Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 383846, 20 pages
http://dx.doi.org/10.1155/2015/383846

Research Article

Modeling Message Queueing Services with Reliability Guarantee in Cloud Computing Environment Using Colored Petri Nets

Jing Li,[1,2,3] Yidong Cui,[2] and Yan Ma[4]

[1]*State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[2]*School of Software Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China*
[3]*School of Electronic Information Engineering, Qiongzhou University, Sanya 572022, China*
[4]*Institute of Network Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China*

Correspondence should be addressed to Jing Li; lijinghnhb@bupt.edu.cn

Received 11 October 2014; Revised 30 March 2015; Accepted 2 April 2015

Academic Editor: Antonino Laudani

Motivated by the need for loosely coupled and asynchronous dissemination of information, message queues are widely used in large-scale application areas. With the advent of virtualization technology, cloud-based message queueing services (CMQSs) with distributed computing and storage are widely adopted to improve availability, scalability, and reliability; however, a critical issue is its performance and the quality of service (QoS). While numerous approaches evaluating system performance are available, there is no modeling approach for estimating and analyzing the performance of CMQSs. In this paper, we employ both the analytical and simulation modeling to address the performance of CMQSs with reliability guarantee. We present a visibility-based modeling approach (VMA) for simulation model using colored Petri nets (CPN). Our model incorporates the important features of message queueing services in the cloud such as replication, message consistency, resource virtualization, and especially the mechanism named visibility timeout which is adopted in the services to guarantee system reliability. Finally, we evaluate our model through different experiments under varied scenarios to obtain important performance metrics such as total message delivery time, waiting number, and components utilization. Our results reveal considerable insights into resource scheduling and system configuration for service providers to estimate and gain performance optimization.

1. Introduction

Originally designed to help communication between processes in operating systems, message queues are more widely used in various application domains nowadays. In particular in distributed environments, message queues connect different system components and ensure that they can exchange information asynchronously and reliably. Message queues typically support point-to-point communication fashion [1]. Traditional message queues are already widely applied in industry applications, for example, Active MQ [2], Rabbit MQ [3], Kafka [4], Microsoft MSMQ [5], and IBM Websphere MQ, many of which have gained good market repercussion from the users.

With the advent of virtualization technology, cloud computing becomes a popular topic for blogging and white papers and has been featured in the title of workshops, conferences, and even magazines [6]. Clouds can be defined as a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms, and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization [7]. Cloud computing provides lots of advantages, in which the most famous one is the on-demand provisioning of software, hardware, and data as a service, preventing the developers from being concerned with overprovisioning or underprovisioning for a service. Also, within the cloud, the laws of probability give service providers great leverage through statistical multiplexing of varying workloads and easier management [8]. Due to the popularity and merits of cloud computing, message queueing services in cloud computing environment (CMQSs) are broadly adopted

recently in commercial markets besides traditional message queues, such as Amazon Simple Queue Service (SQS) [9] and Microsoft's Windows Azure Queue [10], both of which have better support for advanced availability and scalability.

These novel cloud-based message queueing services, however, pose some serious performance and reliability challenges. For example, online payment services rely on messaging services which can provide high reliability and high performance so as to support the acquisition and processing of real-time data and also help to provide better application services and strive for greater market share for enterprise customers. For commercial cloud queueing services, an effective method is urgently needed to measure and predict the performance of services as well as their reliability and scalability. Specially, when facing system updating, it is essential that cloud services are subjected to a rigorous performance analysis before the launch of new version in order to make sure that the update will not bring up performance decline in some aspects. Moreover, because of the flexibility and loose coupling of CMQSs, performance monitoring and analysis should better be conducted continuously when providing services, so as to acquire customer feedback timely and make adjustments adaptively to achieve optimal performance accordingly. Common performance metrics of interest are the expected event notification latency as well as the utilization and message throughput of the various system components [11]. In addition, the issue of message consistency, one of the most essential features of message queueing services, is also discussed in this paper when modeling CMQSs, and two types of message consistency options are provided in the model.

Considering the advantages and disadvantages of the existing cloud-based message queues, we find that there is no modeling approach for estimating and analyzing the performance of CMQSs. Besides, the proposed prototypes of message queues mentioned in recent literature are mainly concerned with the system scalability while lacking emphasis on reliability. In this paper, we present an analytical method to model the performance of CMQSs, while considering the system performance with reliability guarantee. We also demonstrate a novel simulation modeling method using colored Petri nets, which is named as visibility-based modeling approach (VMA) for the reason that we model a visibility-timeout mechanism in the performance model, which is used to ensure the system reliability as well as availability. Normally, the distributed application driving system will potentially be hierarchically structured into layers [12]; thus, we present our CPN model in a hierarchical fashion accordingly. We compare the outcomes reached by both approaches to validate our model and give reasonable performance analysis according to the results.

The rest of this paper is organized as follows. In Section 2, we present a short review of the available literature. In Section 3, we provide a brief introduction to CMQSs as well as visibility-timeout mechanism. Section 4 describes the background on CPN. In Section 5 some necessary assumptions and the analytical modeling method are provided. Then we depict the simulation model using VMA approach in Section 6. Section 7 is focused on the numerical results of both models as well as the discussions about performance analysis. Finally in Section 8, some conclusions and directions of future work are given.

