Old Clustering Design Note

Overview

The following is a proposal for the design of a clustering solution to increase the scalability of the Qpid AMQP broker by allowing multiple broker processes to collaborate to provide services to application clients connected to any one of these processes. By spreading the connections across different processes more clients can be supported.

Terms & Definitions

A cluster consists of any number of brokers in a fixed order. Each broker in the cluster has a unique name. All brokers in the cluster know the set of brokers in their cluster and agree on the ordering of those brokers. The first broker in the cluster is the leader; all brokers agree on the current leader. The mechanisms for achieving and maintaining this structure will be described below.

Each client is connected to one broker in the cluster, to whom it sends its requests in the same way it would were it connected to a broker that was not part of a cluster. Clients that implement the AMQP specification should be able to work with a clustered broker unaltered. However, when in a cluster, a broker will need to alter its response to certain client requests. An objective of the design is to minimise the scope and complexity of this altered behaviour.

Brokers in a cluster all connect to each other. Though one socket between any two brokers would suffice, it is simpler to implement if we assume that each broker will be a client of each other broker. Thus there will in fact be two connections between any two members, one in each 'direction'. This way we can reuse as much of a non-clustered brokers behaviour as possible.

A broker will need to distinguish between sessions with an application client and sessions where the 'client' of the socket in a session is actually another broker.

Outline of Approach for Clustering

Stated simply, the cluster will:

- replicate exchanges by broadcasting the original Exchange.Declare and Exchange.Delete messages to all members of the cluster.
- replicate queues by broadcasting the original Queue.Declare and Queue.Delete messages to all members of the cluster
- replicate bindings by broadcasting Queue.Bind messages to all members of the cluster
- relay messages from a copy of an exchange in one broker to the equivalent exchange in another broker where necessary to ensure that consumers on any broker in the cluster receive messages that are published to any broker in the cluster

Private queues exist in real form in only one broker; the broker that receives the original declaration from a client. All the other brokers in the cluster set up a proxy for this queue. This proxy can be bound to exchanges as a normal queue can, but whenever a message is routed to it, that message is simply relayed on to the broker in which the real queue exists. However, though multiple queue proxies may exist with the same target member, if these are bound such that all of them match a given message, only one copy of that message should be relayed to the target member.

Copies of shared queues will exist at each broker in the cluster. These are bound to exchanges as usual and can have consumers registered with them. In addition to consumers from the application clients, these shared queue copies track the number of consumer for that queue that are held on other brokers. They use this information to fairly distribute messages between all consumers.

The clustering in general involves propagation of certain methods received by one broker in the cluster from a client to all the other cluster members. Specifically those methods concerned with the setup of routing information are propagated allowing all members of the cluster to play their part in the routing of messages from and to clients distributed across the cluster.

In particular the cluster will propagate all Exchange.Declare, Exchange.Delete, Queue.Declare, Queue.Delete and Queue.Bind messages. It will also propagate Basic.Consume and Basic.Cancel messages that refer to shared queues.

The propagation can be carried out synchronously or asynchronously with respect to the original client request. In other words the broker that receives one of these messages from a client will send an equivalent message to the other brokers and can then wait until it receives responses from these brokers before it sends the confirmation message back to the client. Alternatively it could return a response to the client immediately. A hybrid approach could also be used. In general the originating broker waits for n responses, where $0 < n <$ number of members in the cluster. The value of n to be used will be set through policies to achieve the required latency v. consistency trade offs for a particular situation.
Cluster Management

As mentioned above the cluster is defined to be an agreed set of member brokers in an agreed order. This helps reasoning about consistency. The 'first' member of the group acts as the leader and issues authoritative statements on who is in or out of the cluster. All brokers in the cluster store the last received membership announcement from which they can infer the current leader.

Ordering is maintained by requiring that new members join through the leader. A prospective new member can connect to any other member in the cluster, but these other members should pass on the join request to the leader.

