deleted, or until the containing exchange-space is reset or destroyed.

3 API overview

The existing AMQP 0-9-1 wire protocol is used unchanged. Exchange-spaces and exchange-points within them are encoded as new exchange types in the system.

3.1 Declaring exchange spaces

Exchange spaces are declared[3] by invoking `Exchange.Declare` with an exchange type

[2] In this prototype implementation, exchange-points are also *modelled* as exchanges.

[3] "Declaration" in the AMQP sense of asserting the existence of, creating if necessary.

of `x-exchangespace`.[4] The name given to `Exchange.Declare` will be used as the name of the exchange space. The `arguments` table should contain a string-valued entry named `implementation`, which is used to select an implementation variant.[5] For the prototype, the string should have the value `prototype`. Other entries in the `arguments` table are exchange-space-implementation-specific parameters; for the prototype implementation, no parameters are needed.

3.2 Securing exchange spaces

The prototype implementation will have the ability to manage access to created exchange-spaces by using RabbitMQ's existing ACL-like facilities for restricting access to exchanges (as specified in the Statement of Work). While we do have some nascent ideas on controlling authorisation in a distributed setting, development of such ideas is out-of-scope for this phase of development.

The broker holding the exchange declared as type `x-exchangespace` should have users and permissions configured appropriately for restricting access to the declared exchange. Brokers wishing to join an exchange-space must be able to log into the broker on which the `x-exchangespace` exchange is declared, and must be able to both read from and write to the exchange.

3.3 Joining and leaving exchange spaces

Joining (i.e., becoming a participating peer in) an exchange space is done implicitly as part of the declaration of a particular exchange point. When an exchange-point is declared, the exchange-space management function is contacted by a behind-the-scenes process. No explicit client action besides declaration of the exchange-point is required.

Leaving an exchange space is done implicitly as part of exchange-point deletion; this is a point of awkwardness in the current design because while there are explicit representations of exchange-spaces and exchange-points, there are no representations of the behind-the-scenes relay processes. Not having a direct handle on relay processes means that all changes to their configuration happen as side-effects of actions on other objects.[6]

[4]Non-standard exchange types are required by the AMQP 0-9-1 specification to have names starting with `x-`.

[5]This is an instance of the *strategy pattern.*

[6]Two possible approaches to avoiding these problems occur to me: either reify the relay processes themselves, or move joining and leaving to being actions that occur on bind and unbind. Neither approach is entirely satisfactory. For the prototype phase of work, we can leave the situation unresolved and concentrate on the main aspects of the system.

3.4 Declaring exchange points

Exchange points are declared by invoking `Exchange.Declare` with an exchange type of `x-exchangepoint`. The name given to `Exchange.Declare` will be used as a *local* alias for the exchange point, and does not need to correspond to the name of the exchange point itself. An exchange-point then, from the declaring client's (and the client's local broker's!) point-of-view, has *two* names: the local alias, for use only with the local broker, as well as its "true name", the combination of its exchange-space's name and the exchange-point's unique name within the space.

The `arguments` table given to `Exchange.Declare` must contain the following entries:

- a table-valued entry named `exchangespace`, with contents specific to the implementation variant used but containing at least a string-valued entry named `implementation`, which is used to select an implementation variant. For the prototype, the string should have the value `prototype`.
- a string-valued entry named `exchangename` containing an AMQP exchange name such as might be supplied in the name argument to `Exchange.Declare`. This is the name of the exchange-point itself, within the remote exchange-space.
- a string-valued entry named `exchangetype` containing an AMQP exchange type name such as might be supplied in the exchange type argument to `Exchange.Declare`.

The `arguments` table may also contain:

- a table-valued entry named `arguments` containing a nested set of exchange declaration arguments such as might be supplied in the arguments argument to `Exchange.Declare`. If this entry is omitted, an empty table is assumed.

Taken together, the `exchangename`, `exchangetype` and `arguments` fields in the `arguments` table given to `Exchange.Declare` for the exchange point are sufficient to specify the kind of exchange point being declared.

If the exchange-point declaration fails, including implementation-internal failures such as problems contacting or negotiating with the exchange-space management function, the usual AMQP error reporting procedures are used to notify the declaring client of the problem. All of AMQP's existing restrictions on exchange declarations—that the exchange types must match, and so on—apply to exchange points, as well; that is, if one client declares an exchange-point with type "fanout", and another declares one with the same name but with type “direct”, the second declaration will fail.

