Scalability and Locality Awareness of Remote Procedure Calls: An Experimental Study in Edge Infrastructures

Javier Rojas Balderrama, Matthieu Simonin

To cite this version:

Javier Rojas Balderrama, Matthieu Simonin. Scalability and Locality Awareness of Remote Procedure Calls: An Experimental Study in Edge Infrastructures. CloudCom 2018 - 10th IEEE International Conference on Cloud Computing Technology and Science, Dec 2018, Nicosia, Cyprus. pp.40-47, 10.1109/CloudCom2018.2018.00023 . hal-01891567

HAL Id: hal-01891567

https://hal.inria.fr/hal-01891567

Submitted on 9 Oct 2018

Scalability and Locality Awareness of Remote Procedure Calls: An Experimental Study in Edge Infrastructures

Javier Rojas Balderrama
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
javier.rojas-balderrama@inria.fr

Matthieu Simonin
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
matthieu.simonin@inria.fr

***Abstract*—Cloud computing depends on communication mechanisms implying location transparency. Transparency is tied to the cost of ensuring scalability and an acceptable request responses associated to the locality. Current implementations, as in the case of OpenStack, mostly follow a centralized paradigm but they lack the required service agility that can be obtained in decentralized approaches.**

In an edge scenario, the communicating entities of an application can be dispersed. In this context, we focus our study on the inter-process communication of OpenStack when its agents are geo-distributed. More precisely, we are interested in the different Remote Procedure Calls (RPCs) implementations of OpenStack and their behaviours with regards to three classical communication patterns: anycast, unicast and multicast. We discuss how the communication middleware can align with the geo-distribution of the RPC agents regarding two key factors: scalability and locality. We reached up to ten thousands communicating agents, and results show that a router-based deployment offers a better trade-off between locality and load-balancing. Broker-based suffers from its centralized model which impact the achieved locality and scalability.

***Index Terms*—Edge computing, Remote Procedure Calls, Message Oriented Middeware, Grid'5000, OpenStack, reproducible research**

I. Introduction

Telecommunication operators currently push towards novel application deployments that benefit from the distributed nature of the infrastructures. A classical model for such infrastructures considers many sites (i.e., computing resources and storage capacity) along the path from the core to the edge of the network. Few milliseconds of latency may be registered between the core and the closest sites while long-haul communication may occur at the edge of the network. In this context, cloud-based applications can potentially be deployed across different geographically distributed sites. In a standard approach, parts of the application requiring low-latency interaction with the end user can be placed close to the edge while the others can be placed closer to the core.

OpenStack[1] is largely used to build and manage cloud computing platforms. Nevertheless, the original design of OpenStack targets the case where a small number of large sites is the norm. In the aforementioned model, OpenStack would need to operate a large number of small sites. Hence, OpenStack control-plane agents (responsible for managing the virtual resources) need to be scaled-out and shifted into geographically dispersed locations. This new deployment model for OpenStack raises different challenges. First, the scalability of the internal management system. OpenStack is designed in an elastic manner: most agents can be replicated for load-sharing purpose. Elasticity defers the scalability challenge to the underlying database and communication middleware. Second, the geo-distribution of agents needs to be taken into account. It is an opportunity to provide a certain level of locality to the end-user. On the one hand, users expect low-latency access to their application meaning the data path created for an application must remain as local as possible. On the other hand, the control traffic can also benefit from locality by keeping states (e.g., database) and inter-process communication also as local as possible. Thus, locality can reduce the completion time of requests by minimizing long-haul communication.

OpenStack control traffic provides a high-level Remote Procedure Call (RPC) abstraction and endorses different implementations: centralized broker-based implementations sit next to more decentralized brokerless or hybrid approaches such as those described in the AMQP 1.0 standard [1].

This paper presents an evaluation of the RPC layer of OpenStack in a geo-distributed context. The study focus more precisely on:

- **Scalability**. What is the impact of having a massive number of communicating agents on the application and the underlying message bus?
- **Locality**. What is the impact of the geo-distribution of the communicating agents, and how can it be mitigated?

The contribution of this paper is an extensive experimental evaluation of RPC and bus agents deployment in two different configurations: centralized and decentralized. This contribution is aligned to the efforts of the professional community of harnessing current deployment capabilities to edge infrastructures [2], [3].

[1] https://openstack.org

The rest of the paper is structured as follow: Section II reviews the main criteria of the evaluation. Section III describes the metrics we focus on, and the framework we built to perform reproducible and automatized benchmarks in a geo-distributed context. Experimental results are presented and discussed in sections IV, and V. Finally, the paper ends with a conclusion and outlines future directions.