Once connected to a member of the group the new member issues a join request, to which the leader responds by sending a new membership announcement to all members including the new member. It will also initiate the replay messages required to replicate cluster state to the new member; the other cluster members also participate in this message replay. Once it has processed all the replayed messages and is therefore up to date with respect to cluster state, the new member can start accepting client connections.

State is transferred through (a) Exchange.Declare methods for all exchanges, (b) Queue.Declare messages for all queues, (c) Queue.Bind requests for all queue bindings in all exchanges and (d) Basic.Consume requests for all consumers of shared queues at each node. The leader is responsible for replicating all exchanges, shared queues and their bindings. Other members are responsible for replicating private queues hosted by them and the bindings for these queues as well as consumer counts for shared queues. The replay of messages from the leader must be processed before those from other cluster members (as e.g. bindings for private queues require that the exchanges have already been declared). The completion of the required replay of messages from a broker is signaled by a Cluster.Synch message. Messages received after this are 'live' messages received through the receiving broker being treated as a normal member.

Failure of a broker may be detected by any other broker in the cluster in the course of trying to communicate with that broker. Failures are handled by sending a suspect message to the leader of the cluster, who verifies the suspected broker is down and issues a new announcement of membership, with the failed broker removed if the failure is verified. In addition to discovery of failure during normal communication, each broker member is responsible for periodically pinging the 'previous' broker (i.e. the broker that occurs just before itself in the ordered membership list). The leader will assume responsibility for pinging the last member to join the group.

The leader may itself fail. This may be detected by the next broker in the list, in which case that broker responds by assuming leadership and sending an announcement of the new membership list with the old leader removed. It may also be detected by other brokers. As they cannot send a suspect warning to the leader, they send it to the broker next to the leader.
Message Handling Changes and Protocol Extensions

To incorporate clustering while reusing the same communication channel for intra-cluster communications and extension to the protocol is proposed. It is not necessary for clients to know about this extension so it has no impact on the compliance of the broker and can be treated as a proprietary extension for Qpid. The extension consists of a new class of messages, Cluster, which has the following methods:

Cluster.Join

Sent by a new member to the leader of the cluster to initiate the joining process. On receiving a join the leader will try to establish its own connection back to the new member. It will then send a membership announcement and various messages to ensure the new member has the required state built up.

Cluster.Membership

Sent by the leader of the cluster whenever there is a change in the membership of the cluster either through a new broker joining or through a broker leaving or failing. All brokers should store the membership information sent. If they are waiting for responses from a member that is no longer part of the cluster they can handle the fact that that broker has failed. If it contains a member to whom they have not connected they can connect (or reconnect).

Cluster.Leave

Sent to the leader by a broker that is leaving the cluster in an orderly fashion. The leader responds by sending a new membership announcement.

Cluster.Suspect

Sent by brokers in the cluster to the leader of the cluster to inform the leader that they suspect another member has failed. The leader will attempt to verify the falure and then issue a new Cluster.Membership message excluding the suspected broker if it has failed leaving it in if it seems to be responding.

Cluster.Synch

Sent to complete a batch of message replayed to a new member to allow it to build up the correct state.

Cluster.Ping

Sent between brokers in a cluster to give or request a heart beat and to exchange information about loading. A ping has a flag that indicates whether it expects a response or not. On receiving a ping a broker updates its local view of the load on that server and if required sends its own ping in response.

In addition to this new class, the handling of the following is also altered. The handling of each message may depend on whether it is received from an application client or from another broker.

Connection.Open

A broker needs to detect whether the open request is from an application client or another broker in the cluster. It will use the capabilities field to do this; brokers acting as clients on other brokers require the 'cluster-peer' capability.

If a broker receives a Connection.Open from an application client (i.e. if the cluster-peer capability is not required) it may issue a Connection.Redirect if it feels its loading is greater than the loading of other members in the cluster.

Exchange.Declare

On receiving this message a broker propagates it to all other brokers in the cluster, possibly waiting for responses before responding with an Exchange. Declare-Ok.

Queue.Declare

On receiving this message a broker propagates it to all other brokers in the cluster, possibly waiting for responses before responding with a Queue.Declare-Ok.

