Theoretical and Applied Informatics
ISSN 1896-5334
Vol. 26 (2014), no. 1, 2
pp. 69 - 85

Highly available and fault-tolerant architecture guidelines for clustered middleware servers

Maciej ROSTAŃSKI[a], Krzysztof GROCHLA[b], Aleksander SEMAN[b]

[a] University of Dabrowa Gornicza
ul. Cieplaka 1C, 41-300 Dabrowa Gornicza, Poland
mrostanski@wsb.edu.pl

[b] Proximetry Poland, Sp. z o.o.
Al. Rozdzienskiego 91, Katowice, Poland
{kgrochla,aseman}@proximetry.pl

Abstract: The paper presents the result of an evaluation of the performance of different message broker system configurations, which lead to the construction of the specific architecture guidelines for such systems. The examples are provided for an exemplary middleware messaging server software - RabbitMQ, set in high availability - enabling and redundant configurations. Rabbit MQ is a message queuing system realizing the middleware for distributed systems that implements the Advanced Message Queuing Protocol. The scalability and high availability design issues are discussed and the possible cluster topologies and their impact is presented. Since HA and performance scalability requirements are in conflict, scenarios for using clustered RabbitMQ nodes and mirrored queues are interesting and have to be considered with specific workloads and requirements in mind. The results of performance measurements for some topologies are also reported in this article.

Keywords: high availability, fault tolerance, middleware messaging, RabbitMQ, clustered systems architecture.

1. Introduction

Due to the cloud development, the programming paradigms have shifted. The applications, devices or appliances form the distributed parts of the whole solution. As they serve more and more users, they need to connect and scale. The components of a larger application need constant connection between themselves, or to user devices and data. The messaging task needs to be supported by the system internally, or using external frameworks or systems. As Richardson writes in [1]: "Future applications (..) [will be] always on, cloud hosted, and accessible anywhere. Input and processing are continuous and automatic, and deliver a filtered stream of information that the user wants, as it happens."

The middleware layer, often referred to as a 'glue' between different system components, allows communication between them. Message queuing, also called message-oriented middleware, is in fact an architectural pattern. It is based on an implementation of a system part called the *message broker*, an intermediary program which performs message validation, message transformation and message routing functions. The message broker provides a common infrastructure for interactions between elements of the distributed systems, which interact by sending or receiving messages, and is a recent alternative for distributed interaction between the components, or entities, of an information processing system.

In this paper we describe the design considerations of scalability and high availability (HA) improving architectures, using RabbitMQ software, an open source message broker and queuing server that is becoming more and more popular as a middleware. The balance between HA and scalability is challenging because of contrary requirements - the scalability and performance-optimisation mechanisms are in principle hindered by high availability or fault tolerance solutions, which prefer stability and durability over performance. The papers's objective is to provide guidelines for such design, based on presented performance results and previous work. For this purpose, this paper presents possible configuration scenarios for a RabbitMQ cluster of servers, which combine scalability with high availability / fault tolerance (HA/FT) requirements. RabbitMQ is therefore described as middleware solution and clustering options are presented, as well as HA possibilities. The scenarios were implemented for test-field studies, whose results are presented. Most of the available literature or reports such as [2], [3] or [4] concentrates on the scalability issues and performance results, or, from a different perspective, strictly on high availability / fault-tolerance solutions for queuing [1]. This paper aims to bring a novelty in discussing solutions that combine both requirements, as it is a most probable industry scenario, and there is a lack of information on the results of similar experimentations.

The paper is organized as follows: the design requirements for middleware system are presented and briefly explained – specifically, scalability and high availability concerns are discussed. The next part includes a short summary of RabbitMQ, the message broker used in research. The main part includes message broker configuration scenarios for scalability and high availability; the experimental results of constructed systems are presented for comparison. Finally, conclusions are revealed.

2. Messaging middleware with RabbitMQ as an example

Message queuing is thoroughly described, for example, in [5], [6] and [7]. It is often based on a publish/subscribe-like interaction [7]. The message queuing is an alternative to Classifications, which are complementary to the publish/subscribe model of a distributed information system [8]. Classifications involve techniques such as message passing, shared spaces or remote invocations and constitute solutions to the middleware layer challenges. Middleware systems are also subject to numerous

studies, concentrating on networking and concurrent design. There is a concept of using patterns in overall software architecture, with [9] as a main example, or for security related applications ([10], [11]).

From the architect's perspective, message-oriented middleware can be seen as a (1) queuing system, where messages are concurrently pulled by consumers, as well as (2) subscription-based exchange solution, allowing groups of consumers to subscribe to groups of publishers, resulting in a communication network or platform, or a message bus [7]. Such bus or queuing system has to be able to scale in terms of geographical distance as well as in terms of devices or applications served. Quoting Jones et al. [4], "the distribution of information sent from the publishers to the hub to be distributed to the necessary subscribers allows for applications to run while relying on data from other locations, wherever they may be."

