Proceedings of the 2014 Federated Conference on
Computer Science and Information Systems pp. 879–884

DOI: 10.15439/2014F48
ACSIS, Vol. 2

Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ

Maciej Rostanski
Academy of Business in Dabrowa Gornicza
ul. Cieplaka 1C, 41-300 Dabrowa Gornicza, Poland
Email: mrostanski@wsb.edu.pl

Krzysztof Grochla, Aleksander Seman
Proximetry Poland, Sp. z o.o.
Al. Rozdzienskiego 91 40-203 Katowice, Poland
Email: {kgrochla, aseman}@proximetry.pl

***Abstract*—The paper presents a performance evaluation of message broker system, Rabbit MQ 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. Since HA and performance scalability requirements are in conflict, scenarios for using clustered RabbitMQ nodes and mirrored queues are presented. The results of performance measurements are reported.**

I. Introduction

MODERN distributed systems have modular architecture. The applications, devices or appliances which are distributed parts of the whole solution, need to connect and scale. The applications need to connect to one another as components of a larger application, or to user devices and data. Nowadays, messaging, understood as an information flow or a network rather than a stack, needs to be supported by the system. As Richardson writes in [21]: ”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. The modern requirement is to overpower the limits of point-to-point communication, and, moreover, to do it in a non-synchronous fashion. This is also referred to as a time- , space- and synchronisation-decoupling [6], and is especially important, given the fact, that the distributed systems now involve thousands of entities, which may be distributed throughout vast geographical distances, and whose behaviour and even location may vary in time. Message queuing also called message-oriented middleware is an architectural pattern. It is based on a message broker, an intermediary program which realizes message validation, message transformation and message routing functions. Message broker provides common infrastructure for interactions between elements of the distributed systems, which interact by sending or receiving messages. It is a recent alternative for distributed interaction between components, entities of an information processing system. Message queuing is thoroughly described, for example, in [2], [3] and [6]. It is often based on a publish/subscribe-like interaction [6]. The message queuing is an alternative to Classifications, which are complementary to the publish/subscribe model of a distributed information system [9]. 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 [5] as a main example, or for security related applications ([7], [8]).

This work was supported by Proximetry Poland

This paper describes design considerations of scalability and high availability (HA) improving solutions 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.

This paper presents possible configuration scenarios for a RabbitMQ cluster of servers, which combine scalability with high availability / fault tolerance (HA/FT) requirements. For this purpose, RabbitMQ is described as middleware 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 [1], [19] or [13] concentrates on the scalability issues and performance results, or, from a different perspective, strictly on high availability / fault-tolerance solutions for queuing [21]. This paper aims to bring a novelty in discussing solutions that combine both requirements, as it is a probable industry scenario.

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.

II. RabbitMQ as a middleware

From designer'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 [6]. 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. [13], "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 [17], among which the most important are STOMP: Streaming Text Oriented Messaging Protocol [20] and AMQP: Advanced Messaging Queuing Protocol [11].

Within this paper the AMQP-defined messaging architecture is used. Within the core of the message broker architecture are queues; every message received by the RabbitMQ always is placed in a queue. Messages in queues can be stored in memory (memory-based) or on a disk (disk-based). Second important elements of the RabbitMQ are *exchanges* - the delivery service for messages. The exchange used by a publish operation determines if the delivery will be direct or publish-and-subscribe, for example. 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. This is how AMQP brings the various messaging idioms together - clients can select which exchange should route their messages [15].

A. Specific system design requirements for middleware

1) 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) 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),

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

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 applications are designed to have a high level of service uptime. HA solutions may feature many elements, e.g: system management, live replacement (hot-swap), component redundancy and failover mechanisms. Common strategy is to avoid single points of failure in the system. This 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.

Typical HA solution 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. Because these systems are not directly coupled, they utilize standard network connections to communicate failovers. This can cause failover latencies to take several seconds to complete. 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, possibly with the use of clustering.

Contrary to HA, which implies a service level in which both planned and unplanned outages do not exceed a small stated value [18], 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) [4].

But FT has its problems; 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.

B. Scalable and fault-tolerant middleware

Message broker, being one of the most crucial components of distributed system, is supposed to be fault-tolerant. That means, typical HA configuration (as described in II-A1) is not the best option; restarting message broker on another node in case of failure would introduce a timeout span, as the service is being restarted and prepared for operation, but, what is worse, the message queue of failed message broker would be lost entirely. For message broker, both HA and FT solutions were considered:

1) HA (Active/Passive solution): in which the downtime of message broker 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 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 [16] in order to manage message broker and restart it (or migrate) when necessary, using available resources.

