Kim and Park *Hum. Cent. Comput. Inf. Sci.* (2020) 10:31
https://doi.org/10.1186/s13673-020-00238-6

Human-centric Computing and Information Sciences

RESEARCH Open Access

Hybrid decentralized PBFT Blockchain Framework for OpenStack message queue

Youngjong Kim* and Jinho Park*

*Correspondence: opensys@gmail.com; j.park@ssu.ac.kr
School of Software, Soongsil University, 369 Sangdo-ro, Dongjak-gu, Seoul, Republic of Korea

Abstract

Cloud computing based on OpenStack is widely used as a distributed computing platform. OpenStack has progressed at a rapid pace, incorporating a variety of service modules; it is supported by many companies, has a community of active developers, and a diverse user base. OpenStack uses message queue to coordinate and exchange operation and status information between services. OpenStack supports various message queue services including RabbitMQ, Qpid, and ZeroMQ, whereas its distribution architecture uses RabbitMQ. As an OpenStack's message queue service, RabbitMQ runs on a controller node as a centralized service. In case of the centralized service, increased usage may cause slowed response times and security vulnerability. This paper proposes a Hybrid decentralized Practical byzantine fault tolerance Blockchain Framework with two-step verification for OpenStack message queue service. When compared to existing OpenStack message queue service, OpenStack with the proposed framework demonstrates identical reliability a faster response time by approximately 46.75% with a two-step verification process and decentralization approach. Additionally, a reduction in the security vulnerability in the OpenStack message queue information with saving the message queue information into each node by blockchain-based decentralized data duplication approach.

Keywords: Cloud, OpenStack, Message queue, Hybrid, Decentralization, PBFT Blockchain

Introduction

OpenStack, which is the open-source Infrastructure-as-a-Service (laaS) Platform Project, is widely used in various fields. The OpenStack project is developing at a rapid pace as a result of the participation and support of a number of major companies and a community of active developers.

OpenStack comprises nodes such as controller, compute etc. Based on its role [1], each node contains Nova, Cinder, Glance etc., which are components of the OpenStack services and form the laaS platform [2].

Each component of OpenStack uses the message queue service to check, exchange and coordinate the information related to the operation and status of the components [3], and the message queue service of OpenStack is the centralized approach service that is executed at the controller node [4].

OpenStack supports tools and libraries including RabbitMQ [5], Apache Qpid [6] and ZeroMQ [7] as message queue service, and OpenStack distributions use RabbitMQ by default [8].

The OpenStack message queue service based on RabbitMQ, which is a centralized approach service, has to cope with performance degradation problem as all the information is saved in the message queue of the controller node; additionally, the requests for checking, exchanging, and coordinating information related to the operation and status of the components are present in the message queue server of the controller node.

Blockchain [9], a decentralization approach, can be used to improve the performance speed by distributing requests for checking, exchanging and coordination that comprise the OpenStack message queue service, while reducing the security risk with the data duplication approach [10]. However, this approach has difficulty processing the information that is updated in real time; this is because of the transactions problem [11] that occurs due to the overhead of the consensus algorithm of the blockchain technology.

The OpenStack message queue service based on a Hybrid decentralized Practical byzantine fault tolerance Blockchain Framework (HdPBF) with a two-step verification approach is proposed in order to improve the processing speed of the OpenStack message queue service, by distributing requests using blockchain and reducing the security risks with the data duplication approach. The practical byzantine fault tolerance (PBFT) Blockchain Framework is based on the PBFT algorithm [12], which is an asynchronous [13] algorithm among the consensus algorithms of the blockchain approach, to efficiently process transactions and ensure reliability by double-checking the information with the message queue server. Despite the quick processing speed of the PBFT algorithm, delay in results may occur due to large volumes of requests and delay in real-time syncing of the blockchain peer.

The proposed scheme can ensure reliability of the message queue service, which is a basic component of OpenStack, through a two-step verification approach, which is a combination of the centralization and blockchain- based decentralization approach. The first step is performing the decentralization approach query; this query checks the information with the HdPBF peer, and in case there is no information, centralization query is performed to check the information with the message queue server as the second step. This method improves the performance by distributing requests for the message queue information to each node through the blockchain based decentralization approach and reduces the security risk associated with the message queue information by saving the message queue information into each node.