RabbitMQ is an open source message broker and queuing server that can be used to let disparate applications share data via a common protocol or to simply queue jobs for processing by distributed workers. RabbitMQ middleware supports many messaging protocols [12], among which the most important are STOMP: Streaming Text Oriented Messaging Protocol [13] and AMQP: Advanced Messaging Queuing Protocol [14].

For the purposes of this paper the AMQP-defined messaging architecture was used. The queues are most important concept of message broker structure; every message received by the RabbitMQ is always placed in a queue, which in turn can be stored in memory (memory-based) or on a disk (disk-based). Second important element of the RabbitMQ is *exchange* - the delivery service for messages. The exchange used by a publish operation determines if the delivery will be either *direct* or *publish-and-subscribe*. A client chooses the exchange used to deliver each message as it is published. The exchange looks at the information in the headers of a message and selects where they should be transferred to[15].

2.1. Specific system design requirements for middleware

2.1.1. Resiliency

In order to be resilient (which means to be able to deal with internal failures), the system needs to implement some forms of high availability (HA) or fault tolerance (FT). In general, HA and FT systems are designed with two different design principles in mind. Given the availability (A) formula (eq. 1),

$$A = \frac{MTBF}{MTBF + MTTR} \tag{1}$$

HA aims to minimize downtime and IT service disruption; so the common goal in HA is to increase *Mean Time Between Failure* (MTBF) and decrease *Mean Time to Repair* (MTTR). HA solutions are in principle designed to have a high level of service uptime and may feature many elements, e.g: system management, live replacement (hot-swap), component redundancy and failover mechanisms. To avoid single points of failure in the system can be difficult, because demands on such systems include not only ensuring the availability of important data, but also efficient resource sharing of the relatively expensive components.

Contrary to HA, which implies a service level in which both planned and unplanned outages do not exceed a small stated value [16], fault-tolerant (FT) systems tend to implement as much component redundancy and mirroring techniques as possible, in order to eliminate system failures completely (this is of course from client's perspective, in fact introducing redundant components will make component failures occur faster) [17], [18]. But FT has its problems; the performance degradation is another concern. As an example, let's discuss mirroring a single server. Besides handling all of the file transfer work for network users, the primary server may have to process additional I/O as it passes information along to the mirror server. This can also add substantial processor overhead if system usage is heavy. In effect, RAM, CPU and network performance is degraded.

2.1.2. Scalability

Scalability is an architectural characteristic, which can be defined as a capability to cope and perform under an increased or expanding workload. A system that scales well will be able to maintain or even increase its level of performance or efficiency when tested by larger operational demands. In terms of message-queuing, or even publisher/consumer exchange system, this would mean the possibility of increasing processing speed or message throughput, user capacity, etc.

2.1.3. Combination

A typical solution that requires either resiliency or scalability or both, involves clustering - *symmetrical* (all nodes have similar capabilities) or *asymmetrical* (nodes have different possibilities and inventory). Clustering in this context can be described as the use of two or more systems loosely coupled to provide system level redundancy – or provide more resources for operation. Because such systems are not directly coupled, they utilize standard network connections to communicate failovers. Typically, there is a middleware software solution to provide a failover mechanism between the two systems. But this middleware has to be protected with HA in mind as well.

2.2. Scalable and fault-tolerant middleware

For message broker, both HA and FT solutions were considered:

a) HA (Active/Passive solution)

in which the downtime of message broker (MB) service is expected in case of planned or unplanned unavailability of primary server. Queues and messages have to be persistent (disk-based), and message broker can be restarted elsewhere in the system. It is possible to base such solution on virtualization, where MB running host can be virtualized and rely on hypervisor built-in HA mechanism. This would cause hypervisor to run another instance of virtual machine (VM) in case of a failure of primary MB guest or even virtualization host. Another active/passive solution is to deploy clustering HA solution like pacemaker [20] in order to manage message broker and restart it (or migrate) when necessary, using available resources.

b) FT (Active/Active solution)

means that the planned or unplanned downtime of message broker does not have any effect on queuing system. Typically it is implemented by MB leveraging clustering mechanism built-in RabbitMQ, which is developed strictly for such situations, and replicates queues on every RabbitMQ node in the cluster. RabbitMQ nodes failure monitoring, and IP load-balancing techniques are explained further in detail. Active/active solution can also be based on virtualization, where MB running host can be virtualized and, for example, marked as FT-demanding in VMware vCenter virtualisation environment. This would create a VM mirror image called "replica", updated in real-time, ready to be run in case of a failure of primary MB host.