2. State of the Art

Numerous approaches to evaluate system performance are available in literature. Software performance engineering (SPE) [13] was firstly proposed by Smith for integrating performance prediction techniques into the software engineering process over two decades ago. Since then a large number of performance models are developed and can be classified into various categories, for example, queueing networks, layered queues, types of Petri nets, and stochastic process algebras, which were surveyed in [14]. Particularly, the authors of paper [15] have a special focus on component-based performance evaluation methods and survey their applicability in industrial use. In [16], some problems of existing modeling methods are summarized, for example, a lack of theoretical justification for the performance model and a difficulty in establishing causality across the subsystems for modeling distributed systems, in which events from different subsystems provided by different vendors need to be correlated. Also, the authors present expectations for the future techniques and tools in the paper.

For modeling the CMQSs described in our paper, we aim at addressing both of these two problems mentioned above. We employ both mathematical and simulation modeling methods to verify and evaluate the performance metrics of interest together, and the results achieved by the mathematical model can fit well with the simulation model under the out-of-order option. Although the analytical model described in this paper can only capture a subset of the whole system features (by reflecting the out-of-order model), it can still be seen as a heuristic method for solving the analytical model which can fully justify the simulation performance model theoretically. Moreover, for simulation model, we present a detailed timed colored Petri nets [17, 18] model in hierarchical fashion, for the purpose of giving prominence to the different subsystems that comprise the whole distributed message queueing system in cloud environment.

In academic world, some studies have been conducted to apply modeling approaches to messaging systems. The Palladio Component Model (PCM) [19] is one of the approaches used for software performance prediction. It is an architecture description language supporting design time performance evaluation of component-based software systems, and it can be extended to model specific kinds of systems; for example, the authors in [20] combine PCM with a performance completion [21] for message-oriented middleware (MOM) so that the software architects can specify and configure message-based communication using a language based on messaging patterns (e.g., publish-subscribe or competing consumers). They adopt a model-to-model transformation fashion which integrates the low-level details of a MOM into the high-level software architecture model and use a SPECjms2007 benchmark case to evaluate and verify their model. Also, [22] exploits the possibility of extending the PCM so as to support

the modeling of event-driven service-oriented systems. The authors define a set of mapping strategies between system elements and PCM model elements in order to eliminate the semantic gap between the system implementation and the architecture model caused by the approach proposed in [23] to modeling the asynchronous events using synchronous service calls.

In [11] the authors present a modeling methodology for message-oriented event-driven systems using a case study in the supply chain management domain. The methodology is based on queueing Petri nets [24] and the authors use a standard benchmark to test and verify their model. Also, the authors present a set of generic modeling patterns under different system scenarios, which can be applied in other performance models.

Time analysis is an important performance metric for message transmission, which is concerned by researchers. For example, in [25], the message-scheduling problems are discussed and an asynchronous communication model where messages are delivered in the form of packets is proposed. The authors analyze the packets completion time in a nonlinear way and propose an optimization algorithm in the case of priority messages. However, the message queues in the model and algorithm are only considered on the side of the sender and thus cannot predict the whole message delivery time through senders, brokers, and receivers.

As the features of messaging infrastructure (e.g., space, time, and synchronization decoupling between system components) are essential in distributed systems like clouds, products like Amazon SQS and Microsoft's Windows Azure Queue are becoming more and more popular and are widely used in industry. However, in literature, the discussion about analysis and improvements of cloud message queueing services still needs more attention and concerns. Some researchers have made preliminary exploration of combining message queues with cloud computing environment.

The EQS introduced in [26] presents an architecture of message queue which can scale elastically in the cloud. The authors implement a prototype of EQS based on an existing messaging library (i.e., ZeroMQ) and deploy and evaluate it on a cloud infrastructure (i.e., Amazon EC2) using cloud platform management tool Scalr. However, the results should not be taken as pure performance results but more as a validation of the elastic and dynamic nature of their algorithm in consideration of the dependence on best-effort nature of Amazon EC2 network I/O. Besides, the storage used in EQS is centralized rather than distributed, which is not realistic in cloud environment and could dramatically limit the system availability.

In [27], the authors present the BDQS, that is, a scalable cloud-based queueing service which is built on a distributed storage system Cassandra [28]. BDQS can provide at-least-once and best-effort in-order message delivery.

Another cloud-based message queueing service named SDQS is introduced in [29]; the authors implement the prototype of the message queue on top of a distributed in-memory cache, IBM WebSphere eXtreme Scale (WXS) [30], which is a special type of storage system. Based on the elasticity and fast access of the in-memory cache, SDQS can provide high scalability and availability, as well as multiple message delivery options. However, the authors focus their evaluation of the prototype on scalability rather than message delivery time and conduct the experiments of different process stages separately, that is, firstly the sending process and then the forwarding process, which is unlikely to happen in practical systems.

Since Hadoop has gained extensive usage in applications, a distributed message queueing system HBaseMQ, which is implemented as a light-weight client library to HBase [31], is presented in [32]. The authors use HBase system tables to represent queues, messages, and clients (i.e., senders/receivers), in which the "timestamp" columns are used to rapidly pinpoint a message by its timestamp. HBaseMQ inherits many well-tested cloud-ready properties from Hadoop/HBase such as scalability and fault tolerance. Moreover, HBaseMQ can provide two delivery guarantees, that is, "at most once" and "at least once." However, the performance evaluations of HBaseMQ are conducted coarsely and message delivery time analysis is not provided in the paper.

Because the popular commercial queueing service like Amazon SQS does not guarantee the order of messages, nor does it guarantee the exactly once delivery, the authors of [33] propose a hierarchical distributed message queue (HDMQ), which eliminates these defects and outperforms SQS in both throughput and delivery latency. HDMQ is also proved to support in-order and exactly once message delivery.