OpenStack message queue service

The concept of the message queue operation, which is a centralization approach service, is shown in Fig. 1 [14].

The publisher creates exchanges of Direct, Topic or Fanout, and the consumer creates individual queues for use, which are binding to the exchange.

The structure of the OpenStack message queue service and the OpenStack services based on the RabbitMQ, which is the default message queue service of OpenStack distributions, is shown in Fig. 2 [15].

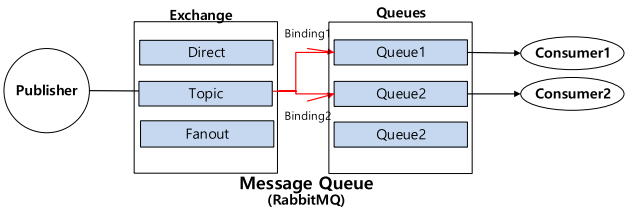

Fig. 1 Concept of message queue

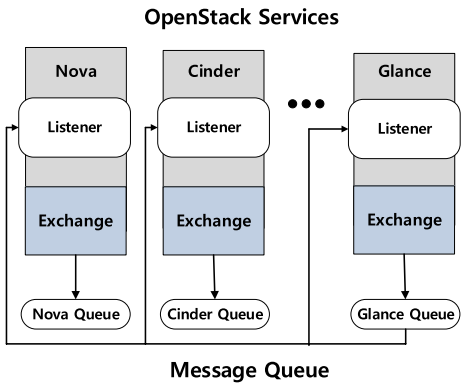

Fig. 2 OpenStack service with message queue

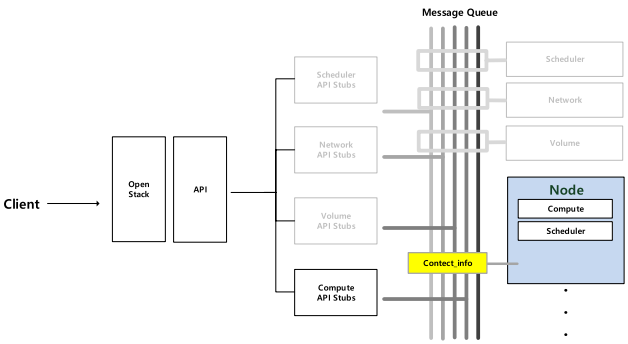

Fig. 3 OpenStack architecture with RabbitMQ based message queue

Services such as Nova, Cinder, Glance etc., which are OpenStack services, check, exchange and coordinate the information related to operation and status of services through the message queue associated with each service.

The entire composition and flow of the OpenStack services and message queue service based on the RabbitMQ is shown in Fig. 3 [16].

If a request from a client using the OpenStack was generated, after the process of authorization of the OpenStack is completed, the resource of nodes through the API of each service is used, and each service and node checks, exchanges, and coordinates information related to the operation and status through the message queue [17].

One of primary operating procedures of Nova, which is a core component of the OpenStack service, is the message queue service based on the RabbitMQ of the OpenStack, with an example that an OpenStack user requests the vnc-console of the instance that is already created, is shown in Fig. 4.

The operating procedure of the message queue service based on the RabbitMQ of the OpenStack in Fig. 4 is described below in steps 1 to 7.

1. A client requests the vnc-console of the instance.
2. The nova-api makes the RPC call through the nova-rpcapi.
3. The nova-rpcapi sends the RPC call to execute the get-vnc-console at the nova-compute that has the requested instance and becomes a block status until the return value of the connect_info is received.
4. The RPC call message of the nova-rpcapi is delivered to the nova-compute through the RabbitMQ.
5. The nova-compute returns the result of execution through the RabbitMQ.
6. The nova-rpcapi returns the connect_info to the nova-api.
7. The nova-api returns the connect_info to the client.

When a user issues a request to the vnc-console of the instance, the nova-api makes a request to the nova-rpcapi for the RPC call, and the nova-rpcapi issues the get-vnc-console call message into the message queue, and waits for the result. The nova-compute returns the connect_info of the message queue, which is the result of executing the get-vnc-console to the nova-rpcapi, and the nova-api returns the connect_info to the client.

Based on the operating procedure shown in Fig. 4, requests for checking, exchanging and coordinating the operating related information and the status information of services through the message queue are present on the message queue server, which is a centralized approach service. A performance degradation problem is experienced in case a large volume of requests are generated.