II. Experimental model

Classical unicast RPCs (i.e., point-to-point) require a tight spatial coupling [4] of the client and the server (they must know each other) and a strong synchronisation (server should be listening for incoming communication). On a message oriented middleware (MOM), clients and servers may also declare their communication interest through an abstract *target*. This target allows agents to decouple the messaging entities using higher-level communication patterns like anycast and multicast. OpenStack agents use concurrently those three RPC patterns. They serve us as a reference for designing three evaluation scenarios:

- **Anycast Scenario** (AS) when a request is delivered to only one server interested in the target.
- **Unicast Scenario** (US) when a request is delivered in a point-to-point fashion.
- **Multicast Scenario** (MS) when a request is replicated and sent to all servers interested in the target.

The identification of above scenarios is the first step to break up the complexity of the evaluation. Indeed, as described in the next sections, there are different parameters that can influence a scenario's behaviour. We provide some background for each identified parameter, and we also state our choices for the evaluation.

A. Agents geo-distribution

RPC agents and bus agents may be deployed on geographically distant locations. This geo-distribution can force the messages exchange by communicating entities to travel over long-haul links. The different network characteristics of those links [5] can result in a degradation of the application performance, or worse, application failures. The former may be mainly due to the introduction of latency in message transfers, and the latter may be the consequence of packet loss that cannot be handled by the application or the underlying protocol (e.g., TCP). In this paper we mainly focus on studying the impact of the geo-distribution of the agents on AS. Indeed in this scenario, the bus usually load balance the requests on the available servers. This strategy could be optimized to take into account the actual distance between the communicating entities. In US and MS the bus has to transfer the requests to a predefined set of agents, this defers any optimization to the initial agents placement.

B. Scenarios size

We evaluate the scalability by increasing the size of the scenario in terms of RPC agents, and the resulting number of transferred messages. We define the size in a different way for each scenario. The size of AS corresponds to the number of RPC clients and RPC servers. The size of US is the number of targets. Therefore, it increases proportionally the number of RPC agents: one client and one server for each target. Finally, the size of MS is only the number of RPC servers receiving the messages. In a realistic deployment there are more clients than servers. That is the case in AS while they are equal in US. Note that in MS only one client is considered.

C. RPCs flavours

A synchronous call blocks the client until the return value goes back to the client application. In some situations, blocking the client control flow is not required and the return value is backtracked to the client application once it is ready. For this reason asynchronous calls are often exposed in RPC libraries. The *fire-and-forget* pattern is a special type of asynchronous call where the client does not even need to receive the return value of the function. OpenStack specifies different RPC types including `rpc-call` and `rpc-cast`. The `rpc-call` is synchronous and the `rpc-cast` is fire-and-forget. Both types also differ on the delivery guarantee: `rpc-call` follows the *at-least-once* semantics whereas `rpc-cast` only offers the *at-most-once*. Moreover, a `rpc-cast` is likely dropped in case of congested traffic.

In our evaluation, RPCs flavours are tested separately. In OpenStack AS and US support both `rpc-call` and `rpc-cast`. MS only supports a special version of `rpc-cast`, namely `rpc-fanout`, to broadcast asynchronously a request to a set of servers. Note that the calling is more sensitive to the geo-distribution of the agents than casting because it needs waiting the server response. As a result, it pays twice the latency of the link between the client and the server. Additionally, more resources need being provisioned on the message bus to handle calls (e.g., return channel), so calls may consume more resources.

D. Bus topology

The aforementioned RPCs flavours can be implemented on different backends using different drivers. Most of the time, operators rely on a broker-based backend (e.g., RabbitMQ[2]) as they are robust, and battle-tested drivers based on the AMPQ 0.9-x standard [6]. Nevertheless, the AMQP 1.0 [1] defines *routers* as new type of intermediary. OpenStack supports both backends to implement the previously described patterns. In the following we discuss the deployment strategies implemented in our study.

Broker-based message bus: A broker is a general purpose MOM acting as an intermediary between the messaging entities. A broker stores (in a queue) and forwards messages, allowing a spatial and temporal decoupling between communicating agents. But in the context of RPCs communication, this decoupling is questionable as clients and servers must be synchronised. For scalability reasons, brokers can be clusterized into a big logical broker. RPC clients and servers interact

[2] https://rabbitmq.com

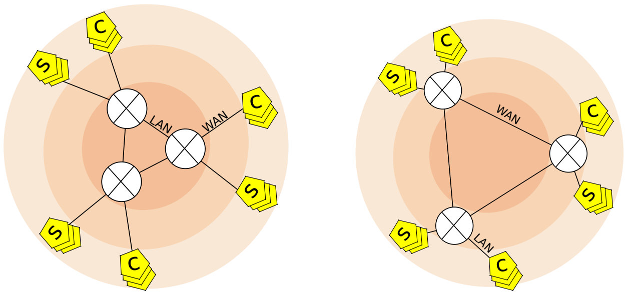


Fig. 1: Bus agent configurations. Centralized (left) where agents are interconnected through a LAN and they are connected to RPC agents using a WAN; decentralized (right) where agents are interconnected through a WAN and they are connected to the RPC agents using a LAN. In both cases RPC clients (C) and RPC servers (S) are located in the edge.

with this logical broker while the load is distributed among different bus agents. In our study we refer to RabbitMQ, the de facto standard in OpenStack, as the reference broker implementation.