3.5 Deleting exchange points and exchange spaces

To delete an exchange point or exchange space, invoke `Exchange.Delete` as usual.

An awkwardness exists here when deleting an exchange point: in AMQP, exchanges can be deleted without first declaring them, but here to delete an exchange point the broker must first know which exchange space to delete the point from, so the point must first have been declared on the local broker.

dThis is a syntactic problem with the AMQP protocol as it stands. If this prototype design were to be taken to production, the awkwardness would need to be worked around by some out-of-band, non-AMQP resource management tool, perhaps agent- or GUI-based, that could manipulate structures internal to the prototype implementation without being restricted to the existing AMQP operations.

A fully generalised model of AMQP will not have this awkwardness: resource deletion will include enough information to precisely locate the resource to delete without having to construct it beforehand.

3.6 Binding to and unbinding from exchange points

A similar awkwardness exists here as described for deleting exchange points: they *must* be declared on the local broker before any kind of binding is attempted. Given this, however, binding to exchange points can be done as usual for AMQP: issue `Queue.Bind` and `Queue.Unbind`, mentioning the local alias for the exchange point that was set up in the `Exchange.Declare`.

3.7 Publishing to exchange points

Messages sent to locally-declared `x-exchangepoint` exchanges using `Basic.Publish` will be delivered to other participants in the exchange-space that have subscribed to the exchange-point, with implementation-specific QoS and reliability guarantees. See section 4.6 for details on QoS levels provided by the prototype implementation.

The `mandatory` and `immediate` flags to `Basic.Publish` are not supported in the prototype implementation, and must be set `false`.[7] A channel error will be signalled if either is set `true`. Similarly, `Tx`-class transactions are not supported in the prototype; if a `Basic.Publish` is issued to an `x-exchangepoint` exchange over a channel on which `Tx.Select` has been called, a channel error will be signalled.

[7]The reason for this is that it is not clear what they mean in a distributed setting.

4 Prototype implementation

The prototype implementation will provide a single exchange-space implementation variant, `prototype`. Coordination between participants in an exchange-space will be centralised (at the broker on which the exchange-space exchange was declared), analogous to the centralised tracker in early variants of Bittorrent, but the actual exchange of data between exchange-points will be decentralised, managed directly by the exchange-point participants.

This API is flexible enough to accommodate alternative implementations of the exchange-space management function, requiring changes only to the specific arguments passed in the `Exchange.Declare` command used to create an exchange-space. For example, if a decentralised exchange-space network management function were implemented, the parameters to `Exchange.Declare` would include “seed” addresses that would be used to bootstrap the network into existence. An agent-based implementation would provide specifics of the agents that were involved in the network management function.

4.1 Exchange-point declaration parameters

When declaring an exchange-point, the table-valued `exchangespace` field of the `arguments` table of the `Exchange.Declare` command must contain the following fields in addition to the mandatory `implementation` entry:

- `host`, a string: the hostname of the AMQP broker on which the exchange-space is declared.
- `space`, a string: the name given to `Exchange.Declare` when the *exchange-space* was declared.

It may also contain:

- `port`, an integer: the port number of the AMQP broker on which the exchange-space is declared. If omitted, 5672 is used.
- `username`, a string: the user name to use when connecting to the AMQP broker on which the exchange-space is declared. If omitted, `guest` is used.

- `password`, a string: the password to use when connecting to the AMQP broker on which the exchange-space is declared. Optional unless `username` is provided; if omitted, `guest` is used.

- `virtualhost`, a string: the virtual host to use when connecting to the AMQP broker on which the exchange-space is declared. If omitted, / is used.

The synchronisation technique and protocol used to implement exchange-points for the prototype has yet to be determined, though it is likely to be similar to the experiments embodied in `http://github.com/tonyg/pika/blob/master/examples/demo_relay.py`. (It's also worth noting that the particular technique chosen is not visible at the API.)

4.2 Potential implementation alternatives

This API definition makes use of the strategy pattern: alternative implementations following the defined interface can be selected by varying the `implementation` arguments given to `Exchange.Declare` commands.

In order to get a feel for the space of possible implementations, it's useful to consider an analogy between this system and the Bittorrent synchronisation system. In Bittorrent, each active torrent consists of exactly one tracker, zero or more peers, and exactly one finitely-bounded set of information to be synchronised (the dataset being distributed). One could say that the tracker is logically identified with the torrent.