HdPBF for OpenStack message queue

PBFT Blockchain Framework

The Blockchain Framework in this paper divided version 0.6 of the Hyperledger Fabric based on PBFT and upgraded the RocksDB [18], which is the backend to the version 5.17.2, and realized the PBFT Blockchain Framework that upgraded Go [19], which is

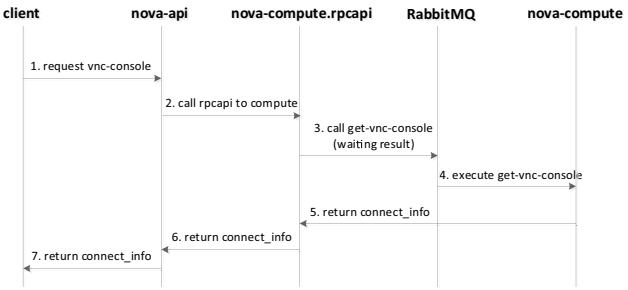

Fig. 4 Operating procedure of RabbitMQ based message queue service

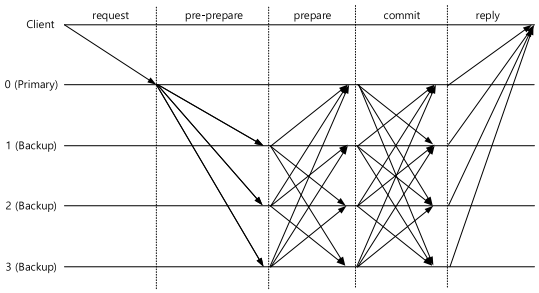

Fig. 5 PBFT Algorithm based Block Create Procedure

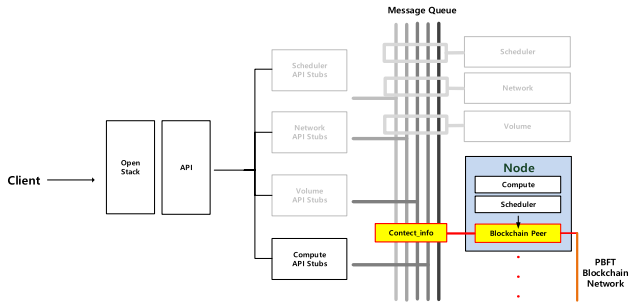

Fig. 6 OpenStack architecture with HdPBF based message queue

the chaincode programming language, to the version 1.11.1. The realized PBFT algorithm-based Block Create procedure is as shown in Fig. 5.

The peer of the PBFT Blockchain Framework is divided into one leader that is elected from the validating peers and non-validating peers, and the validating network follows a procedure similar to the block creation based on the existing PBFT Algorithm comprising validating peers and a leader that participate in the agreement process of Fig. 5, while the non-validating peer is excluded from the agreement process [20].

HdPBF for OpenStack message queue

The entire structure and flow of the OpenStack service and the message queue service containing the HdPBF realized on the basis of the PBFT Blockchain Framework is shown in Fig. 6.

Crucial Information for operation within the message queue is saved into the HdPBF peer, and each node that is the individual HdPBF peer synchronizes the saved information with the HdPBF peers. The detailed structure and flow is shown in Fig. 7.

In case of a large volume of requests for information, synchronization between HdPBF peers may be delayed due to overhead based on the consensus algorithm of the blockchain approach. In such cases, the information is checked by local query to the HdPBF peer as the first step, and in case there is no result, the information is checked by query to the centralized message queue server as the second step. The same reliability as the one from the existing message queue service can be ensured through this two-step verification approach.