Message broker, being one of the most important components of a distributed system, should be as fault-tolerant as possible. That means that the typical configuration for high availability (as described in 2.1.1) is not the best option. If the service is being restarted and prepared for operation restarting message broker on another node in case of failure, it would introduce a timeout span, but, what is worse, the message queue of failed message broker would be lost entirely.

The second solution described in option b) is a typical Active/Active topology and is recommended as more reliable and scalable at the same time. The virtualization variant was considered but not implemented, because it would introduce additional conditions and variables to the experiments and is a subject for another study. This paper's research is thus based on a cluster of RabbitMQ message brokers and its characteristics.

2.3. RabbitMQ cluster setup and operation

The clustering built into RabbitMQ was designed with two goals in mind:

- allowing consumers and producers to keep running in the event of node failure,
- linearly scaling messaging throughput by adding more nodes [1].

With clustering, a client can connect as normal to any node within a cluster. If that node should fail, and the rest of the cluster survives, then the client should notice the closed connection, and should be able to reconnect to some surviving member of the cluster, as given in [12].

The design decision that had to be made was an IP addressing of the cluster. As RabbitMQ documentation [12] describes, it is not generally advisable to hardcode node hostnames or IP addresses into client applications: this introduces inflexibility and will require client applications to be edited, recompiled and redeployed should the configuration of the cluster change or the number of nodes in the cluster change. As in general, this aspect of managing the connection to nodes within a cluster is beyond the scope of RabbitMQ itself. RabbitMQ's authors recommend a more abstracted approach, including a dynamic DNS service which has a very short TTL configuration, or a plain TCP load balancer (for example HAproxy [19]), or some sort of mobile IP achieved with pacemaker or similar technologies [20], [21]. For the purpose of this study, HAproxy was chosen as a load balancer between clients and cluster nodes.

Finally, testing environment needs to be monitored for two flow control mechanisms in RabbitMQ that may interfere with fast publishers. Network connections were configured with low speeds and the testing scenarios were implemented with those constraints in mind.

3. Clustering scenarios

The maximization of the systems performance suggests that content of the queues should not be replicated throughout the cluster. The queue owner node has full information about it; other nodes in the cluster only know the queue's metadata and a pointer to the node where the queue actually is stored. This solution allows to limit storage space requirements and increase performance – replicating messages to every node would result in increase of network and disk load for every node, keeping the performance of the cluster the same (or worse) [1]. Regardless where publish is made, message will end up on the queue owner node. This leads to main performance optimization technique: to increase performance for every added node by spreading queues across nodes.

On the contrary to performance-driven requirements for queues, there is a need for queue to be redundant when the main goal is to achieve high availability and fault tolerance. If a queue owner node fails, all of the messages within a queue are gone. An active-active redundancy option is possible; any queue can be mirrored. The mirrored queue is achieved by creating slave copies of the queue on other nodes in the cluster. It can be copied on every node, but the designer is able to specify a subset of nodes in the cluster for a queue to live on.

Both situations are presented on Fig. 1.

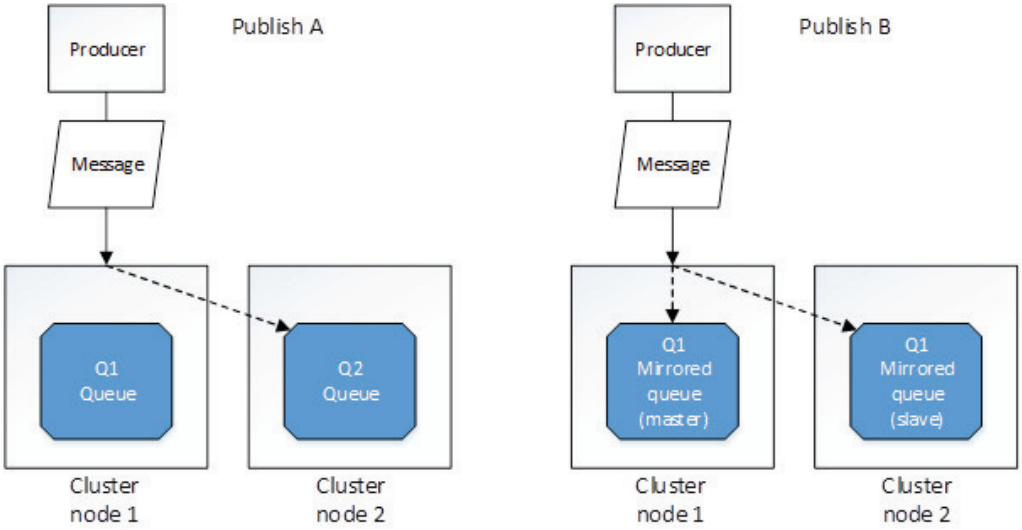

Fig. 1. Publishing to queues: a) on another node, b) to the mirrored queue

The design of the cluster and its queues can support the following:

- Creating fully mirrored queues on every node in order to achieve HA; create very efficient connection between nodes and create RAM nodes for quick distribution of messages,
- Creating spread queues, but configure mirrored queues for at least one master and one slave (allowing for one node failure),
- Creating fully spread queues and do not mirror them, but make them durable instead – all of the nodes are disk based, and in the event of failure, message broker is restarted elsewhere.