Queue.Bind

Again, this is replicated to all other brokers, possibly waiting for responses before sending back a Queue.Bind-Ok to the client.

Queue.Delete

On receiving this message a broker propagates it to all other brokers in the cluster, optionally waiting for responses before responding to the client.

Basic.Consume

If the consume request is for a private queue, no alteration to the processing is required. However, if it is for a shared queue then the broker must additionally replicate the message to all other brokers.

Basic.Cancel

If the cancel request is for a subscription to a private queue, no alteration to the processing is required. However, if it is for a shared queue then the broker must additionally replicate the message to all other brokers.

Basic.Publish

The handling of Basic.Publish only differs from the non-clustered case where (a) it ends up in a shared queue or (b) it ends up in a 'proxy' for a private queue that is hosted within another member of the cluster.

When the published message ends up in a shared queue, the broker must be aware of whether the message was published to it by another broker or by an application client. Messages that come from other brokers are dispatched to the local brokers own application client subscribers. Messages that come from application clients are either dispatched to the next application client or relayed to another broker. A round-robin scheme applies here where each subscriber, whether a 'real' subscriber or a consumer in a relay link to another broker, gets its 'turn'.

In other words the allocation of a message to a consumer on a shared queue happens at the first broker to receive the publish request from the application. All brokers signal their local consumer count by propagating the Basic.Consume (and Basic.Cancel) messages they receive from clients so each broker has a local view of the cluster wide distribution of consumers which can be used to achieve a fair distribution of messages received by that broker.

As each broker can receive messages from the application, strict round-robin delivery is not guaranteed, but in general a fair distribution will result. Brokers should remember the next consumer to receive messages from the application and also the next consumer to receive messages from the cluster.

A local broker's view of consumer distribution is updated asynchronously with respect to message publishing and dispatch. This means that the view might be stale with regard to the remote consumer counts when the next consumer for a message is determined. It is therefore possible that one broker directs a message to a broker that it thinks has a consumer, but when that message arrives at the remote broker the consumer has disconnected. How this is handled should be controlled through different policies: pass it on to another broker, possibly with the redelivered flag set (particularly if it goes back to the broker it came from), discard the message or hold on to it for a finite period of time and deliver it to any application consumer that subscribes in that time.

The situation just described is essentially the same situation as in a non-clustered case where a consumer disconnects after a message has been sent to it, but before it has processed that message. Where acknowledgements aren't used the message will be lost, where acknowledgements or transactions are used the message should be redelivered, possible out of sequence. Of course in the clustered case there is a wider window in which this scenario can arise.

Where the messages is delivered to a proxied private queue, that message is merely relayed on to the relevant broker. However, It is important that where more than one proxied queue to the same target broker are bound to the same exchange, the message only be relayed once. The broker handling the Basic.Publish must therefore track the relaying of the message to its peers.

Failure Analysis

As mentioned above, the primary objective of this phase of the clustering design is to enable the scaling of a system by adding extra broker processes that cooperate to serve a larger number of clients than could be handle by one broker.

Though fault tolerance is not a specific objective yet, the cluster must allow for the failure of brokers without bringing the whole system to a halt.

The current design (and implementation) only handles process failures entirely satisfactorily. Network failures* result in the exclusion of brokers from the cluster and will behave reasonably only where the view of reachability is consistent across the cluster. Network partitions between the cluster nodes will result in independent clusters being formed and there is currently no provision for merging these once the partition heals.

- failures here means anything that causes a tcp stream to fail; a relatively straightforward improvement would be to buffered unacknowledged requests that have been broadcast allowing attempts to re-establish a tcp connection on failure and replaying the messages (assuming idempotent messages)

The group abstraction described above does not provide virtual synchrony. When a broker fails while performing a broadcast to the group, the result will not be uniform across the other members. Where synchronous propagation is used, the client will be ware of this state as it will not have received the response from the broker and will reissue the request on failing over to another broker. (The current failover as implemented in the Qpid client will actually recreate all state required by the client).