2) FT (Active/Active solution): means that the planned or unplanned downtime of message broker doesn't 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.

The second solution is a typical Active/Active topology and is recommended as more reliable and scalable at the same time. Additionally, virtualization was not considered, because it would introduce additional conditions and variables to the experiments and is a subject for another study. This paper's research is concentrated on RabbitMQ message broker clusters and its characteristics.

C. 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, and linearly scaling messaging throughput by adding more nodes [21]. 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. [17]

The design decision that had to be made was an IP addressing of the cluster. As RabbitMQ documentation describes, it's 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 [12]), or some sort of mobile IP achieved with pacemaker or similar technologies [16]. For this study, HAproxy was chosen as a load balancer between clients and cluster nodes.

III. 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.

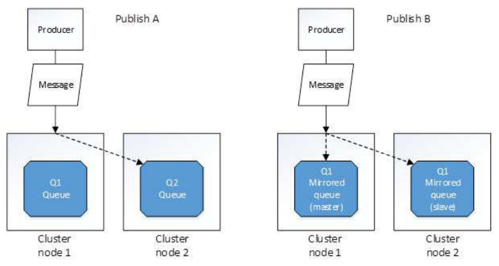

Fig. 1. Publishing to queues: a) on another node, b) to the mirrored queue.

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) [21]. 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.

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

1) 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,
2) Creating spread queues, but configure mirrored queues for at least one master and one slave (allowing for one node failure),
3) 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) is a compromise between those two.

A. Cluster and queues configuration

Considering a three-node cluster, one can come up for specific testing scenarios that can provide comparable results for performance assessment [14]. Those results may provide an answer, whether given configuration is useful for a specific real-world scenario [10]. For testing purposes, there were following implications made: a) cluster may include up to three nodes, b) queues are created in the cluster as a single

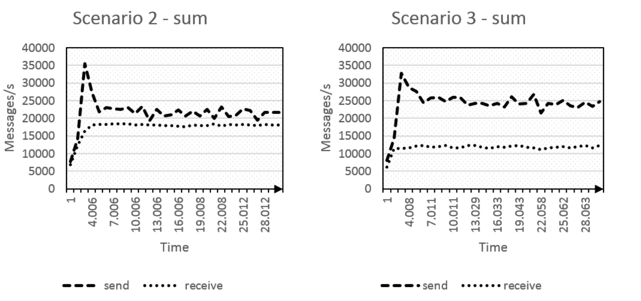

Fig. 3. The comparison of two-queue, one-client-per-queue publishes and consumes using scenario 2 and scenario 3.

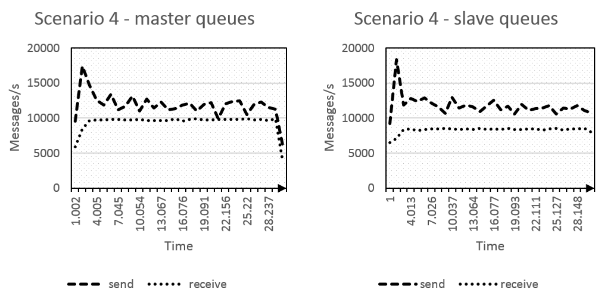

Fig. 4. Mirrored queues performance results example.

(non-mirrored), fully mirrored, and spread (mirrored to one node) queue, c) all of configuration scenarios are put to the three tests:

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

Possible configuration possibilities for two queues are presented on Fig. 2. On the left, 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 and adding one node for scalability (scenario 5) or spreading mirrored queues on available nodes (scenario 6).

B. Initial testing and results

All of scenarios were tested with commodity-equipped virtual machines (single core, 4GB RAM, 8GB HDD) 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. In 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.

The results of scenario 2 and scenario 3 (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 is interesting and should be 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. This design could be more appropriate with large number of publishers and consumers, but such study is out of scope for this paper.

The performance of sending and receiving to mirrored queues, as in scenarios 5 and 6, 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.

C. Detailed evaluation

The initial tests results show the importance of load balancing the traffic between the clients and cluster nodes. The message publishing or consuming rate depends whether the client was redirected to the:

1) ”master” node (the node which is the master for the specific queue being used),
2) ”slave” node (the node which specific queue is being replicated onto),
3) ”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 three nodes - two disk-based and one RAM-based. Such configuration assures 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, a single queue was tested for performance when configured as:

1) ”single” queue (not mirrored at all, for performance comparison and load balancing issues evaluation described before),
2) ”spread” queue (mirrored to one other node in the cluster, as suggested in scenario 6),
3) ”mirrored” queue (mirrored to every other node in the cluster).