Within above listed possibilities, 1) is a scenario for maximum fault-tolerance, 3) is a scenario allowing some downtime for maximum performance (which is HA scenario) and 2) presents a compromise between those two.

3.1. Cluster and queues configuration – preliminary testing

Considering a three-node cluster, one can come up for specific testing scenarios that can provide comparable results for performance assessment [22]. Those results may provide an answer, whether given configuration is useful for a specific real-world scenario [23]. For preliminary testing purposes, and for re-creating typical real-world scenario, following implications were made: a) cluster may include up to three nodes, b) queues are created in the cluster as a single (non-mirrored), fully mirrored, and spread (mirrored to one node) queue, c) all of configuration scenarios are put to the three tests:

- single publishes and consumes: there are single publishers and consumers for both queues,
- balanced conversations: there are three publishers and three consumers for every queue (as many as cluster nodes),
- many conversations: there are some publishers and some consumers.

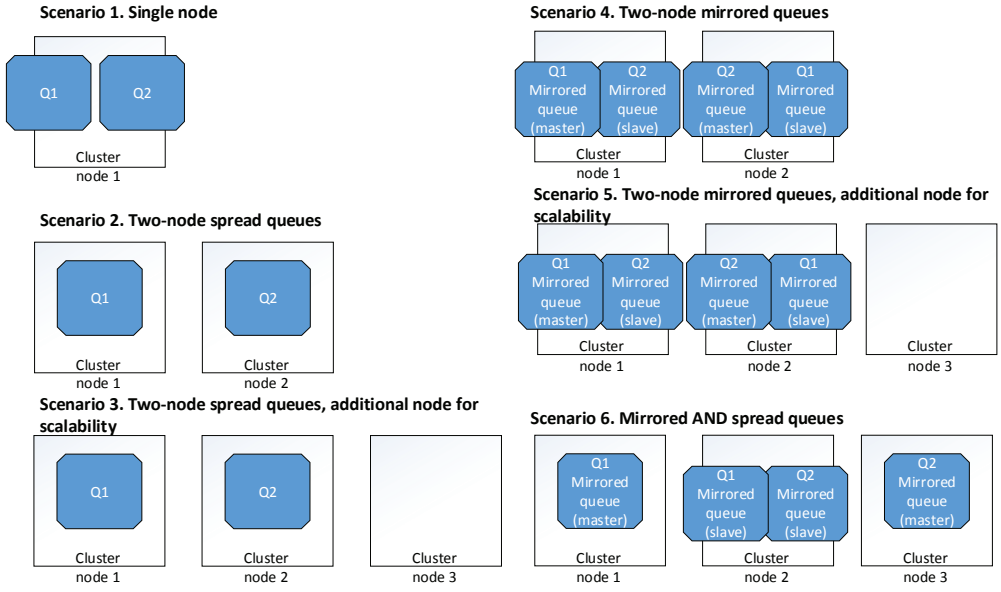

Fig. 2. Performance-driven vs. Fault-Tolerant-driven testing scenarios

Possible configurations for two queues implementation are presented on Fig. 2. On the left (scenario 1,2,3), there is no fault-tolerance; queues are not mirrored and the cluster is configured for performance scalability; on the right, queues are mirrored for resiliency – this is possible only using two nodes minimum; adding the third node creates two possibilities – mirroring queues on two nodes and adding one node for

scalability (scenario 5) or “spreading” mirrored queues on available nodes (scenario 6).

3.2. Preliminary testing results

The equipment used for testing involved identical virtual machines (single core, 4GB RAM, 8GB HDD) put on one hypervisor, which eliminated any possible networking issues. The hypervisor host was equipped with Intel i7 CPU and 32GB RAM. There was no resource overload. The most interesting observations were as follows.

Conclusion A - there is practically no difference between the performance of publishing to single or multiple queues on one node. Scenario 1 is viable and does not introduce any performance problems. This question doesn’t need any more evaluation.