Router-based message bus: Routers can be organized in a custom topology and dynamically computes routes for messages. They are stateless and do not store messages. Therefore, routers require producers and consumers to be temporally coupled. Similarly to brokers, a router mesh can be extended by adding new routers and links to the original ones. We use Apache qpid-dispatch-router[3] as the reference implementation in our study.

In our evaluation, we consider two bus configurations: a RabbitMQ cluster, and a complete mesh of qpid-dispatch-routers. In order to embrace the edge requirements in terms of geo-distribution of the agents, we consider two possible deployments for the bus agents shown in Fig. 1:

- Centralized deployment. The bus agents are deployed in the same latency domain (e.g., the same location with LAN connectivity). In this deployment long links are found between the RPC agents and the bus agents.
- Decentralized deployment. In order to reach a better locality, the bus agents are pushed closer to the RPC agents. Therefore, links between bus agents become longer in comparison to the previous case.

Note that RabbitMQ proposes *federation* as an alternative deployment model. This model has limited support for message routing and requires to statically configure the messages path accross the different brokers. These limitations make this model not applicable to the RPC implementation of OpenStack and thus is not considered in the current study.

III. Tools and metrics

We have developed the Ombt orchestrator,[4] an application to evaluate the performance of communication agents and RPC agents in synthetic edge configurations. Ombt orchestrator is based on two tools: Ombt and EnOSlib. The Oslo messaging benchmarking tool[5] (Ombt) uses the library of OpenStack for messaging[6] to measure the latency and throughput of RPC and notification transactions. EnOSlib [7] is a high level library for deploying and executing applications, and collecting results in a distributed fashion.

Ombt orchestrator eases the management of the parameters described in section II. First, the geo-distribution of agents is emulated by means of traffic control capabilities offered by the Linux kernel's network stack. Second, given a size, it deploys any of the three scenarios. Moreover, clusters of brokers of any size and complex topologies of routers are supported. Experimental campaigns can be performed in an automatic manner across a (large) infrastructure with thousands of cores on various testbeds. Alongside the agents, a dedicated controller is deployed assessing system metrics by means of additional tools: Telegraf and InfluxDB.[7]

In fact, we identify two types of metrics: those from the application layer and those coming from the system. The leveraged applications provide metrics such as latency distribution and message rate. These metrics can be analyzed along with the potential execution errors raised by the applications. It is important to note that we differentiate two types of latency: the *round-trip latency* (a.k.a. RTT latency) and the *one-way latency*. The former is associated with RPCs of type `rpc-call` and it is the time taken from the client application to send the request and receive the answer from the server. It is measured exclusively on the client side. The latter is associated with RPCs of type `rpc-cast` and corresponds to the time for a request to go from the client application to the server application. Latency is thus measured both in the client and server application. In this case, a strict time synchronisation is required between the RPC agents. In terms of implementation that synchronisation is done with a dedicated NTP cluster. For the system metrics, we are specially interested in CPU and RAM consumption, active TCP connections and network traffic for all agents.

IV. Results

Following subsections present the common experimental setup and results. We performed two set of experiments based on centralized and decentralized deployments of the bus agents as shown in Fig. 1.

A. Setup

All the experiments ran on the Paravance[8] cluster of Grid'5000 [8] composed of 72 physical nodes, each having 16 cores and 128 GB RAM. Qpid-dispatch-router 1.0.1 and RabbitMQ 3.7.4 were deployed. Oslo messaging 5.35.0 was used as RPC library on Debian 9.3. The release version of ombt orchestrator was 1.1.1. This testing framework allows

[3] https://qpid.apache.org/components/dispatch-router/

[4] https://github.com/msimonin/ombt-orchestrator

[5] https://github.com/kgiusti/ombt

[6] https://github.com/openstack/oslo.messaging

[7] https://www.influxdata.com

[8] https://www.grid5000.fr/mediawiki/index.php/Rennes:Hardware

running experimental campaigns in a fully automated manner. A campaign corresponds to a set of combined parameters. For each generated combination a benchmark is run and several metrics are collected. For instance, Table I shows a set of parameters where 12 (2 call types×6 bus configurations) combinations of common parameters are combined with 6 agents configurations for each scenario, resulting in 72 combinations of parameters per campaign. Each combination is calibrated to run within 5 minutes with a timeout of double that time. If the timeout is triggered an execution is marked as failed becoming our service-level agreement (SLA). For the sake of conciseness in this paper, we only present a subset of all evaluated combinations.