Realizing that cloud message queueing services are becoming popular in practical use encourages researchers to consider designing systems that can better cooperate with the CMQSs so as to achieve good system performance as well as high system utilization. In [34], the authors provide a highly scalable and distributed job management system CloudKon, which is built upon cloud computing building blocks (Amazon EC2, SQS, and DynamoDB). In [35], the author introduces a novel conceptual model Iris, a decentralized messaging framework, to provide a much simpler way to specify, design, and implement distributed, cloud-based services.

Petri net [36] is another widely used modeling technique for system performance analysis. Pioneered by Carl Adam Petri in his doctoral dissertation, Petri net was recognized as a powerful mathematical theory to describe concurrent and asynchronous systems. Due to the state space explosion problem caused by analyzing large complex systems, Jensen introduced an expansion form of Petri net, that is, colored Petri nets (CPN) [17, 18], which can reduce the size of system model dramatically through attaching colors to tokens in order to distinguish various tokens that represent different system types or components. At present, colored Petri nets are gathered with various features which include multiple colorsets, hierarchy, time, and high-level programming language (e.g., CPN ML programming language [17, 37]). As CPN can be used to model extensive systems or services which contain concurrent behaviors, a large number of researchers have exploited CPN in sophisticated ways to conduct studies in their own research fields. For example, in [38], CPN is used to model network protocols, in [39], workflow executions are modeled, and in [40] manufacturing systems are analyzed.

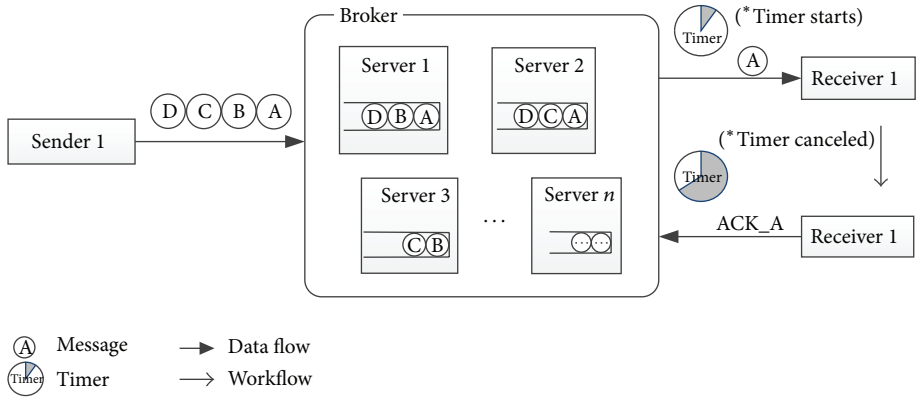

Figure 1: Message delivery process in CMQSs.

Moreover, specialized applications like train classification, greenhouse management, ERP projects, traffic control [41], or embedded systems are also able to be modeled using CPN.

3. System Description

In this paper, we model broker-based message queueing service into the cloud computing infrastructure, and the most famous CMQSs of this kind are the commercial application Amazon SQS [9].

Implementing message queueing service into the cloud computing environment has the advantage of delivering messages with low cost and high availability and scalability. Distributed message queueing service deployed in the cloud can queue messages quickly and reliably while one component in the application environment can consume the messages that are generated by another component. Also, the queue used in this process can be deemed as a temporary repository for messages that are waiting for processing; for example, Amazon SQS has a message retention period that expands from 60 seconds to 14 days which can be determined by the function *SetQueueAttributes()* provided in the application programming interface. To ensure continuous available services, a persistent distributed storage system is also needed in cloud-based message queueing services, for example, Cassandra in BDQS [27], IBM WXS in SDQS [29], and HBase in HBaseMQ [32].

In order to prevent loss of messages and guarantee the reliability of system, CMQSs adopt a control mechanism named visibility timeout. This mechanism assigns a visibility state to every message, indicating whether a message is available to be processed. In other words, this visibility state can be assumed as a lock; a message with visible state is seen as unlocked while a message with invisible state is locked. A message newly received by broker is initially in visible state (i.e., unlocked); it becomes in invisible state (i.e., locked) while being processed; meanwhile, a time window known as visibility timeout is set especially on this message to prevent simultaneous retrieving of the message from other receivers during this period. Under the circumstance of successful message transmission, an acknowledgment notification needs to be issued by the receiver so that the broker can delete the corresponding message without hesitation. Otherwise, if the message timer expires (i.e., timeout happens), broker can reset the visibility state of the message as visible, which means the message is available to be processed again. An example of message delivery process in broker-based CMQSs is illustrated in Figure 1.

Message consistency has been the subject of intense scholarly discussion since the emergence of CMQSs. Although Amazon SQS provides extremely high availability, it cannot guarantee message ordering as well as single delivery. This is because of the replication storage of messages and the server sampling scheme, which are adopted in the system to guarantee high availability. As demonstrated in Figure 1, broker replicates messages in multiple server nodes to prevent single point failure. When retrieving messages, one of the nodes is randomly sampled and returns a message in its queue. Thus if different nodes which are retaining different replications of messages are selected during successive processing requests, messages could be delivered without ordering. In both [29, 33] the authors provide an improvement of message consistency in their systems, which are claimed to guarantee in-order and no duplication delivery. Our modeling method takes message consistency into account and presents two different models accordingly, that is, the in-order and out-of-order models. We address the modeling of in-order delivery type based on the solution provided in [29], that is, using another queue, queue-index, to maintain the order of messages; every time before processing a message, the index needs to be accessed first so as to indicate the oldest message stored in system and then begins the processing of the corresponding message. In general, although the two types of models are distinct with respect to different message consistency options, the modeling methods are similar in other parts. We introduce our modeling method in the case of out-of-order delivery option at first and then depict the differential between the two models.