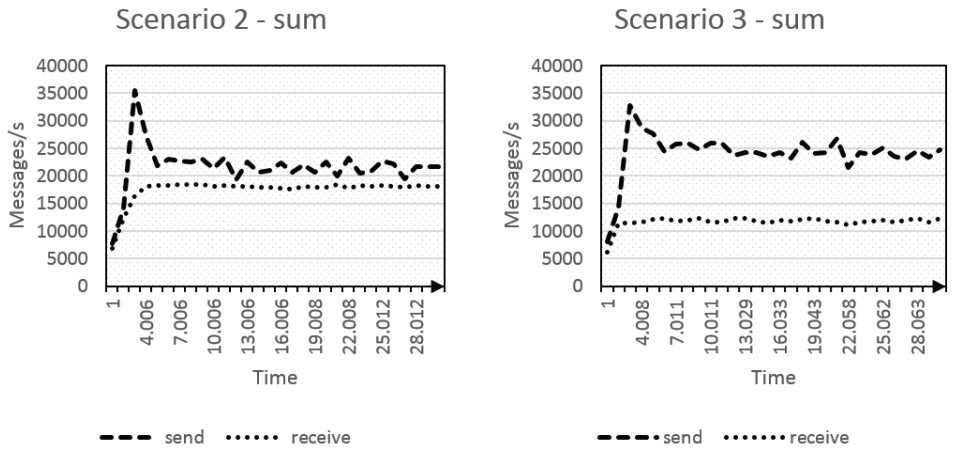

Fig. 3. The comparison of two-queue, one-client-per-queue publishes and consumes using scenario 2 and scenario 3

Conclusion B - the results of scenario 2 and scenario 3 (there is more than one node in the cluster) show significant improvement of performance over scenario 1. Fig. 3 presents exemplary results of single publisher and consumer for both queues, summarized for comparison. These results are expected, however designer has to keep in mind such configuration is not fault-tolerant – if a node fails, the queue is no longer available for publishing or consuming. The difference between scenario 2 and scenario 3 (additional node for scaling) results is interesting and was immediately chosen a subject for another study – adding supplementary node allowed faster publishing, but the consuming rate dropped, as the cluster nodes communication introduced an overhead. In effect, whole system performance was kept on the same level. As this

design could be more appropriate with large number of publishers and consumers, the decision to test multiple scenarios was made.

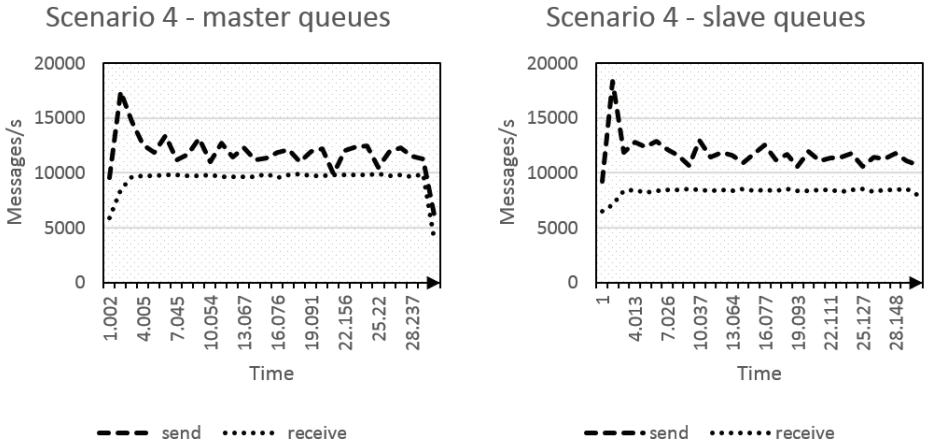

Fig. 4. Mirrored queues performance results example

For the scenarios 4,5,6 (mirrored queues for resiliency), the performance of sending and receiving to mirrored queues is significantly worse. Fig. 4 shows typical results (scenario 4 is shown). The difference between publishing and consuming to the master (owner) node compared to publishing and consuming from the slave node is as expected - publishers are unaffected, but the consumers suffer from intra-cluster traffic (master-to-slave) overhead. Most important results summary is shown on Tab. 1.

Scenario	**Scenario 3 (performance)**	**Scenario 5 (mirrored)**	**Scenario 6 (spread mirrors)**
Average publish rate [msg/s]	33296.59	8553.28	12668.86
Average consume rate [msg/s]	16162.00	5087.00	8231.55

Tab. 1. Summary for most important scenarios (10 publishers, 10 consumers)

Most important observations are, as indicated by multiple tests (ca. 50 re-runs), the performance of single queue drops significantly when this queue is mirrored throughout entire cluster for fault-tolerance. Full mirrored queue is therefore not as good architectural choice as it would seem, especially if there are frequent moments of only one producer active. The performance of *spread* queues is about 10%-20% minimum better than full–mirrored queues on a three-node cluster. There is no

significant difference in RAM / disk node effect on mirroring. Spread queues are stable; performance degradation is however visible when receiving by many clients at once.

3.3. Detailed evaluation

Upon further examination of the preliminary test results, the most important observation was that the results are not always the same:

a) different master-slave queue configuration combinations produced different performance results,
b) even non-mirrored queues had different results as well.