Parameter	**Values**					
	Anycast					
Clients	1 000	2 000	4 000	6 000	8 000	10 000
Servers	20	20	20	20	20	20
Pause	0.1	0.2	0.4	0.6	0.8	1.0
Messages	300 000	300 000	300 000	300 000	300 000	300 000
	Unicast					
Targets	1 000	2 000	4 000	6 000	8 000	10 000
Pause	0.1	0.2	0.4	0.6	0.8	1.0
Messages	300 000	300 000	300 000	300 000	300 000	300 000
	Multicast					
Servers	1 000	2 000	4 000	6 000	8 000	10 000
Pause	1.0	1.0	1.0	1.0	1.0	1.0
Messages	100	100	100	100	100	100
	Common					
Call type	`rpc-call`, `rpc-cast`					
Bus conf.	1 broker, 1 router, 3 brokers, 3 routers, 5 brokers, 5 routers					

TABLE I: Parameters of the centralized bus experiments. A campaign involves the combination of call types and bus configurations for each of six sets of number of clients, number of servers, pause between requests to the server and number of messages for the patterns AS US, and MS (72 combinations).

B. *Centralized deployment*

In this set of experiments we consider a bus deployed in a centralized way (see Fig. 1 left). Bus agents are located in the core infrastructure. Two different experiments have been performed: a plain scalability study, and a variation in which RPC agents are pushed to the edge of the network.

1) Scalability study: The goal here is to observe the impact of an increasing number of RPC agents (or the targets depending the scenario) requests on the communication bus.

Parameters: Table I shows all parameters considered for these experiments. An execution campaign is associated to each scenario: AS, US, and MS. We consider a LAN connectivity between the bus and the RPC agents. In AS, the number of clients is increased to reach 10K. In parallel, the inter-request delay (pause parameter) is set to maintain a constant request rate on the message bus to 10K/s. 20K messages per second on the bus are generated (requests and return values are accounted for 2 messages) in `rpc-call` cases. This message rate is the estimated load of the periodic tasks (i.e., heartbeat) of 80K compute nodes deployment in OpenStack. Each client sends 300 messages (300K messages sent in total) to attain the same benchmark duration. Twenty servers are set to keep up all the request load. Then in US, the number of targets evolves as in the previous scenario to have a constant request rate and benchmark duration. Finally, in MS an increasing number of servers is set and only `rpc-fanout` calls are evaluated. Note that, in this last scenario, one request sent by the client generates 10K requests sent to all servers.

Results: Fig. 2 shows the boxplot for the three scenarios. Each one has a specific behaviour summarized as follows.

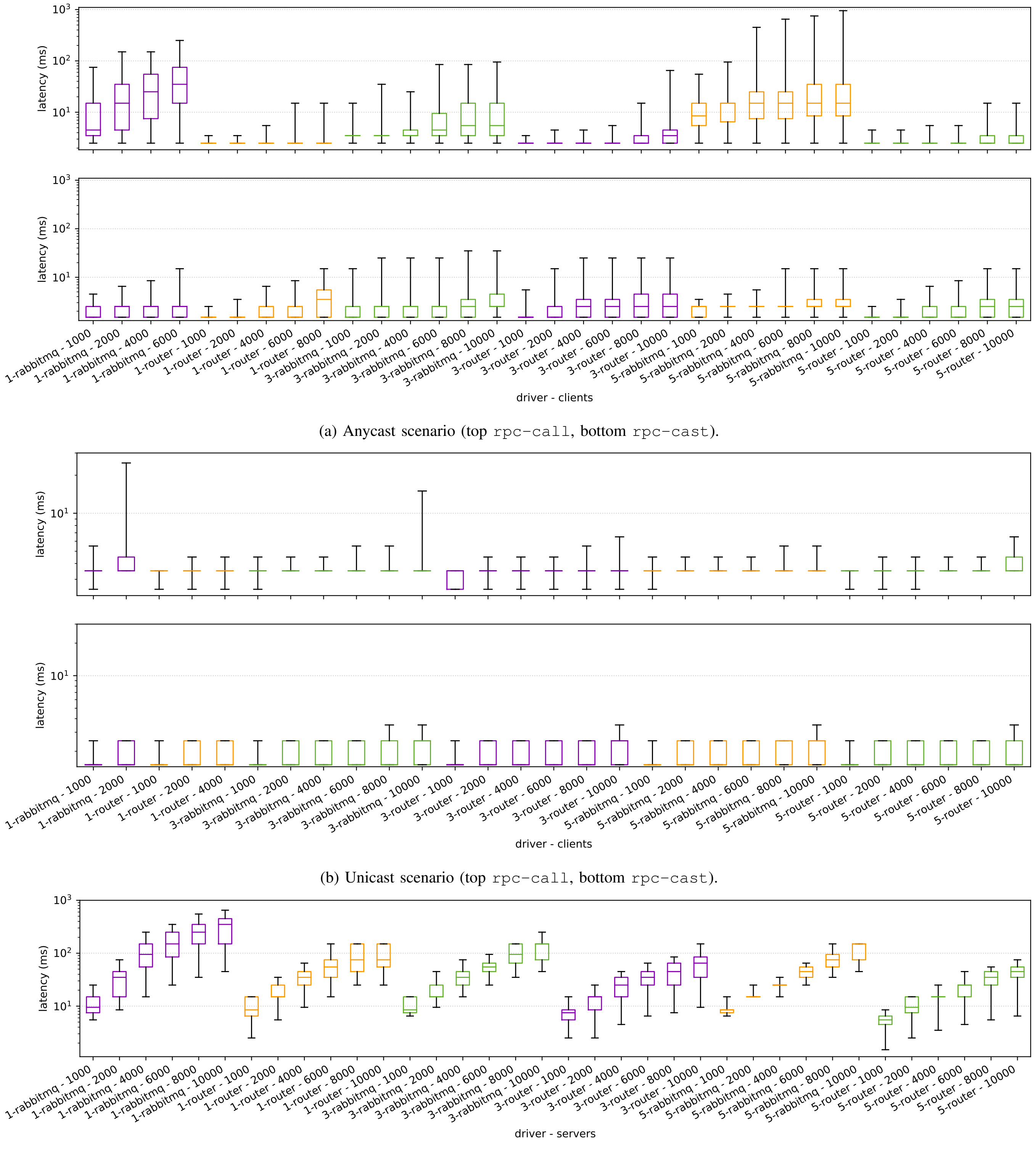
In AS, executions for configurations with only one bus agent never completed for more than 6K clients for brokers and 8K for routers as shown in Fig. 2a. It is due to the default maximum system limit of open file descriptors (16384) or the SLA. This effect is partially verified with the number of TCP connections in Table II. The driver implementation for brokers requires twice the number of connections than routers (i.e., one to send and another to receive). Concerning memory and processors consumption metrics. Brokers require a lot more resources compared to routers with ratios going from at least 9 to 17 times in case of RAM, and from at least 8 to 27 in case of CPU cores. Brokers have higher latencies for both messages types. In order to account a complete communication between agents, `rpc-call` requires at least twice the time. In general, both bus agents perform in the same way. The only difference is brokers report a bigger latency when the number of clients increases.