In the exchange-point/exchange-space system, each exchange space consists of exactly one network management function, zero or more peers, and zero or more unbounded streams of information to be synchronised (the exchange points themselves). One could say that the network management function is logically identified with the exchange-space.

Multiple implementation variants for Bittorrent are possible: the tracker can be centralised or decentralised; peers can employ different transports and protocols for performing the synchronisation task ahead of them. Similarly, multiple implementation variants for exchange-spaces are possible: the network management function can be centralised or decentralised, and peers can coordinate in many different ways to synchronise their views of the information flowing through the exchange points contained within the exchange space. A decentralised network management function would be analogous to the Distributed Hash Table used to manage Bittorrent trackers in a decentralised way.

4.3 Network substrate independence

One of the purposes of this prototype is to integrate with the DIF implementation being developed in parallel with it. As it stands, our implementation of this API uses AMQP over TCP to communicate from peer to peer and from peer to exchange-space network manager. When the DIF implementation becomes available, all that will be required will be formalisation of the means of specifying the addresses involved: again, the strategy pattern instance inherent in the use of the `arguments` table in `Exchange.Declare` commands gives us a convenient location to place the required syntax.

4.4 Broker independence and AMQP 1.0

Both the design presented here and the concrete implementation being developed are, to some extent, broker-independent. While we do take advantage of some special features of RabbitMQ for ease and speed of development, it would be possible to operate in an entirely implementation-neutral fashion (at a cost in robustness, complexity and efficiency) by changing the design, reifying implementation structures such as exchange-spaces, exchange-points, and exchange-point relays, turning them into ordinary AMQP applications.

Mapping the design onto AMQP 1.0 has yet to be discussed: it is unclear whether it will involve the kind of alteration discussed in the previous paragraph, or whether a more natural embedding will be possible.

4.5 Relationship to generalised AMQP work

The generalisation of AMQP mentioned in the introduction is still being developed, in parallel with this prototyping work and the aforementioned work on a generalised network transport interface. This prototyping work will help clarify some issues in the generalisation work, particularly those surrounding enrolment and the roles of queues and of relays.

4.6 Confirmation of receipt

The prototype implementation will deliver on a best-effort basis. Failure of a network element[8] on the path between two peer brokers could lead to message loss. Therefore, application-level acknowledgements, timeouts, and retransmissions will be required in situations where application semantics require confirmation of delivery.

[8]For example: netsplits, routing misconfiguration, and other connectivity failures; firewall reboots; broker restarts; unexpected TCP resets; and so on.

For example, let us imagine a system where an exchange-point is used to relay job requests to some remote service. The service instance receiving the requests should send an acknowledgement of receipt to the originator of each request. If the jobs complete quickly enough, the acknowledgment and the reply (if there is one) can be bundled together in a single message; if the jobs are long-running, a separate acknowledgement message should be sent when a job is received from the broker, with a reply message following later once the job completes. Furthermore, on receipt of a duplicate request, a duplicate acknowledgement[9] should be sent (without actually running the job again). If the originator of a request does not receive an acknowledgement in a reasonable amount of time, it should retransmit the request. This way, robust confirmation of delivery can be achieved, and a wide variety of responsibility-transfer schemes can be straightforwardly implemented without depending on the fine implementation detail of the exchange-point system.

5 Example

In this example,

- an exchange-space, `market`, is created by user Margaret on her AMQP server at `amqp.market.example`;
- user Alice, who runs an AMQP server at `amqp.acme.example`, runs a program that wishes to subscribe to and publish to the exchange-points `bids` and `mailbox` within `market`;
- user Bob, who runs an AMQP server at `amqp.brandcorp.example`, runs a program that also wishes to subscribe to and publish to the exchange-points `bids` and `mailbox` within `market`;
- the exchange-point `bids` is a topic exchange;
- the exchange-point `mailbox` is a direct exchange;[10]
- we assume that all machines have prearranged direct TCP-level connectivity available.

The pseudocode snippets given below are non-normative: they are simply illustrative examples of possible commands that could be issued in a scenario such as the one we are examining in this section.

[9]And a duplicate reply, if a reply is available at the time.

[10]One interesting benefit of direct exchanges is that it is not possible to make a wildcard binding to the exchange: if two parties are communicating via a direct exchange with a secret routing key, then all other things being secured there is no means by which an eavesdropper can overhear their conversation other than guessing the routing key being used. (Interestingly, it doesn't look like this kind of 'security' is possible in AMQP 1.0.)