The conclusion was, that this is the impact of load balancing the traffic between the clients and cluster nodes. To be more specific, the message publishing or consuming rate depends whether the client was redirected to the:

- "master" node (the node which is the master for the specific queue being used),
- "slave" node (the node which specific queue is being replicated onto),
- "empty" node (the node which is part of the cluster but the queue resides on other nodes)

If the client is redirected onto "master" or "slave" nodes, the published messages do not need to be communicated to every node in the cluster, which has good effect on performance. Otherwise, message sending/receiving rates drop.

For detailed information on this impact, the cluster was set up with different architectures - using disk-based and RAM-based queues. Every configuration assured that if queue is mirrored, it always resides on at least one disk-based node, and messages are written to disk and can be retrieved even after power failure. For such cluster, queues were tested for performance while load balancer was configured with alternative scenarios:

- Master node not receiving connections from clients (“no master”),
- Master and slave node not receiving connections from clients (“only empties”),
- Queues are "mirrored" to every other node in the cluster (“only slaves”)

The effect of load-balancing is shown on Fig. 5 and Fig. 6. In this test, six clients were sending messages to cluster and were load-balanced on any cluster node. On Fig. 5, the connection to “master” node is being served first, an every other connection is suppressed; this causes the entire exchange to last 20s. On Fig. 6, load balancer omitted “master” node. The message sending rates were much more ‘fair’ for every client. Overall time of sending was 11s.

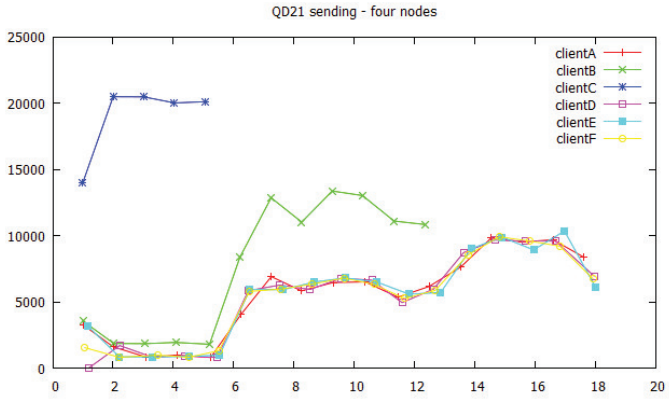

Fig. 5. Performance of six clients sending to cluster, one to “master” node. Overall time: 20s

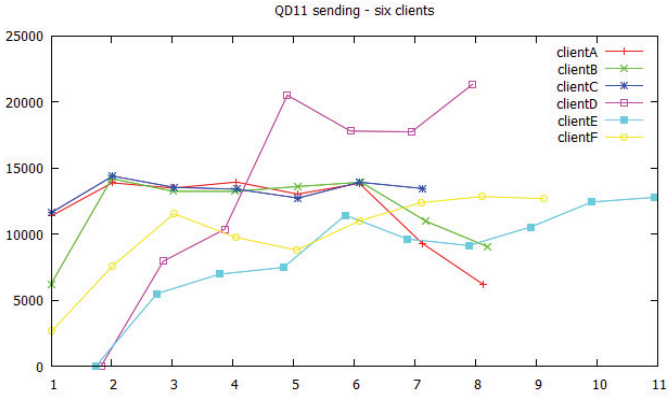

Fig. 6. Performance of six clients sending to cluster, no connection to “master” node. Overall time: 11s

Similar load-balancing tests have shown, that there is unfair balance between clients receiving messages from “slave” and “empty” nodes. The “slave” nodes served clients only after every other connection ended (Fig. 7).

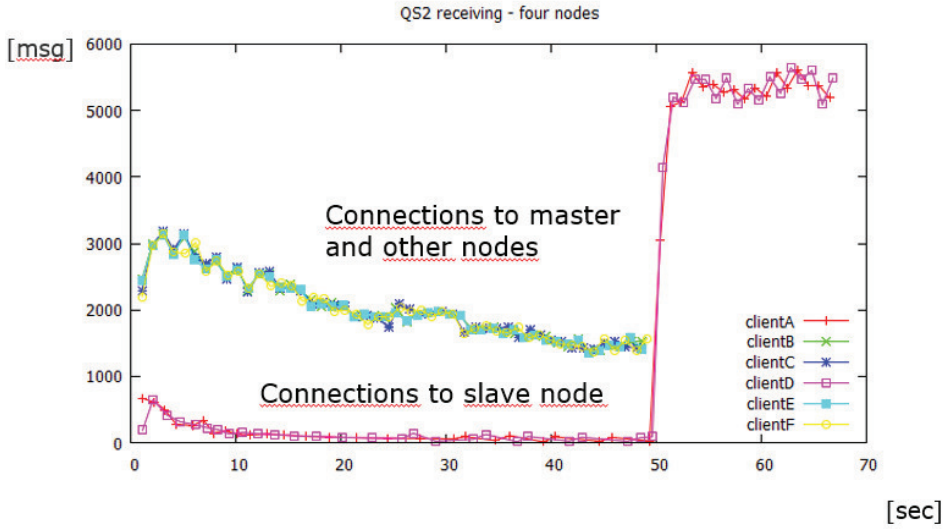

Fig. 7. Performance of six clients sending to cluster of “master”, “slave” and “empty” nodes example.