4. Background on CPN

Colored Petri nets (CPN) are a language for the modeling and validation of systems in which concurrency, communication, and asynchronization play a major role. Modeling complex

processes in terms of CPNs is a nontrivial task [42]. This paper uses CPN to model CMQSs, so some basic concepts are presented here. The following definition for CPN is given by Jensen [43].

Definition 1 (CPNs). A CPN is a 9-tuple defined as CPN = $(\Sigma, P, T, A, N, C, G, E, I)$, where the following statements are true.

(i) Σ is a finite set of nonempty types, also called colour sets.

(ii) P is a finite set of places.

(iii) T is a finite set of transitions.

(iv) A is a finite set of arcs such that

(a) $P \cap T = P \cap A = T \cap A = \emptyset$.

(v) N is a node function. It is defined from A into $P \times T \cup T \times P$.

(vi) C is a colour function. It is defined from P into Σ.

(vii) G is a guard function. It is defined from T into expressions such that

(b) $\forall t \in T: [\text{Type}(G(t)) = B \wedge \text{Type}(\text{Var}(G(t))) \subseteq \Sigma]$,

where the type of a variable v is denoted as Type$[v]$ and the set of free variables in an expression e is denoted as Var$[e]$.

(viii) E is an arc expression function. It is defined from A into expressions such that

(c) $\forall a \in A: [\text{Type}(E(a)) = C(p)_{\text{MS}} \wedge \text{Type}(\text{Var}(E(a))) \subseteq \Sigma]$,

where p is the place of $N(a)$ and $C(p)_{\text{MS}}$ denotes the set of all multisets over $C(p)$.

(ix) I is an initialization function. It is defined from P into closed expressions such that

(d) $\forall p \in P: [\text{Type}(I(p)) = C(p)_{\text{MS}}]$.

Currently, CPN Tools is by far the most widely used Petri net tool [42]. We adopt CPN Tools version 4.0 to conduct our modeling, availing our CPN model of the new features supported by this version, for example, real timestamp, time colorsets, and advanced communication.

As an industrial-strength computer tool for constructing and analyzing CPN models, CPN Tools realizes the graphical construction, editing, simulation, and analysis of CPN model through eleven different pallets. In the aspects of graphical construction and editing, CPN Tools provides a hierarchical method to make sure that the constructed model is clear to understand but without affecting system functions. Two ways are available to achieve hierarchical design: one is using subtransitions and the other is using fusionsets. Both of them can connect different places or transitions in different pages; more specifically, subtransition exhibits complex submodels using the concept of subnet whereas fusionset builds bridges between several submodels through combining distinct places. To create hierarchical descriptions, one should first divide large systems into several small nets, identify relationships between these subnets, and then define interfaces both among subnets and between subnets and upper layer nets. Usually, a system can be divided into multilayers and the complexity of the original system leads to deciding how deep the model hierarchy would be the most suitable. Abstractly speaking, in upper levels, model contains system outline without detail whereas in lower levels every important behavior is depicted in detail.

Additionally, time plays a significant role in a wide range of concurrent systems, and different time control strategies may have huge impact on the performance of systems. It is possible to capture system event execution time and analyze the control relationship between events through the concept of time included in CPN model. CPN Tools adds time stamp on tokens so as to keep tracking of the execution time of system events, thus further utilizing the time model to calculate performance metrics such as events delays, system throughput, system components utilization rate, and queue length.

5. Analytical Queueing Model

In this paper, a message queueing system with reliability guarantee is considered, in which the visibility-timeout mechanism is used to ensure the system reliability; that is, no message is lost during the delivery. We can consider the message delivery process as a Markov process. A message that is produced by an application in a client-end server (i.e., sender) which is waiting to be sent to server nodes in the cloud (i.e., broker) for transmission process will be called customer. Assume the arriving process of customers is Poisson stream with the intensity value equal to λ [44]. Thus, the interarrival times of customers are in accordance with the exponential distribution and on average a new message will be produced every $1/\lambda$ time.

The service process of customers in the whole system can be considered containing two phases. The first phase is the process of transmission for customers from senders to brokers and the second phase is the process of message forwarding from brokers to other applications in client-end servers (i.e., receivers). The two processes are considered to be cascaded in system model. However, as these two phases are asynchronous processes rather than synchronized with each other, there are independent waiting buffers (i.e., queues) inside themselves, respectively; we assume that the capacities of both buffers are infinite. The service time in both processes is considered exponentially distributed with parameters μ_1 and μ_2, respectively. Thus the average service time of the two phases is $1/\mu_1$ and $1/\mu_2$, respectively. Actually, before processing, it is time consuming to search in the message queue; thus, system side scalable acceleration is needed [45]. Nevertheless, in cloud computing environment, the scalability can be realized by adding CPUs or using state-of-the-art CPUs because multicore can execute linear

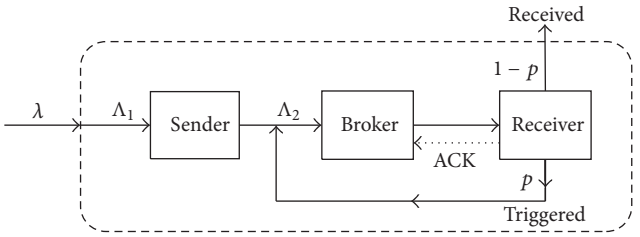

Figure 2: Queueing performance model for cloud message queueing services with reliability guarantee.

searching for queue with shorter depth (i.e., message number) in parallel; as a result, the performance of queue searching can be improved easily. Moreover, according to [45], searching speed of message queue when the order of message reception is different from the receiver's expectation is accelerated; this means the extra searching time caused by message ordering requirement is very small. In the light of these reasons and the fact that the message searching time is in microseconds, we assume that the time is so small which is negligible in our model, regardless of message consistency options.

