OpenStack internal messaging at the edge: In-depth evaluation

Ken Giusti
Javier Rojas Balderrama
Matthieu Simonin
Who's here?

Ken Giusti

Javier Rojas Balderrama
Matthieu Simonin

Fog Edge and Massively Distributed Cloud Working Group (a.k.a FEMDC)
Wed. 5:30pm - 6:10pm
Challenges at the edge

Conceptual challenges

- Scalability
- Locality
- Placement
- Resiliency
- ...

[Diagram showing layers of sites: Core Network, Regional sites, Local sites, Edge sites, DC1, DC2]
**Messaging challenges at the edge**

**Conceptual challenges—the messaging perspective**

- **Scalability** increase the number of communicating agents
- **Locality**
  - Keep control traffic in the same *latency domain* (site) as much as possible
  - Mitigate control traffic over WAN:
    - APIs
    - Database state accesses and internal management
      (Thursday, 9AM: *Keystone in the context of Fog/Edge MDC*)
    - **Remote Procedure Calls**

➢ Scope: Openstack’s **Remote Procedure Calls** in a massively distributed context
What’s next?

- Oslo.messaging
- Scalability evaluation
- Locality evaluation
- Lessons learnt
Openstack Internal Messaging
Openstack Interprocess Communications

Remote Procedure Call (RPC)
Openstack Interprocess Communications

- Oslo.messaging
  - Part of the OpenStack Oslo Project
    - [https://wiki.openstack.org/wiki/Oslo](https://wiki.openstack.org/wiki/Oslo)
  - APIs for messaging services
  - Remote Procedure Call (RPC)
    - Inter-project control messaging in OpenStack
  - Abstraction - hides the actual message bus implementation
    - Opportunity to evaluate different messaging architectures
Remote Procedure Call (RPC)

- Synchronous request/response pattern
- Three different flavors:
  - Call - typical request/response
  - Cast - request/no response expected
  - Fanout - multicast version of Cast
- How does the message get to the proper server???
oslo.messaging Addressing (Targets)

- Service Addressing
  - Project assigns servers a well known address
    - Example: “Service-A”
  - Server subscribes to that address on message bus
  - Clients sends requests to “Service-A”
  - Represented by a **Target** Class in the API
  - Unique to a particular Server
    - Direct messaging
  - Or Shared among Servers
    - Load balancing/Multicast
oslo.messaging Alternative Message Buses

- Supports multiple underlying messaging implementations:
  - RabbitMQ Broker (based on AMQP 0-9.1 prototype)
  - Apache Qpid Dispatch Router (AMQP 1.0 ISO/IEC 19464)
Broker and brokerless approaches to RPC

- Brokered RPC Messaging (RabbitMQ)
  - Centralized communications hub (broker)
  - Queues “break” the protocol transfer
  - Non-optimal path
Broker and brokerless approaches to RPC

- Brokered RPC Messaging (RabbitMQ)
  - Centralized communications hub (broker)
  - Queues “break” the protocol transfer
  - Non-optimal path
Broker and brokerless approaches to RPC

- Brokered RPC Messaging (RabbitMQ)
  - Centralized communications hub (broker)
  - Queues “break” the protocol transfer
  - Non-optimal path
Broker and brokerless approaches to RPC

- Brokerless (Apache Qpid Dispatch Router)
  - Deployed in any Topology
  - Dynamic *Routing Protocol* (Dijkstra)
  - Least Cost Path between RPC Client & Server
Broker and brokerless approaches to RPC

- Brokerless (Apache Qpid Dispatch Router)
  - Deployed in any Topology
  - Dynamic Routing Protocol (Dijkstra)
  - Least Cost Path between RPC Client & Server
Broker and brokerless approaches to RPC

- Brokerless (Apache Qpid Dispatch Router)
  - Deployed in any Topology
  - Dynamic Routing Protocol (Dijkstra)
  - Least Cost Path between RPC Client & Server
The Tools used for Testing
Oslo messaging benchmarking toolkit

Oslo Messaging Benchmarking Tool
ombt:  https://git.io/vp2kX

Ombt orchestrator
oo:  https://git.io/vp2kh
Scalability evaluation
Evaluation

Methodology

  - Six scenarios (synthetic / operational)
  - Two drivers
    - Router
    - Broker
  - Three comm. patterns
    - Cast
    - Call
    - Fanout
  - Grid’5000 testbed
Scalability evaluation

Synthetic scenarios (TC1, TC2, TC3)

- Single (large) shared distributed target
  - Global shared target
  - Scale the # of producers, constant throughput
  - How big a single target can be?
- Multiple distinct distributed targets
  - Many targets running in parallel
  - Scale the # of targets
  - How many targets can be created?
- Single (large) distributed fanout
  - Scale the # of consumers
  - How large a fanout can be?
Scalability evaluation