In US, results of both call types are more similar than AS (Fig. 2b). Since each *target* is assigned to one client and one server, the number of connected agents is twice the number of targets. Additionally, no extra queuing (in the broker case) nor buffering (in the router case) is expected for any target. Nevertheless, the single bus configuration suffers the same limitation as AS. The system metrics are not presented here but they follow the tendency of the AS scenario.

Finally, in MS (Fig. 2c), we observe that both bus agents support the communication pattern in all configurations. Note

Metric	**Bus conf.**	**Clients**					
		1 000	2 000	4 000	6 000	8 000	10 000
RAM (MB)	1 broker	7 735	14 444	21 470	28 268		
	1 router	519	1 286	1 937	2 888	3 906	
	3 brokers	6 935	15 463	23 426	30 445	36 725	40 854
	3 routers	400	826	1 547	2 286	3 713	4 326
	5 brokers	9 583	18 468	28 095	32 659	39 779	45 060
	5 routers	616	1 187	1 712	2 824	3 885	4 565
CPU cores	1 broker	24	22	21	21		
	1 router	1	1	2	2	2	
	3 brokers	27	40	37	47	51	53
	3 routers	1	2	2	2	3	6
	5 brokers	27	37	49	49	54	57
	5 routers	2	2	2	4	3	4
TCP conn.	1 broker	2 632	4 632	8 628	12 628		
	1 router	1 033	2 030	4 025	6 025	8 025	
	3 brokers	2 612	4 639	8 637	12 638	16 643	20 638
	3 routers	1 046	2 047	4 040	6 035	8 038	10 040
	5 brokers	2 655	4 656	8 656	12 656	16 658	20 656
	5 routers	1 051	2 070	4 057	6 048	8 047	10 048

TABLE II: Results of the anycast scenario. System metrics for `rpc-call` call type. Maximum values obtained during the benchmark for memory usage, number of processors and TCP connections.



(a) Anycast scenario (top `rpc-call`, bottom `rpc-cast`).

(b) Unicast scenario (top `rpc-call`, bottom `rpc-cast`).

(c) Multicast scenario (`rpc-fanout`).

Fig. 2: Comparative results of latency for different configurations of bus implementations and number of agents.

that servers do not require any response from the clients to accomplish the message communication, so there are less active TCP connections compared to the two previous scenarios. Although, the difference between brokers and routers in terms of latency is still present.

2) Locality study: Relying on a packet lossless network or a nonexistent latency is not realistic scenario in an edge configuration. The goal of this second experiment is to evaluate

Parameter	Values		
	Paired		
Clients	100	250	1000
Servers	20	20	20
Pause	0.1	0.1	0.1
Messages	150 000	375 000	1 500 000
	Combined		
Delay (ms)	0, 5, 10, 25, 50, 100		
Loss (%)	0, 0.1, 0.5, 1		
Call type	`rpc-call`, `rpc-cast`		
Bus conf.	3 brokers, 3 routers		

TABLE III: Parameters of anycast scenario. A campaign includes latency and packet loss to the call type for and bus configuration. These parameters are combined with each set of paired parameters: number of clients, number of servers, pause between request to the server and number of messages (288 combinations).

the RPC agents behaviour deployed at the edge under influence of a link delay and packet loss.