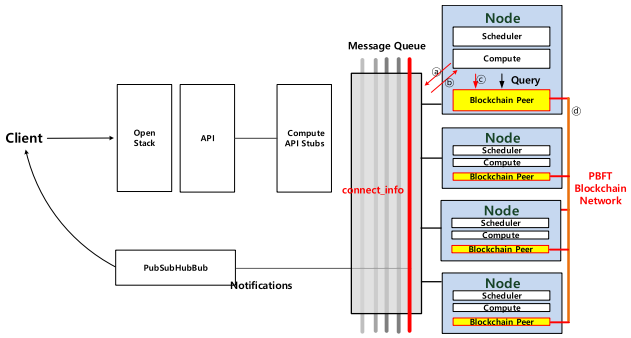

Fig. 7 OpenStack architecture with HdPBF based message queue details

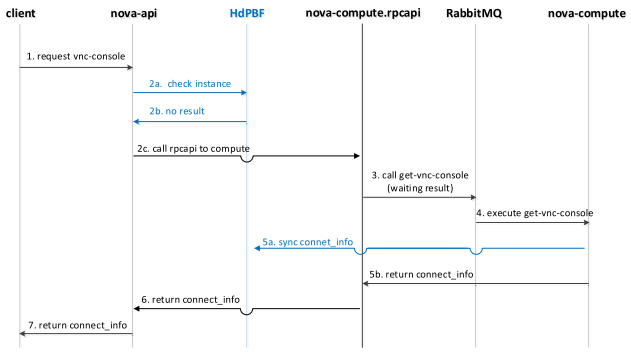

Fig. 8 Operating procedure of the two-step verification of HdPBF based message queue service for synchronizing each HdPBF peer

In case an OpenStack client sends a request initially, the information that is not synchronized with HdPBF peer comprising HdPBF is created in the message queue of the OpenStack. For the information created only in the OpenStack message queue and not synchronized, the HdPBF ensures reliability by performing the two-step verification. The HdPBF peers are synchronized for the corresponding information. The operating procedure of the OpenStack message queue service, which comprises HdPBF is shown in Fig. 8. As one of principal operating procedures of Nova, the two-step verification is executed, with an example of the case that a user of the OpenStack requests the vnc-console from the instance of the information that is already created initially and synchronization between HdPBF peers is completed.

The operating procedure of the OpenStack message queue service comprising HdPBF that executes the two-step verification shown in Fig. 8 is described below from steps 1 to 7.

1. A client requests the vnc-console of the instance.
2a. The nova-api checks if there is connect_info of the instance at HdPBF.
2b. In case of no returned information, 2c is executed.
2c. The information on connect_info is requested to the nova-compute through the nova-rpcapi, which is the RPC API.

3. The nova-rpcapi sends the RPC call to execute the get-vnc-console to the nova-compute that has the requested instance and assumes a block status until the connect_info is returned.
4. The RPC call message of the nova-rpcapi is delivered to the nova-compute through the RabbitMQ.

5a. The nova-compute synchronizes the result of execution through HdPBF.

5b. The nova-compute returns the result of execution through the RabbitMQ.

6. The nova-rpcapi returns the connect_info to API.
7. The nova-api returns the connect_info to the client.

In the case of a first time request sent by an OpenStack client, the operating procedure of the OpenStack message queue service comprising HdPBF is shown in Fig. 9, with an example of the case in which connect_info is created based on the initial call made by a user of the OpenStack, on the condition that the information between HdPBF peers is synchronized after completing the procedure in Fig. 8 and the vnc-console of the instance that is already created is requested again.

The operating procedure of the OpenStack message queue service comprising HdPBF shown in Fig. 9, in which the nova-api checks the information with itself as the HdPBF peer based on the request of a client is described below in steps 1 to 4.

1. A client submits a request to the vnc-console of the instance.
2. The nova-api checks if the connect_info of the instance at HdPBF exists.
3. If the information on the connect_info exists, it reads the connect_info.
4. The nova-api returns the connect_info to the client.

In cases of the connect_info, which is the core information for operation of Nova, if the information is created upon initial execution, it performs the two-step verification as shown in Fig. 8; synchronization between HdPBF peers is made, and only the 1 step Query in Fig. 9 is executed after synchronization.

The two-step verification is executed only when an initial request for the information is created at the message queue of the OpenStack service. Subsequently, only the first step query is executed, and as further repetitive requests for the identical instance are made, there are fewer cases for the two-step verification to be carried out.