Parameters

- # of calls, # of agents (producers, consumers), call types
- Bus topologies
  - RabbitMQ cluster of size 1, 3, 5
  - Complete graph of routers of size 1, 3, 5
  - Ring of routers up to size 30 (inc. latency between routers)
- Bus load
  - Light: 1K msgs/s
  - Heavy: 10K msgs/s

```bash
> oo test_case_2 --nbr_topics 100 --call-type rpc-call --nbr_calls 1000
```
```bash
> oo campaign --incremental --provider g5k --conf conf.yaml test_case_1
```
Scalability evaluation

Memory consumption: Single shared target (TC1) - rpc-call

- Single shared target (TC1) - rpc-call
  - >25GB
  - ~12GB each
  - <10GB each
  - <5GB
  - ~2 GB each
  - <2GB each

max # of supported agents

![Graph showing memory consumption](image)
Scalability evaluation

CPU consumption: Single shared target (TC1) - rpc-call

- >20 cores
- ~3 cores
- >15 cores each
- ~2 cores each
- <10 cores each
- ~1 core each
Scalability evaluation

Latency: Single shared target (TC1) - rpc-call - 8K clients
Scalability evaluation

Latency: Multiple distinct targets (TC2) - rpc-call - 10K Targets
Scalability evaluation

Latency: single large fanout (TC3) - rpc-fanout - 10K consumers
Wrap up

- All test cases
  - Routers are lightweight (CPU, memory, network connections)
- Single shared distributed target:
  - Implicit parallelism is observed only with routers (single queue in brokers)
  - Scale up to 10K producers
- Multiple distinct distributed targets:
  - Similar behaviour for both drivers because of short buffering
  - Scale up to 10K targets (20K agents)
- Single distributed fanout:
  - Router is less sensitive to the size of broadcast
  - Scale up to 10K consumers
Locality evaluation
**Locality evaluation**

Conceptual challenges: reminder

Multisite: producers and consumers spread over different distant locations

- Scalability

- Locality
  - Keep traffic in the same *latency domain* as much as possible
Locality evaluation
Strategies: Centralized message bus

- **Producer in site 1**
  - Need to break the symmetry of the consumers
    - Give less `rabbit_qos_prefetch_count/rpc_server_credit` to remote consumers
    - Effects depends on
      - Latency
      - Actual workload

- **Producer in site 2**
  - Sub-Optimal data path

➢ Bad locality in the general case
Locality evaluation

Strategies: Sharded Message bus

- E.G : Nova CellsV2
- Strict locality
  - A shard can be a *latency domain*
  - Traffic remains in a shard
- Caveat :
  - Routing requests (consumer index, inter-shard communication),

➢ Routing is deferred to the application
Locality evaluation
Strategies: Alternative to sharding

- A tree of routers
  - Routing is transparent to the application
  - How locality is ensured?
Two levels of locality

- **Strict locality:**
  - Closest mode
  - Local consumers are picked over remote
  - Caveat: local consumers backlog

- **Load-sharing**
  - Balanced mode
  - Cost is associated with every consumer
  - Consumer with the lower cost is picked
  - Cost is dynamic

**Locality evaluation**

Strategies: Decentralized bus (amqp 1.0 only)
Locality evaluation
Decentralization of the message bus

➢ Cost is dynamic
➢ Load sharing is locality aware

Up to n=30 sites (ring)

Increase the message rate
Increase the intersite latency

Evaluate the locality in message delivery

Locality vs message rate (LAN latency / inter-router cost=1)

99% Local
66% Local

Increasing load

Locality vs Latency (message rate = 10K/s, inter-router cost=1)

66% Local

Increasing inter-site latency

- 95% Local
Locality evaluation
Decentralization of the message bus

From High Availability OpenStack to **High Locality** OpenStack

- One **single** Openstack
- Shared bus (router mesh)

➢ **RPCs locality only**

In the future
- APIs locality
- Database locality

➢ **Join the FEMDC !**
Lessons learnt
Lessons learnt

- Communication layer of OpenStack
  - Two implementations: AMQP 0-9.1 / rabbitmq and AMQP 1.0 / qpid-dispatch-router

- Centralized deployments
  - Similar scalability
  - Router are lightweight and achieve low latency message delivery esp. under high load

- Decentralized Deployments
  - A mesh of routers offers guarantees in the locality of the messages
  - 2 levels of locality: strict / locality-aware load sharing

- Toward a high-locality OpenStack for multisite deployment
  - Leveraging a router mesh
  - Same ideas need to be applied to APIs and database

- FEMDC working group is studying different options
  - Wed. 4:40pm - 5:20pm
OpenStack internal messaging at the edge: In-depth evaluation

http://bit.do/oo-jupyter-tc1  kgiusti@redhat.com
http://bit.do/oo-jupyter-tc2  javier.rojas-balderrama@inria.fr
http://bit.do/oo-jupyter-tc3  matthieu.simonin@inria.fr
http://bit.do/oo-tc1-ring30