Parameters: Table III shows all parameters considered for the experiment. The delay and loss rate correspond to the network emulated between the RPC agents and the bus agents. The number of servers remains constant whereas the number of clients increases to generate different load on the bus. For all campaigns a cluster of three brokers and a complete mesh of three routers are considered. The results with more agents does not provide newer elements for discussion. Those results, verified in preliminary tests, are not included here. Only `rpc-call` messages were evaluated because the results of `rpc-cast` in the previous experiment did not show a clear difference between brokers and routers. That behaviour was also verified during the preliminary tests with delay and packet loss.

Results: Fig. 3 shows the boxplots of the message latency while varying the link delay, loss rate, and number of clients for both bus implementations (broker and router). We noticed the absence of application errors during all campaigns. It means that the underlying protocols or applications such as TCP or Oslo messaging manage properly the loss of packets. We also observed the introduction of packet loss only increases the inter-quartile range (IQR) because of retries of packet transmission. Latency in all cases is registered as expected according the parameters configuration.

In summary, a centralized deployment of buses may fit well in an edge environment paying the additional latency, slightly bigger with brokers. This configuration is appropriate in most situations where the bus configuration is transparent to the RPC agents because they work as a sole entity beyond the number of configured instances. Routers support better the request message load before crashing because of overload but the magnitude in both configurations remains the same. In contrast, in a centralized configuration there is no benefit of locality between RPC agents since all communications transit the core of the infrastructure where buses are located.

C. Decentralized deployment

In this experiment we consider a bus deployed in a decentralized way (see Fig. 1 right). Bus agents are shifted closer to the edge where RPC clients and RPC servers are deployed. This time, the goal is to observe the impact of the geo-distribution of all actors on the benchmarks completion while decentralizing the communication bus.

Parameters: Table III details all parameters considered for the experiment. The delay and the loss values are only applied between bus agents unlike Section IV-B2. RPC agents and bus agents are considered in the same latency domain.

Results: Fig. 4 shows, similarly to precedent experiment, boxplots of the message latency while varying the execution parameters. We first observe missing results corresponding to a violation of the SLA. Under the influence of delay and packet loss the broker internal communication became very unstable making broker agents crash. Conversely, no timeout violation was observed with routers. In such deployment the brokers scalability is greatly impacted by the presence of delay and loss. For instance, with no loss and high latency, only a limited number of clients were supported (less than 250 for 50 ms). In the same way, the impact on the scalability was bigger with higher loss. Routers did not suffer those problems: low latency (and thus high throughput) was always achieved. Note that, in this experiment, latency remains lower than the imposed between bus agents.

Indeed, a router assigns a message cost to each possible link that leads to a server. The router then sends the message on the path achieving the least cost. Initially the cost associated to a remote server is higher than the cost associated to local servers. As the cost associated to a server increases with its load, some messages may be offloaded to remote locations. In proportion non-local delivery was rare in our experiments. As a rough estimate, an imposed message rate of 100 message per second leads to 90% of local delivery. This drops to 66% for 1 000 messages per second. In summary, the routing strategy makes the load-balancing locality aware. This explains why, in most cases, the latency perceived by the application is lower than the actual latency between the bus agents.

Figure 5 (left) provides a more detailed distribution of the latency found in brokers. The anycast *target* implementation in RabbitMQ leverages a single queue located in a specific bus agent (the agent where the first server was connected). A request from a client to a server can follow one of the three paths identified in same figure (right) leading to the three modes observed in the latency distribution. As a consequence, locality using a cluster of brokers only occurs for a small portion of messages. For higher loss, TCP re-emission may occurs leading to another modes to appear in addition to the three main identified.

In conclusion, decentralizing a cluster of brokers is only possible under small latency constraints as scalability is greatly impacted by bad network conditions. In decentralized scenarios, routers mitigate the effect of the geo-distribution of RPC agents by achieving a locality-aware load balancing. Messages can thus flow directly from the clients to the local servers. Additionally, as suggested by system metrics of the Table II also verified in this case, routers are lightweight and fit better the capacity constraints of micro datacenters at the edge.