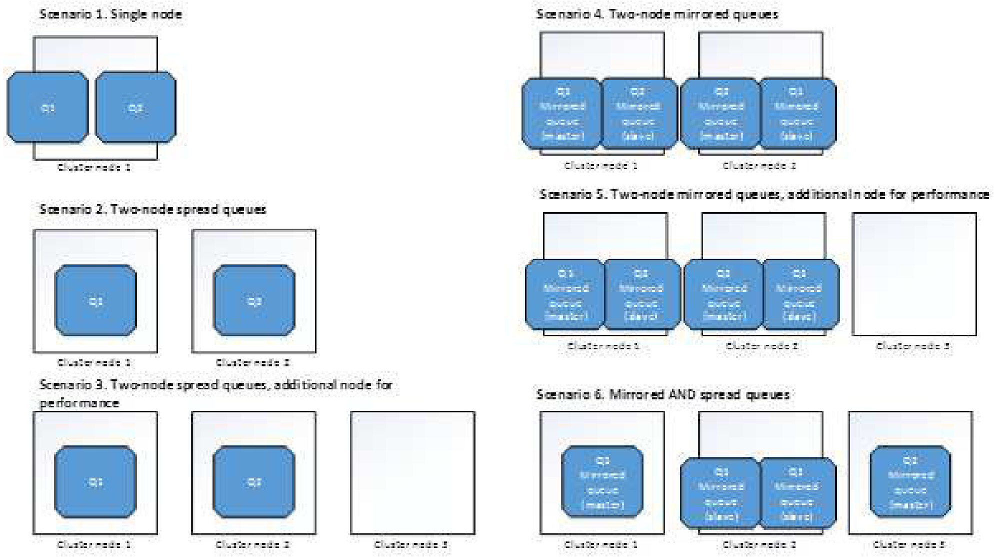
Scenario 1. Single node
Q1
Q2
Cluster node 1
Scenario 2. Two-node spread queues
Cluster node 1
Cluster node 2
Scenario 3. Two-node spread queues, additional node for performance
Cluster node 1
Cluster node 2
Cluster node 3
Scenario 4. Two-node mirrored queues
Mirrored queue (master)
Mirrored queue (slave)
Scenario 5. Two-node mirrored queues, additional node for performance
Scenario 6. Mirrored AND spread queues

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

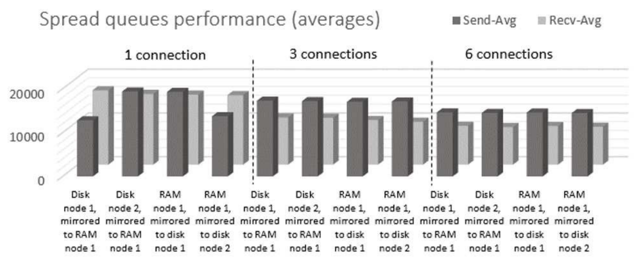

Fig. 5. Spread queues performance extensive testing results comparison

The results are presented in Table 1. For every queue configuration average message publication/consummation rates (for 9 consecutive tests) are shown, as well as standard deviation. The effect of load-balancing is visible mostly for single client consuming from mirrored queues—if the client's request is redirected to an empty node, the average message receive rate is significantly lower. Also, the effect is visible when sending to single-hosted queue, as the two other nodes are empty, therefore they need to redirect a request.

Fig. 5 presents exemplary detailed extensive testing run results in quantitative form, but gathered for comparison and conclusions. Every queue configuration was created on master disk or RAM node, then tested for a minute-long run ten consecutive times. Such tests were conducted for one, three and six simultaneous connections.

Most important observations are, as indicated by mutliple 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 fullmirrored 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.

IV. 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.

Relevant studies show 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, which can be a subject of next authors' study. This is authors' contribution in discussing solutions that combine both requirements, as it is a real industry scenario.

To summarize results, 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.

TABLE I
SUMMARIES FOR MOST IMPORTANT SCENARIOS (10 PUBLISHERS, 10 CONSUMERS)

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

REFERENCES

[1] 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.

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

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

[4] P. Buchwald, "The Example of IT System with Fault Tolerance in a Small Business Organization", in: *Internet—Technical Development and Applications 2*, Springer 2012, pp. 179–187

[5] F. Buschmann et al., it Pattern-oriented software architecture: a system of patterns, John Wiley and Sons, Inc. New York, NY, USA ÂŠ1996 ISBN:0-471-95869-7

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

[7] 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)

[8] 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

[9] 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.

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

[11] 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

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

[13] 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

[14] 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

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

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

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

[18] M. Rostanski, "High Availability Methods for Routing in Soho Networks", in *Internet - Technical Developments and Applications 2*, Springer 2011, pp. 154–152

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

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

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