The performance is maximized by minimizing the execution of query to the centralized message queue server through the network by a local query to the HdPBF peer. To this end, synchronization of the core information of the OpenStack message queue is performed as the first step. To ensure the authenticity of the information comparable

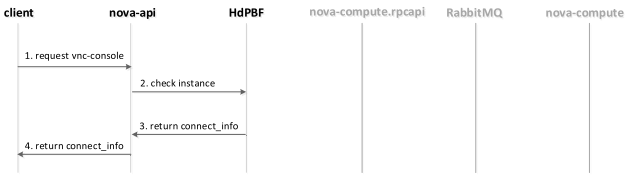

Fig. 9 Operating procedure of HdPBF for synchronized information

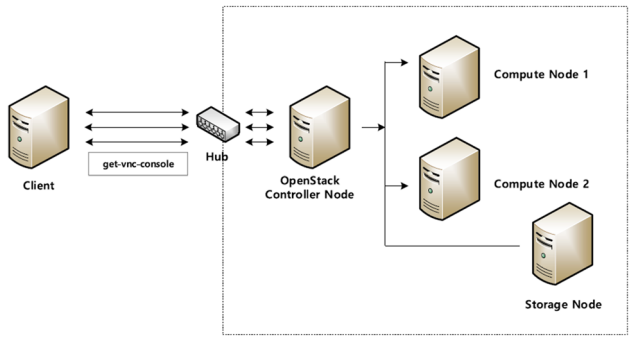

Fig. 10 Diagram of the experimental setup

Table 1 Hardware and software characteristics

Criteria	Hardware		OS	EA
	CPU	RAM (GB)		
Controller node	Intel Xenon 6130, 2.10 GHz	32	Ubuntu 18.04 (OpenStack Rocky)	1
Compute Node	Intel Xenon 6130, 2.10 GHz	64		2
Storage	Intel Xenon E5-2609, 1.7 GHz	32		1
Client	Intel Core i5 2.3 GHz	16	MacOS Mojave	1

to the message queue service of the basic composition of the OpenStack, a query is performed on the centralized message queue server as the second step only for the information that may be generated due to overhead of the consensus algorithm, and it is updated in real time but not yet synched with HdPBF peer. The security risk associated with the message queue information can be reduced as the core information for operation of Nova at the message queue is synched with each node that is the HdPBF peer and identically saved in each Node with the data duplication approach.

Approach and performance evaluation

Environment of experiment

As shown in Fig. 10, the set up comprising controller node and compute node, which are basic components of the OpenStack, similar to the actual environment and the storage node, was set up to create the instance to be used for the experiment.

The hardware and software composition of the experiment in Fig. 10 are shown in Table 1.

If a client requests the vnc-console from the instance, the controller node requests the get-vnc-console from the corresponding compute node, and the connect_info is created at the message queue, if the get-vnc-console is called initially on the instance that is already created. After that, every occurrence of calling the get-vnc-console, the connect_info from the message queue is brought and used.

After creating 6 instances in advance in a method similar to the actual environment, the initial vnc-console request is made in sequence for each instance, and the get-vnc-console is called 6 times for one instance, then the call is made 5 times for the first instance and a new instance followed by another 5-time call for the existing 2 instances

and a new instance, then another 5-time call for the existing 3 instances and a new instance, followed by another 5-time call for the existing 4 instances and a new instance and then lastly, the get-vnc-console is called 4 times on all the 6 instances, which creates a total of 100 call requests such that synchronization between HdPBF peers due to the creation of the connect_info can be made in sequence in the process of the experiment. The shell script created for the experiment is shown in Fig. 11.

In the configuration of the OpenStack message queue based on RabbitMQ of the existing OpenStack, the results of verifying the response time by executing the script of Fig. 11 defining the environment of an experiment is shown in Fig. 12.

The average response time for the experiment performed 100 times is 11.0026 ms.

In the configuration of the HdPBF integrated OpenStack message queue based on the two-step verification approach, the results of verifying the response time by executing the script in Fig. 11 defining the environment of an experiment is shown in Fig. 13.

The average response time for experiment performed 100 times is 5.8539 ms.