3.4. Proposed cluster configuration

The extensive testing led to following observations and design proposals:

- During receiving messages, clients connected to “slave” nodes are treated unfair;
- For fair sending, “master” node should not be serving clients;

The most proper cluster configurations should thus utilize layers of nodes:

- Fault Tolerance Layer, that ensures reliability and endurance of messages and queues;
- Scalable Services Layer, that contains easily scalable and configurable nodes.

Proposed architecture is shown on Fig. 8. Two nodes are not servicing any clients and every queue is configured on those two nodes (in “master” and “slave” modes, which can be described as “spread” queues as defined in 2.1), in addition to *n* “empty” nodes of the cluster.

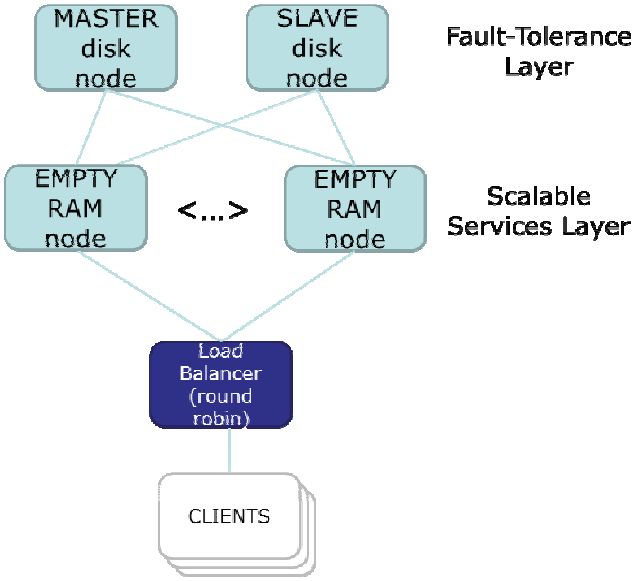

Fig. 8. Proposed two-layer cluster configuration.

4. Conclusions

This paper shows there are many considerations for building clustered middleware and implementing scalable yet fault-tolerant system. Queues need to be distributed evenly, or internal transfers within the cluster will cause performance to drop, especially for receiving clients. There is, however a way to mirror queues asymmetrically, which is shown by experimental results in this paper.

The relevant studies indicate that many more aspects have to be taken under consideration – for example, the disk-based nodes compared to RAM-based nodes performance, or the expected distribution of the clients (publishers and consumers) but results show there is a possibility to create a design principles for specific clients count and message rates requirements and form a two-layered cluster configuration. This is authors' contribution in discussing solutions that combine both requirements, as it is a real industry scenario.

To summarize results, the study shows that while typical single queues on clustered nodes are key to performance, if the requirements include fault-tolerance, performance can still be improved by "spreading" queues to be mirrored only by one more node, as N+1 rule dictates.

Acknowledgement

This work is partially supported by NCBIR INNOTECH Project K2/HI2/21/1 84126/NCB R/13, The Effective Management of Telecommunication Networks Consist of Millions of Devices.

References

[1] A. Videla and J. Williams, *RabbitMQ in action. Distributed messaging for everyone*. Manning, April 2012

[2] M. Altherr, M. Erzberger and S. Maffeis, "iBus - a software bus middleware for the Javaplatform," in: *Proceedings of the International Workshop on Reliable Middleware Systems*, 1999, pp. 43-53.

[3] Salvan, M., *A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo…* [online: http://bit.ly/1b1UGTa], April 2013

[4] B. Jones, S. Luxenberg, D. McGrath, P. Trampert and J. Weldon, "RabbitMQ Performance and Scalability Analysis", project on CS 4284: Systems and Networking Capstone, Virginia Tech 2011

[5] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middleware", in: *Proceedings of the 13th International Symposium on Distributed Computing (DISC99)*, 1999, pp. 1-18

[6] B. Blakeley, H. Harris, and J. Lewis, "*Messaging and Queuing Using the MQI*". McGraw-Hill, New York, NY, 1995.

[7] Eugster et al., "The Many Faces of Publish/Subscribe", in: *ACM Computing Surveys, Vol. 35, No. 2*, June 2003, pp. 114-131.

