Cluster Design Note

Reliable Broker Cluster

This document describes cluster design and implementation as of 19 June 2009.

Overview

A *Reliable Broker Cluster*or just *cluster* is a group of brokers collaborating to present the illusion of a single broker with multiple addresses. The cluster is *active-active*, that is to say each member broker maintains the full state of the clustered broker. If any member fails, clients can fail-over to any other member.

New members can be added to a cluster while it is running. An established member volunteers to provide a state update to the new member. Both updater and updatee queue up cluster activity during the update and process it when the update is complete.

The cluster uses the CPG (Closed Process Group) protocol to replicate state. CPG was part of Open AIS package, it is now part of the corosync package. To avoid confusion with AMQP messages we will refer to CPG multicast messages as *events.*

CPG is a *virtual synchrony* protocol. Members multicast events to the group and CPG ensures that each member receives all the events *in the same sequence*. Since all members get an identical sequence of events, they can all update their state consistently. To achieve consistency, events must be processed in the order that CPG presents them. In particular members wait for their own events to be re-delivered by CPG before acting on them.

Implementation Approach

The cluster implementation is highly decoupled from the broker. There's no cluster-specific code in the general broker, just a few hooks that the cluster uses to modify broker behavior.

The basic idea is that the cluster treats the broker as a black box and assumes that provided it is fed identical input, it will produce identical results. The cluster::Connection class intercepts data arriving for broker Connections. and sends that data as a CPG event. As data events are delivered by CPG, they are fed to the original broker::Connection objects. Thus each member sees all the data arriving at all the members in the same sequence, so we get the same set of declares, enqueues, dequeues etc. happening on each member.

This approach replicates *all* broker state: sessions, connections, consumers, wiring etc. Each broker can have both direct connections and *shadow* connections. A shadow connection represents a connection on another broker in the cluster. Members use shadow connections to simulate the actions of other brokers, so that all members arrive at the same state. Output for shadow connections is just discarded, brokers only send data to their directly-connected clients.

This approach assumes that the behavior of the broker is *determinisitc*, that it is completely determined by the input data fed to the broker. There are a number of cases where this does not hold and the cluster has to take steps to ensure consistency:

- Allocating messages: the stand-alone broker allocates messages based on the writability of client connections.
- Client connection disconnects.
- Timers: any action triggered by a timer may happen at an unpredictable point with respect to CPG events.

Allocating messages

The cluster allocates messages to consumers using CPG events rather than writability of client connections. A cluster connection that has potentially got data to write sends a *do-output* event to itself, allowing it to dequeue N messages. The messages are not actually dequeued until the do-output event is re-delivered in sequence with other events. The value of N is dynamically estimated in an attempt to match it to the rate of writing messages to directly connected clients. All the other members have a shadow connection which allows them to de-queue the same set of messages as the directly connected member.

Client disconnects

When a client disconnects, the directly-connected broker sends a deliver-close event via CPG. It does not actually destroy the connection till that message is re-delivered. This ensures that the direct connection and all the shadows are destroyed at the same point in the event sequence.

Actions initiated by a timer

The cluster needs to do some extra work at any points where the broker takes action based on a timer (e.g. message expiry, management, producer flow control) See the source code for details of how each is handled.

Error Handling

There are two types of recoverable error

- *Predictable* errors occur in the same way on all brokers as a predictable consequence of cluster events. For example binding a queue to a non-existent exchange.
- *Unpredictable* errors may not occur on all brokers. For example running out of journal space to store a message, or an IO error from the journal.

Unpredictable errors must be handled in such a way that the cluster does not become inconsistent. In a situation where one broker experiences an unpredictable error and the others do not, we want the broker in error to shut down and leave the cluster so its clients can fail over to healthy brokers.

When an error occurs on a cluster member it sends an error-check event to the cluster and stalls processing. If it receives a matching error-check from all other cluster members, it continues. If the error did not occur on some members, those members send an error-check with "no error" status. In this case members that did experience an error shut themselves down as they can no longer consistently update their state. The member that did not have the error continue, clients can fail over to them.

Transactions

Transactions are conversational state, allowing a session to collect changes for the shared state and then apply them all at once or not at all.

For TX transactions each broker creates an identical transaction, they all succeed or fail identically since they're all being fed identical input (see Error Handling above for what happens if a broker doesn't reach the same conclusion.)

DTX transactions are not yet supported by the cluster.

Persistence and Asynchronous Journaling

Each cluster member has an independent store, each recording identical state.

A cluster can be configured so that if the cluster is reduced to a single member (the "last man standing") that member can have transient data queues persisted.

Recovery: after a total cluster shutdown, the state of the new cluster is determined by the store of the *first* broker started. The second and subsequent brokers will get their state from the cluster, not the store.

At time of writing there is a bug that requires the stores of all but the first broker to be deleted manually before starting the cluste

Limitations of current design

There are several limitations of the current design.

Concurrency: all CPG events are serialized into a single stream and handled by a single thread. This means clustered brokers have limited ability to make use of multiple CPUs. Some of this work is pipelined, so there is some parallelism, but it is limited.

Maintainability: decoupling the cluster code from the broker and assuming the broker behaves deterministically makes it very easy for developers working on the stand-alone broker to unintentionally break the cluster, for example by adding a feature that depends on timers. This has been the case in particular for management, since the initial cluster code assumed only the queue & exchange state needed to be replicated, whereas in fact all the management state must also be replicated and periodic management actions must be co-ordinated.

Non-replicated state: The current design replicates all state. In some cases however, queues are intended only for directly connected clients, for example management queues, the failover-exchange queues. It would be good to be able to define replicated and non-replicated queues and exchanges in these cases.

Scalability: The current cluster design only addresses reliability. Adding more brokers to a cluster will not increase the cluster's throughput since all brokers are doing all the work. A better approach would move move some of the work to be done only by the directly-connected broker, and to allow messages to "bypass" the cluster when both producer and consumer are connected to the same member.