The comparison of response times from the configuration of the OpenStack message queue based on the RabbitMQ of the existing OpenStack with the configuration of the

```
# Script for test
#
for i in $(seq 0 5)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
done

for i in $(seq 0 4)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
    nova get-vnc-console 3e281189-c5e7-4f67-ba6d-f505e0256b69 novnc
done

for i in $(seq 0 4)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
    nova get-vnc-console 3e281189-c5e7-4f67-ba6d-f505e0256b69 novnc
    nova get-vnc-console 2801c027-6bd2-4982-a2f9-5981078a53ac novnc
done

for i in $(seq 0 4)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
    nova get-vnc-console 3e281189-c5e7-4f67-ba6d-f505e0256b69 novnc
    nova get-vnc-console 2801c027-6bd2-4982-a2f9-5981078a53ac novnc
    nova get-vnc-console b8f5dd13-c71c-4c34-b0db-c0fd88283757 novnc
done

for i in $(seq 0 4)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
    nova get-vnc-console 3e281189-c5e7-4f67-ba6d-f505e0256b69 novnc
    nova get-vnc-console 2801c027-6bd2-4982-a2f9-5981078a53ac novnc
    nova get-vnc-console b8f5dd13-c71c-4c34-b0db-c0fd88283757 novnc
    nova get-vnc-console 9419e089-b072-4c4a-a7b8-9814fd462750 novnc
done

for i in $(seq 0 3)
do
    nova get-vnc-console 3cecae76-0bae-4831-9ebc-db73700fffde novnc
    nova get-vnc-console 3e281189-c5e7-4f67-ba6d-f505e0256b69 novnc
    nova get-vnc-console 2801c027-6bd2-4982-a2f9-5981078a53ac novnc
    nova get-vnc-console b8f5dd13-c71c-4c34-b0db-c0fd88283757 novnc
    nova get-vnc-console 9419e089-b072-4c4a-a7b8-9814fd462750 novnc
    nova get-vnc-console 9419e089-b072-4c4a-a7b8-9814fd462750 novnc
done
```

Fig. 11 Shell script for the experiment

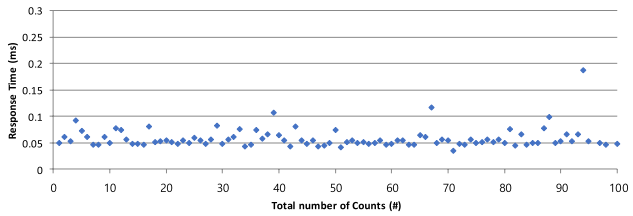

Fig. 12 Response time of RabbitMQ based OpenStack message queue service

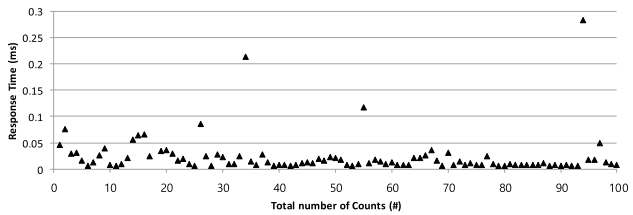

Fig. 13 Response time of HdPBF based OpenStack message queue service

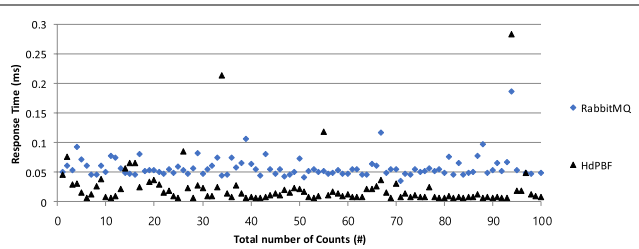

Fig. 14 Comparison of response times between RabbitMQ and HdPBF

Table 2 Comparison of response times between RabbitMQ and HdPBF

	Average (ms)	Total (s)
RabbitMQ	110.026	11.00264
HdPBF	58.539	5.85394

HdPBF integrated OpenStack message queue based on the two-step verification approach is shown in Fig. 14.

The results of the comparison in Fig. 14 are outlined in Table 2.

Based on the results of the experiment, it was confirmed that the average response time of the HdPBF integrated OpenStack message queue service based on the two-step verification approach has reduced by approximately 46.75% when compared to the OpenStack message queue service based on the existing RabbitMQ.

Conclusions

The OpenStack message queue service based on the RabbitMQ, which is a basic component of the OpenStack, is a centralized approach where all requests are concentrated in the centralized message queue server.

It was validated through the experiment that the OpenStack Message queue service integrated with the proposed HdPBF ensures the same reliability as the message queue service with the basic configuration of the OpenStack through the two-step verification approach of performing the decentralization approach query, followed by checking the information at the HdPBF peer, as the first step and performing the centralization query, followed by checking the information with the message queue server, as the second step, results in performance improvement of approximately 46.75% through the blockchain-based decentralization approach distributing the requests. It also reduces the security risk for the message queue information, as the message queue information is synched with all the nodes that are each HdPBF peer and the information is saved in all the nodes identically with the data duplication approach.