Normally, when a customer is served by these two processes, the applications in receiver will retrieve the message successfully. However, because of the unpredictable situations in network, for example, network congestion or link failure, messages may be lost or delayed for a long time during transmission, thus degrading the system performance. These failure events have been considered and modeled in literatures. For example, [46] has discussed the performance of distributed, content-based publish-subscribe systems while considering the uncertainty in underlying overlay networks in the proposed model. In cloud queues, by performing the visibility-timeout mechanism to guarantee system reliability, those messages which are believed to be lost will be sent again from brokers to receivers, which means some customers in the model will be served again through the second process. Let p be the probability of failure (including network failure and client failure), which means the proportion of messages which needs retransmission is p, and thus the number of messages that have been retrieved successfully by the receiver (i.e., the arrival rate in receiver) is reduced by a factor $q = 1 - p$, which is the reliability probability.

According to the above descriptions, we can model the cloud message queueing system as an open Markov queueing model, which is depicted in Figure 2. The model consists of two service processes, that is, Sender to Broker (denoted as Proc1) and Broker to Receiver (denoted as Proc2). Suppose the arriving rates of customers in Proc1 and Proc2 are Λ_1 and Λ_2, respectively. In open Markov queueing model with 2 server nodes (i.e., the sender queue and the broker queue), because of the continuity of customers and the flow conservation feature, under the circumstance of infinite waiting capacity, the arriving rate of customers is equal to the departure rate of queue [44], and we can have the balance equation for a server node i:

$$\Lambda_i = q_{s,i}\lambda + \sum_{k=1,2} q_{k,i}\Lambda_k, \tag{1}$$

where λ denotes the customers arriving rate of the source node s, Λ_i denotes the arriving rate of node i, $1 \le i \le 2$, and $q_{k,i}$ denotes the probability of a customer who has finished the service from node k and goes to node i for the next service. Thus from the model shown in Figure 2, the arriving rate equation for each node can be expressed by

$$\begin{aligned} \Lambda_1 &= \lambda, \\ \Lambda_2 &= \Lambda_1 + \mathrm{p}\Lambda_2. \end{aligned} \tag{2}$$

To solve the equations we have

$$\begin{aligned} \Lambda_1 &= \lambda, \\ \Lambda_2 &= \frac{\lambda}{1-p} = \frac{\lambda}{q}. \end{aligned} \tag{3}$$

Assume the number of customers in node i (including waiting in queue and being served) is l_i and the probability that $l_i = k_i$ is denoted by $p_i(k_i)$. The system matches the hypotheses of Jackson theorem on queueing networks, and therefore solutions follow the product form.

According to [44], in open Markov queueing networks, the joint probability $P(k) = P(k_1, k_2)$, which means the number of customers in sender queue and broker queue is k_1 and k_2, respectively, can be expressed as follows:

$$P(k) = \prod_{i=1,2} p_i(k_i), \tag{4}$$

indicating that the mean number of customers in each node is independent of each other. In our model, the processes Proc1 and Proc2 are both modeled as an M/M/N queueing, so we can derive the mean number of messages in our model from the queue length distribution in M/M/N queueing model, which is

$$p_i(k_i) = \begin{cases} \dfrac{(n_i\rho_i)^{l_i}}{l_i!} S_i, & k < n_i \\ \dfrac{n_i^{n_i}\rho_i^{l_i}}{n_i!} S_i, & k \ge n_i, \end{cases} \tag{5}$$

such that

$$\begin{aligned} &1 \le i \le 2, \\ &S_i = \left(\sum_{j=0}^{n_i-1} \frac{(n_i\rho_i)^j}{j!} + \frac{(n_i\rho_i)^{n_i}}{n_i!(1-\rho_i)} \right)^{-1}, \\ &\rho_i = \frac{\Lambda_i}{n_i\mu_i}. \end{aligned} \tag{6}$$

Note that because we use multithreads to handle each message delivery process, n_i means the number of threads in sender and broker and μ_i means the average message processing time of each thread in sender and broker. So the

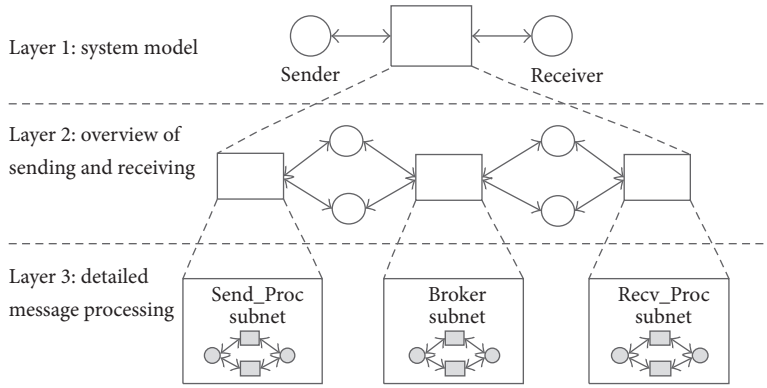

Figure 3: A three-level hierarchical structure of timed CPN model.