[8] M. Franklin and S. Zdonik, "A framework for scalable dissemination-based systems" ,in: *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*. ACM Press, New York, NY, 1997, pp. 94-105.

[9] F. Buschmann et al., *Pattern-oriented software architecture: a system of patterns*, John Wiley and Sons, Inc. New York, NY, USA ©1996 ISBN:0-471-95869-7

[10] X. Yuan and E. B. Fernandez, ``Patterns for Business-to-Consumer E-Commerce Applications", accepted for the International Journal of Software Engineering and Applications (IJSEA)

[11] M. VanHilst, E. B. Fernandez and F. Braz, ``A Multidimensional Classification for Users of Security Patterns", in *Journal of Research and Practice in Information Technology, vol. 41, No 2*, May 2009, pp. 87-97

[12] RabbitMQ documentation [online], http://www.rabbitmq.com/documentation.html, accessed 21.01.2014

[13] The Simple Text Oriented Messaging Protocol website [online], http://stomp.github.io/, accessed 20.01.2014

[14] P. Houston, "Building distributed applications with message queuing middleware" (Whitepaper). Available online at http://msdn.microsoft.com/library/en-us/dnmqqc/html/bldappmq.asp, 1998

[15] J. O'Hara, "Toward a Commodity Enterprise Middleware", *ACM Queue 5 (4)*, June 2007, pp. 48-55

[16] M. Rostanski, "High Availability Methods for Routing in Soho Networks", in Kapczynski A., Tkacz E., Rostanski, M.: *Internet - Technical Developments and Applications 2*, Springer 2011, pp. 154-152

[17] P. Buchwald, "The Example of IT System with Fault Tolerance in a Small Business Organization", in: Kapczynski A., Tkacz E., Rostanski, M.: *Internet – Technical Development and Applications 2*, Springer 2012, pp. 179-187

[18] Grzywak A., Buchwald P., Maczka K., Pikiewicz P., Rostanski M.: "Methods for Information Management Systems Resiliency Improvement", in: Rostanski M., Pikiewicz P. (Eds.): Internet in the information society. Insights on the information systems, structures and applications, Academy of Business in Dabrowa Gornicza 2014, ISBN: 978-83-62897-91-9, pp. 49-60

[19] "HAProxy. The Reliable, High Performance TCP/HTTP Load Balancer". Website: http://haproxy.1wt.eu/, accessed: 21.01.2014

[20] "Pacemaker. A scalable High Availability cluster resource manager". Website: http://clusterlabs.org/, accessed: 18.01.2014

[21] Rostański M.: Prywatny klaster wysokiej dostępności przy użyciu systemów Linux i Pacemaker, Informator o ochronie teleinformatycznej CIIPFocus nr 7, Rządowe Centrum Bezpieczeństwa 2014, s.14-16

[22] S. Nowak, M. Nowak and M. Foremski, "New Synchronization Method for the Parallel Simulations of Wireless Networks", in: *11th International Conference, NEW2AN 2011, and 4th Conference on Smart Spaces, ruSMART 2011, St. Petersburg, Russia, August 22-25, 2011. Proceedings, LNCS 6869*, Springer Berlin Heidelberg, pp. 405-415

[23] K. Grochla, L. Naruszewicz, "Testing and Scalability Analysis of Network Management Systems Using Device Emulation", in: Computer Networks, Springer 2012, pp. 91-100

Wzorce wysokodostępnej i odpornej na awarie architektury dla klastra serwerów middleware

Streszczenie

W pracy przedstawiono wyniki oceny wydajności różnych konfiguracji systemów spełniających rolę rozdzielacza wiadomości (Message Broker), które prowadzą do wyznaczenia konkretnych wytycznych architektonicznych dla takich systemów. Przykład zrealizowano przy użyciu przykładowego oprogramowania serwera komunikacyjnego middleware – RabbitMQ, zestawionego w konfiguracji wysokiej dostępności. RabbitMQ jest systemem kolejkowania wiadomości, który realizuje funkcje pośredniczące (ang. middleware) dla systemów rozproszonych, używając do tego zadania zaawansowanych protokołów kolejkowania wiadomości. W artykule omówiono zagadnienia projektowe dotyczące skalowalności i wysokiej dostępności, jak również przedstawiono możliwe topologie klastrów i ich wpływ na zdefiniowane parametry działania. Ponieważ wymagania HA i skalowalność, a zatem wydajność, są w konflikcie, rozpatrywano scenariusze z różnym wykorzystaniem kolejek w pełni redundantnych oraz dublowanych. W artykule przedstawiono wyniki pomiarów wydajności dla niektórych topologii, jak również konkluzje co do drogi do osiągnięcia optymalnej architektury.