Acknowledgements

This research was supported by the Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2018-2018-0-01419) supervised by the Institute for Information & communications Technology Promotion (IITP).

Authors' contributions

YJ was a major contributor in writing the manuscript as a 1st Author and Corresponding Author. JH was a Co-Corresponding Author. Both authors read and approved the final manuscript.

Funding

The Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2018-2018-0-01419).

Availability of data and materials

The datasets generated during and/or analyzed during the current study are available from the corresponding author on reasonable request.

Competing interests

The authors declare that they have no competing interests.

Received: 16 December 2019 Accepted: 26 June 2020
Published online: 15 July 2020

References

1. Nasim R, Kassler AJ (2014) Deploying openstack: Virtual infrastructure or dedicated hardware. In: 2014 IEEE 38th international computer software and applications conference workshops (COMPSACW), July 2014, pp 84–89
2. Rosado T, Bernardino J (2014) An overview of OpenStack architecture. In: Proceedings of the 18th international database engineering & applications symposium, Porto, Portugal, July 2014
3. Message queuing. https://docs.openstack.org/security-guide/messaging.html. Accessed 17 Nov 2019
4. Beloglazov A, Piraghaj SF, Alrokayan M, Buyya R (2012) Deploying OpenStack on CentOS using the KVM hypervisor and GlusterFS distributed file system." Technical Report CLOUDS-TR-2012-3, Cloud Computing and Distributed Systems Laboratory, The University of Melbourne, Aug 2012
5. Videla A, Williams JJW (2012) RabbitMQ in action: distributed messaging for everyone. Manning, Shelter Island
6. Apache Qpid. http://qpid.apache.org. Accessed 17 Nov 2019
7. Hintjens P (2013) ZeroMQ. Messaging for many applications. Sebastopol, O'Reilly Media
8. Documentation: Table of Contents. https://www.rabbitmq.com/documentation.html. Accessed 17 Nov 2019
9. De Filippi P (2016) The interplay between decentralization and privacy: the case of blockchain technologies. J Peer Prod 71:19
10. Gupta AK, Ostner K (2008) Database backup system using data and user-defined routines replicators for maintaining a copy of database on a secondary server. U.S. Patent 7,383.293 B2, Jun. 2008
11. Zheng Z, Xie S, Dai H, Chen X, Wang H (2017). An overview of blockchain technology: architecture, consensus, and future trends. In 2017 IEEE international congress on big data (BigData Congress). IEEE, New York, pp 557–564
12. Castro M, Liskov B (1999) Practical Byzantine fault tolerance. OSDI 99(1999):173–186

13. Duan S, Reiter MK, Zhang H (2018) BEAT: asynchronous BFT made practical. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM, New York, pp. 2028–2041
14. AMQP 0-9-1 Model explained https://www.rabbitmq.com/tutorials/amqp-concepts.html. Accessed 17 Nov 2019
15. OpenStack Compute (Nova) https://docs.openstack.org/nova/latest/. Accessed 17 Nov 2019
16. AMQP and Nova https://docs.openstack.org/nova/queens/reference/rpc.html. Accessed 17 Nov 2019
17. Sze WK, Srivastava A, Sekar R (2016) Hardening OpenStack cloud platforms against compute node compromises. In: Proceedings of the 11th ACM on Asia conference on computer and communications security, ASIA CCS'16. ACM, New York, pp. 341–352
18. Yang F, Dou K, Chen S, Hou M, Kang JU, Cho S (2015) Optimizing NoSQL DB on flash: a case study of RocksDB. In: 2015 IEEE 12th international conference on ubiquitous intelligence and computing and 2015 IEEE 12th international conference on autonomic and trusted computing and 2015 IEEE 15th international conference on scalable computing and communications and its associated workshops (UIC-ATC-ScalCom). IEEE, New York, pp. 1062–1069
19. Cachin C (2016) Architecture of the hyperledger blockchain fabric. In: Proceedings the workshop on distributed cryptocurrencies and consensus ledgers. July. 2016
20. Sukhwani H, Martínez JM, Chang X, Trivedi KS, Rindos A (2017) Performance modeling of pbft consensus process for permissioned blockchain network (hyperledger fabric). In: 2017 IEEE 36th symposium on reliable distributed systems (SRDS). IEEE, New York, pp. 253–255

Publisher's Note