mean number of messages in the system (including sender and broker) is calculated as

$$\begin{aligned} L &= L_1 + L_2 \\ &= \sum_{k=n_1}^{\infty} (k - n_1)\, p_1(k) + \sum_{k=0}^{n_1} k p_1(k) + n_1 \sum_{k=n_1+1}^{\infty} p_1(k) \\ &\quad + \sum_{k=n_2}^{\infty} (k - n_2)\, p_2(k) + \sum_{k=0}^{n_1} k p_2(k) + n_2 \sum_{k=n_2+1}^{\infty} p_2(k) \\ &= \frac{\rho_1 (n_1\rho_1)^{n_1} S_1}{n_1! (1-\rho_1)^2} + n_1\rho_1 + \frac{\rho_2 (n_2\rho_2)^{n_2} S_2}{n_2! (1-\rho_2)^2} + n_2\rho_2. \end{aligned} \tag{7}$$

Applying Little's law, we can have the sojourn time of messages in Proc1 and Proc2 as

$$\begin{aligned} W &= \frac{L}{\lambda} \\ &= \frac{(n_1\rho_1)^{n_1} S_1}{n_1\mu_1 n_1! (1-\rho_1)^2} + \frac{1}{\mu_1} + \frac{(n_2\rho_2)^{n_2} S_2}{p n_2 \mu_2 n_2! (1-\rho_2)^2} \\ &\quad + \frac{1}{(1-p)\mu_2}. \end{aligned} \tag{8}$$

Finally, assume the one-way network link delay as D_{link}; considering that before a message has been successfully received by the receiver, two times of network transmission processes are needed, and an extra message retransmission process is also needed in case failure occurs, then the total message delivery latency can be gained as

$$T = W + (2 + p) D_{\text{link}}. \tag{9}$$

6. Hierarchical Timed CPN Model

In this section, we introduce our visibility-based modeling approach (VMA) for message queueing services in the cloud concerning reliability using colored Petri nets (CPN). Our approach includes two features: firstly the inherent hierarchical characteristic of timed CPN to afford model complexity and secondly the steps that we employed to construct the detailed submodels from the beginning [47].

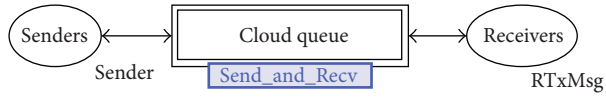

Figure 4: Layer 1 model.

6.1. System Model Architecture. From a programming point of view, we construct timed CPN model in different modules and organize them in different hierarchy levels. We present a three-level hierarchical timed CPN model for CMQSs, as the model structure is shown in Figure 3.

At the top layer of the hierarchical structure is the system model, which represents an overview of system behavior; it is the layer which depicts the semanticity of message queue most clearly. However, we should notice that the double-headed arrows between sender and receiver do not indicate synchronous communication; on the contrary, they are decoupled and exchange events asynchronously due to the distributed environment. The corresponding model is shown in Figure 4. The scenario manifested here is as follows: sender computes the unique identity number of the corresponding queue, locates the queue, and puts message into the server nodes redundantly where the queue exists; similarly, a receiver needs to compute and locate the corresponding queue first when trying to retrieve messages; then a sampled server node which holds the queue returns a message to the receiver according to consistency requirement specified by the receiver. These detailed behaviors are described in submodels, and according to the consistency options discussed in the above section, the submodels can be classified into two types: out-of-order and in-order.

6.2. Detailed Submodels Description. In Figure 5, the square with double border is called subtransition, representing an abstraction of a CPN subnet, that is, model in the lower layer of the hierarchical structure. The middle layer is a detailed expression of subtransition *cloud queue* in layer 1, as well as

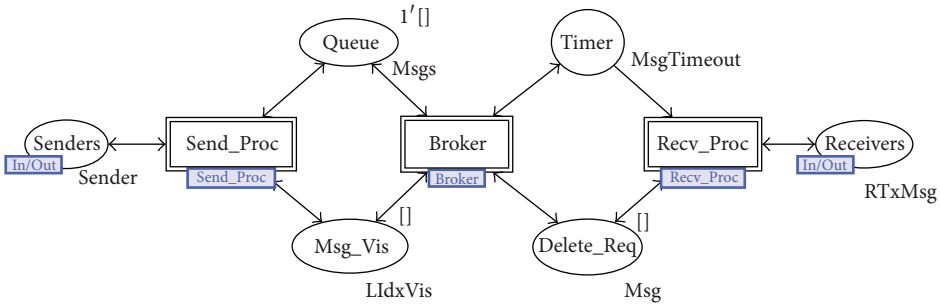

Figure 5: Layer 2 model.

an abstraction of detailed message delivery process models in layer 3. Figure 5 exhibits the model of layer 2, which shows the major system components of CMQSs unambiguously as well as strengthening the impression of asynchronous communication of messages through the information flow. Nevertheless, this layer still provides the procedures of message flow via queues macroscopically for the fact that three subtransitions *Send_Proc*, *Broker*, and *Recv_Proc* hide the detailed behaviors.

In this paper, we describe the three kernel submodels in layer 3 step by step in the following, and here we employ the methodology introduced in [47].