The exchange space is created

Margaret issues the following command to her server at `amqp.market.example`:

```
Exchange.Declare(exchange = 'market',
                 arguments =
                   {'implementation':
                       'prototype'},
                 type = 'x-exchangespace')
```

This command could fail for all the usual AMQP reasons: duplicate name with differing type, permission denied, unsupported exchange type, and so on.

Alice tries to join the space, but fails

Alice issues the following command to her server at `amqp.acme.example`:

```
Exchange.Declare(exchange = 'marketbids',
  type = 'x-exchangepoint',
  arguments =
    {'exchangespace':
       {'implementation': 'prototype',
        'host': 'amqp.market.example',
        'space': 'market',
        'username': 'alice',
        'password': '41iC3'},
     'exchangename': 'bids',
     'exchangetype': 'topic'})
```

Alice's server instantiates an exchange-space relay, behind the scenes, and causes it to try to contact the exchange-space denoted by the `exchangespace` argument to the exchange declaration. This contact fails, ultimately causing the command that Alice issued to fail, because Margaret's server has not been configured to permit a user named `alice` to log in and access the exchange-space embodied by the exchange called `market`.

The command could also have failed for all the other usual AMQP exchange-declaration-failure reasons, including in particular either a *local* exchange already existing with different exchange-type,[11] or the remote *exchange-point* already existing with different exchange-type.[12]

[11]That is, a local exchange called `marketbids` with exchange-type something other than `x-exchangepoint`.

[12]That is, a remote exchange-point called `bids` within the `market` space with exchange-type something other than `topic`.

Margaret configures permissions for Alice and Bob

Alice calls Margaret, who creates a RabbitMQ user[13] on `amqp.market.example` called `alice`, sets the password appropriately, and permits that user to write to and read from the exchange called `market`. Thinking ahead, Margaret does the same for `bob`, to prevent trouble later.

Alice joins the space

Alice reissues her `Exchange.Declare` on `amqp.acme.example`. This time, the behind-the-scenes relay successfully contacts `amqp.market.example`, joins the exchange space, and sets up the internal plumbing required to make the exchange called `marketbids` on `amqp.acme.example` an alias for the exchange-point `bids` in the exchange-space `market`. Alice issues a similar command, setting up a local alias `marketmail` for exchange-point `mailbox` in exchange-space `market`:

```
Exchange.Declare(exchange = 'marketmail',
  type = 'x-exchangepoint',
  arguments =
    {'exchangespace':
       {'implementation': 'prototype',
        'host': 'amqp.market.example',
        'space': 'market',
        'username': 'alice',
        'password': '41iC3'},
     'exchangename': 'mailbox',
     'exchangetype': 'direct'})
```

Alice creates queues and binds them to `marketbids` and `marketmail` as usual.

Bob joins the space

Bob issues commands to `amqp.brandcorp.example`, very similar to those Alice issued, with differences that let the exchange-point aliases being constructed fit smoothly in with Bob's existing application architecture.

```
Exchange.Declare(exchange = 'b',
  type = 'x-exchangepoint',
  arguments =
    {'exchangespace':
       {'implementation': 'prototype',
        'host': 'amqp.market.example',
        'space': 'market',
        'username': 'bob',
        'password': '808'},
     'exchangename': 'bids',
     'exchangetype': 'topic'})
```

[13]In this prototype, anyway. AMQP has no general notion of "user", instead delegating such issues to SASL, so here I'm referring instead to the authentication system currently implemented in RabbitMQ.

```
Exchange.Declare(exchange = 'm',
  type = 'x-exchangepoint',
  arguments =
    {'exchangespace':
       {'implementation': 'prototype',
        'host': 'amqp.market.example',
        'space': 'market',
        'username': 'bob',
        'password': '808'},
     'exchangename': 'mailbox',
     'exchangetype': 'direct'})
```

The network management function helps coordinate Alice's and Bob's local exchange-point aliases

The behind-the-scenes relays in each of Alice's and Bob's local AMQP servers have been coordinating with each other as a result of Alice's and Bob's exchange-space-manipulating commands (the `Exchange.Declare`s and the `Queue.Bind`s).

Alice and Bob communicate

Alice and Bob `Basic.Publish` messages to `marketbids/marketmail` and `b/m`, their respective local aliases for the exchange-points `bids` and `mailbox` in the `market` exchange-space. The behind-the-scenes relays arrange for messages to be delivered to appropriately bound queues.