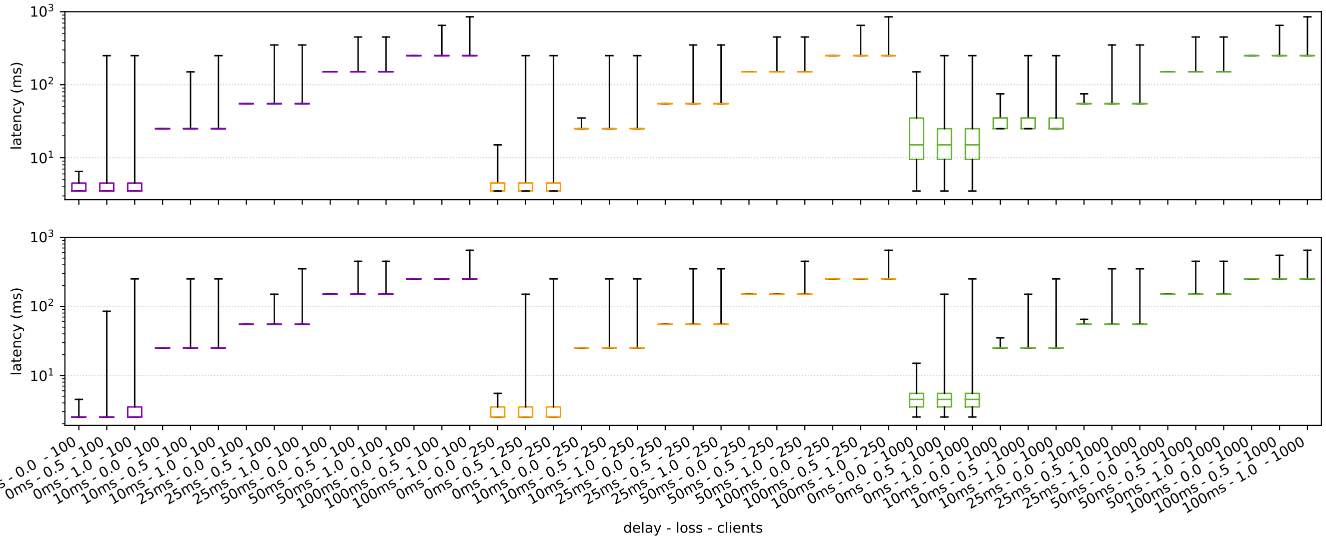


Fig. 3: Results of the anycast scenario in a centralized deployment. Latency boxplots for link delay, packet loss, and number of clients in a 3 agents configuration of `rpc-call` (top brokers, bottom routers).

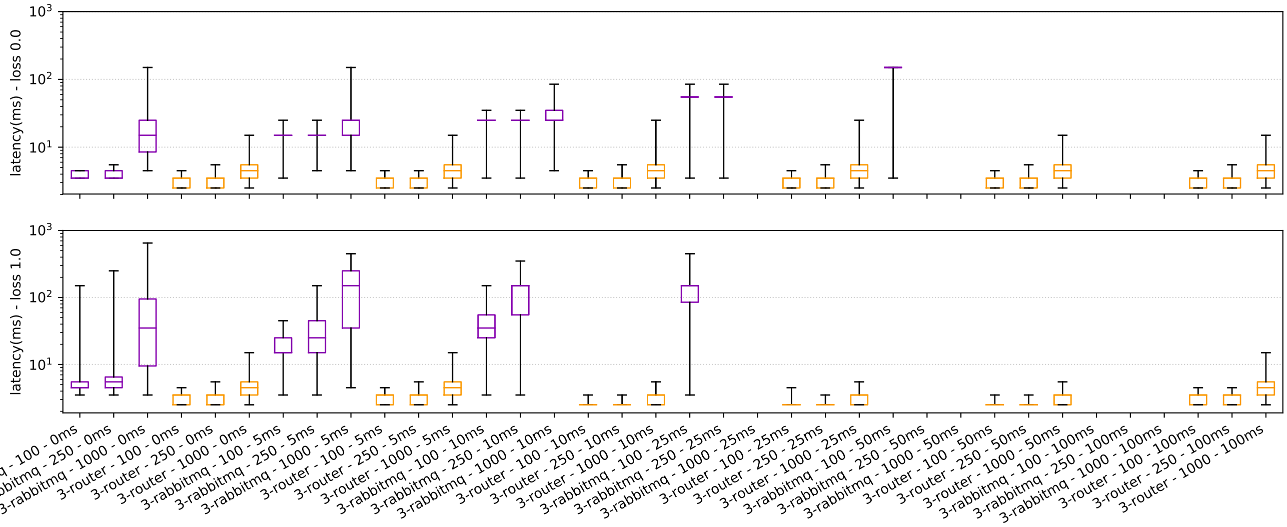


Fig. 4: Results of the anycast scenario in a decentralized deployment. Latency boxplots for bus implementations, number of clients, and link delay of `rpc-call` (top 0% loss, bottom 1% loss).