Step 1 (determine model elements: places). Firstly, we have to map the system components and resources into respective elements of CPN model. Generally, hardware or software resources, such as thread, CPU, disk (IO) processes, and network, are usually modeled using tokens inside places. The number of tokens is used to show the size of available resources in the system. Logical entities such as messages, queues, and timers are also modeled using places and tokens in them. Secondly, because of the color feature provided in CPN, we need to assign a specific colorset to each place in which the tokens will be put on the same color. All the tokens in a place can be expressed as a multiset whose elements are predefined colors in colorsets. From the perspective of structure, there are two forms of colorsets: simple colorsets and compound colorsets; usually the latter are defined using the previously declared former. The colorsets used in our models are presented in Tables 1 and 2. Typical resources such as sender, receiver, thread, and visibility are modeled as simple colorsets while message, message queue, and visibility indication are belonging to compound colorsets. Take message queue as an example; the *List* colorset is commonly used to represent queueing structure in modeling, so we use colset Msgs = list Msg to represent the color of token in the queue that stores all messages sent by related senders. Queue operations such as enqueue, dequeue, and search can be implemented through the list operation interfaces supported by SML language. Particularly, time stamp is a way that CPN provides to indicate the resource usage time; it can only be used in timed colorsets of which the declarations are being appended by the keyword *timed* at the end. Tokens in timed places are timed tokens, which is identified by the charterer @ + and a number followed, indicating the specific model time at which the token is available for output transitions to use. This time stamp can also help to calculate the resources occupation time of system behavior, with the time length equal to the difference of current time stamp and the produce time stamp of the token. For instance, a token with color *MsgTimeout* is a timed token, indicating the specific timeout point of a message.

Table 1: Simple colorsets.

Simple colorsets
(1) colset UNIT = unit timed;
(2) colset INT = int;
(3) colset MsgTimeout = int timed;
(4) colset REQ = with req timed;
(5) colset Sender = with sender;
(6) colset Thread = with thread;
(7) colset MsgContent = with A | B;
(8) colset Visibility = with visible | invisible;

Step 2 (determine model elements: transitions). System behaviors that lead to state change are generally modeled as transitions, and the action of transition fire will cause tokens to disappear or emerge in the places connected with the transition. An arc from place to transition is called output arc and, conversely, input arc. Through output arc, tokens disappear in places while, through input arc, tokens emerge in places. Thus, using tokens, we can model the resources flow of system behaviors explicitly. Function is one of the features provided by CPN to model complex system workflows, objects, operations, or particular restraints. In transitions and arcs, there are special spaces to describe those functions needed in model, called code segments and expressions, respectively. Figure 6 shows the interaction model of message flows, that is, the *Send_Proc* submodel. In this submodel, there are 13 places and 7 transitions in all.

Step 3 (analyze workloads in model). After determining which resources and processes should be included in model, we need to analyze the workloads that system would involve. In distributed computing and storage systems, CMQSs are usually used as a middleware to communicate with independent components; it receives information (i.e., messages)

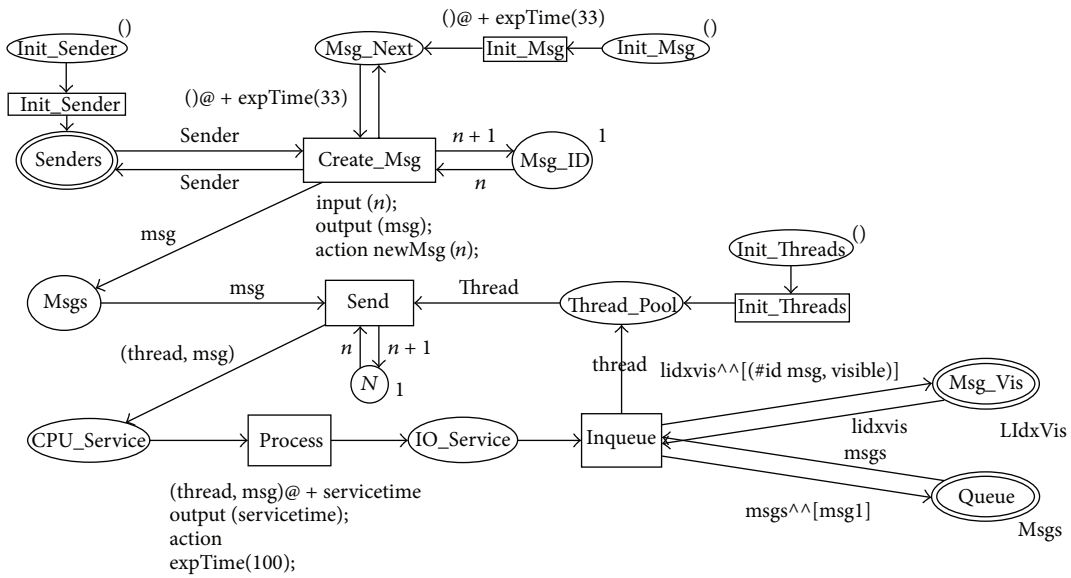

Figure 6: Detailed *Send_Proc* submodel.

Table 2: Compound colorsets.

Compound colorsets
(1) colset Msg = record id: INT * msgContent: MsgContent * AT: INT * IT: INT * OT: INT;
(2) colset MsgTimed = record id: INT * msgContent: MsgContent * AT: INT timed;
(3) colset Msgs = list Msg;
(4) colset ThreadxMsg = product Thread * Msg timed;
(5) colset MsgxTime = product Msg * INT;
(6) colset LMsgxTime = list MsgxTime;
(7) colset IdxVis = product INT * Visibility;
(8) colset LIdxVis = list IdxVis;

from senders and distributes them to receivers. Thus, the cloud message queues workload is mainly composed of one kind of item: message.

For the workload intensity, assume the interarrival times of messages are exponentially distributed with the mean interarrival time equal to $1/\lambda$, and this means the input process of messages is Poisson process with parameter λ. CPN Tools provides function for generating values from exponential distribution: *exponential(r: real)*, where the mean value is equal to $1/r$.