V. Related work

To the best of our knowledge, very few experimental studies have been conducted on MOM in a cloud context. In [9], and more recently in [10] the study of scalability focuses on raw messages throughput, latency and resource consumption. Those studies stick to a single datacenter use case. Even if our study is limited to OpenStack, it fills a gap by evaluating high level construction such as RPC and by evaluating the impact of having a massive number of geo-distributed agents. As observed in the previous sections, locality plays a key role in various domains. First, scalability by enabling parallel transaction handling among different bus-agents. This leads to decentralize the messages distribution while increasing the overall throughput of the application. Second, reliability by better isolating potential failures domains. A bus-agent is not impacted by a remote network failure since it can deliver message locally. This assumes RPC clients and RPC servers to be co-located. Co-location may not be a general configuration, so in such cases a centralized bus deployment may be considered. However, in the OpenStack, stateless agents (e.g., APIs) are usually load-balanced, and stateful agents (e.g., L3 agents) can be replicated following a leader/slaves design. As a consequence, many geographically distant sites can be equipped with all core services of OpenStack and internal RPC traffic can take advantage of the bus locality. This effect is transparent to the application and may be referred as an implicit locality. Alongside RPC traffic, OpenStack agents use HTTP requests between other agents, and execute database

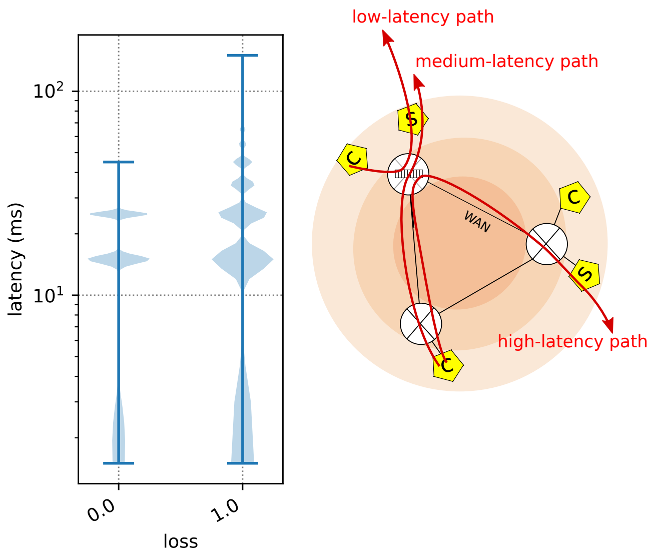


Fig. 5: Violin chart (left) showing the latency modes for brokers (`rpc-cast`, 10 ms inter-agent latency) and the possible message paths in a cluster of three nodes (right).

accesses. Leveraging locality during HTTP requests would require to modify the service discovery of OpenStack in order to be also locality aware. However, achieving implicit states locality during databases transactions is still an open research topic [11].

VI. Conclusion

In this paper, we have studied two different implementations of the Remote Procedure Calls available in OpenStack: a broker-based approach (RabbitMQ) and a router-based approach (Qpid-Dispatch-Router). From the scalability perspective, brokers can be expanded (e.g clustering) by adding new bus agents to cope with an increasing number of connected agents. This allows to distribute the incoming connections and the associated resources consumption among different servers. Nevertheless, under high request rate, message distribution can suffer from bad locality which limits the scalability in this area even in a centralized deployment (e.g RabbitMQ anycast request rate is bound). This locality effect is amplified in a geo-distributed context where it plays a critical role. In this context, the implicit locality achieved by a router mesh contribute to a better scalability because distinct bus-agents can parallelize message handling.

More generally, there is a manifest need of making current geo-distributed infrastructures evolve towards local awareness models minimizing the developing and deployment impact. The road to this *edgification* is not a mere adaptation of current building blocks like communication buses, as it concerns low-level aspects of applications such as messages and data. Implicit locality is a first step in this direction since application code does not require modifications to be applied. In order have a better comprehension, other evaluations are required, namely, configurations with network partitions, fault tolerance.

Our prospective work includes to transpose these results to a fully functional OpenStack and enable locality for the HTTP traffic and database access.

Acknowledgment

The authors would like to thank Orange Labs for partially funding this work under a CRE contract. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

References

[1] *Advanced Message Queuing Protocol (AMQP) v1.0 specification*, International Organization for Standardization, ISO/IEC 19464:2014, May 2014.

[2] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, 2015.

[3] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *The IEEE International Conference on Smart Cloud (SmartCloud 2016)*, New York (NY), USA, Nov. 2016.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[5] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: A platform for high-performance internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[6] *AMQP Advanced Message Queuing Protocol. Protocol Specification v0-9-1*, Advancing open standards for the information society OASIS, AMQP Working Group 0-9-1, Nov. 2008.

[7] R.-A. Cherrueau, M. Simonin, and A. van Kempen, "EnosStack: A LAMP-like stack for the experimenter," in *The IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Honolulu (HI), USA, Apr. 2018.

[8] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *The 6th IEEE/ACM International Workshop on Grid Computing*, Seattle (WA), USA, Nov. 2005.

[9] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann, "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark," *Performance Evaluation*, vol. 66, no. 8, pp. 410–434, 2009.

[10] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *The 11th ACM International Conference on Distributed and Event-based Systems (DEBS'17)*, Barcelona, Spain, Jun. 2017.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, D. Woodford, Y. Saito, C. Taylor, M. Szymaniak, and R. Wang, "Spanner: Google's globally-distributed database," in *The 10th USENIX Symposium on Operative Systems Design and Implementation (OSDI 12)*, Hollywood (CA), USA, Oct. 2012.