In our model, each token that represents message is made up of message ID and content. Message ID is an increasing sequence to ensure both the production and the processing of messages are ordered; we simply assume it as integer number with initial number equal to 1. This is set in the model by putting the initial marking of place *Msg_ID* in Figure 6 as 1 and increasing the marking by 1 each time of message production. Noticing that the message content and length are varied in practical systems, which leads to the unequal process time of messages, we assume in our model that the processing time of messages through system components (e.g., CPU and disk) accords with exponential distribution with parameter μ; that is, the mean service time is $1/\mu$. Additionally, in order to facilitate the data collection process and performance analysis of our model, we join three time stamps into message token (see Table 1): AT (message generation time), IT (message enqueue time), and OT (message dequeue time). The example usage of these time stamps will be explained in Section 7.1.

Step 4 (construct detailed CPN model). After identifying the basic elements which should be included in the model, we can establish a full CPN model that represents various system components and includes all the principal system behaviors, so as to correctly take advantage of the performance analysis results of the model and identify the possible bottleneck that exists in the system. Certainly, just like the iterations and code review of programming used in software engineering, CPN model also needs several times of adjustments and refinements to obtain a model that addresses the initial modeling goals.

Figures 7 and 8 depict the detailed *Broker* submodel and *Recv_Proc* submodel, respectively, that is, the detailed

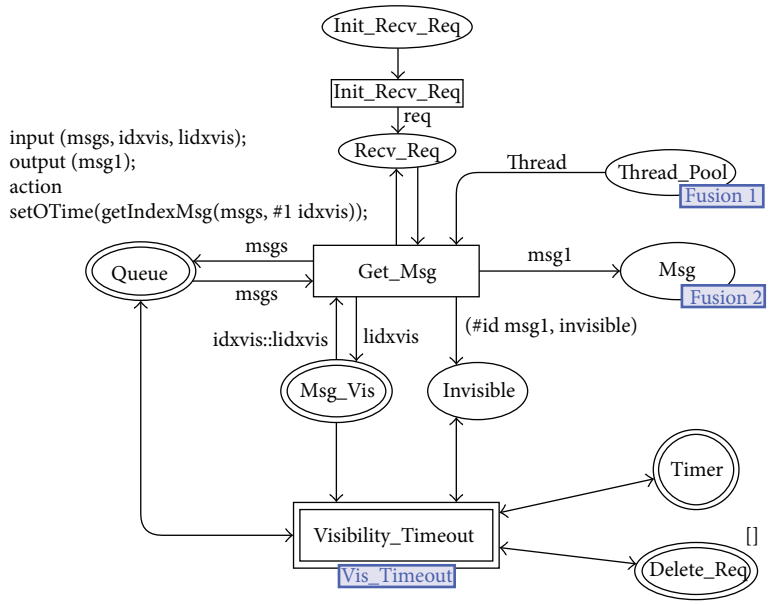

Figure 7: Detailed *Broker* submodel.

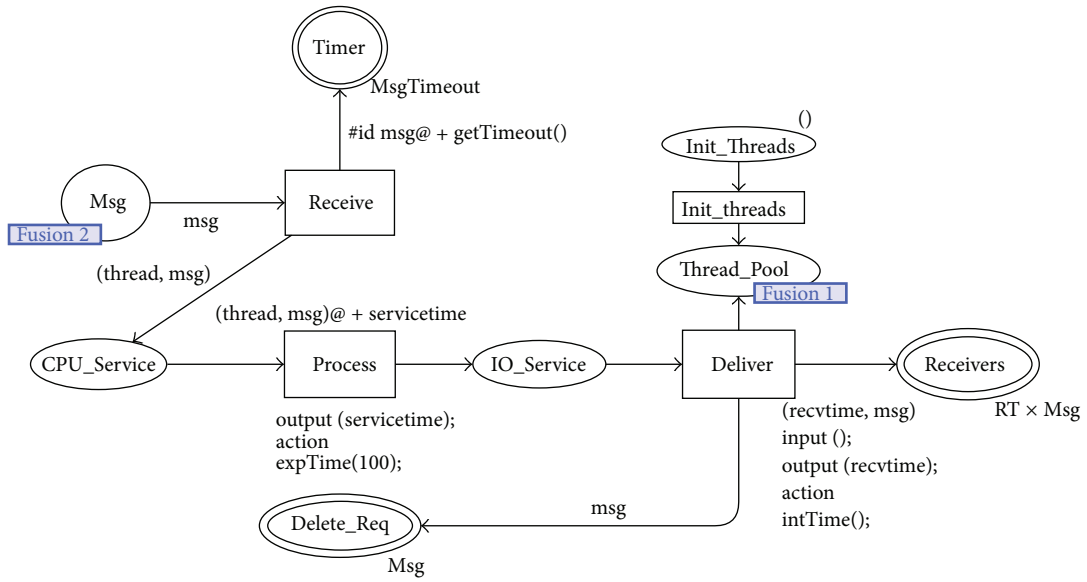

Figure 8: Detailed *Recv_Proc* submodel.

message forwarding process in cloud. Notice in Figure 7, the place *Queue* in *Broker* submodel represents the logical queue of messages in distributed storage cluster in the cloud. Actually, the real contents of messages are stored in replication among multiple data server nodes to support system robustness, as discussed before, and the allowance of simultaneous nodes failure to maintain system liveness is $n - 1$ given the total number of server nodes equal to n (see Figure 1).

6.3. Modeling Threads Contention. In distributed systems, application usually uses multiple threads to work simultaneously in its host to communicate or cooperate with others. The adoption of multithreads mechanism not only enables the processes of tasks in cloud to execute at the same time, but also improves the system resource utilization by sharing. We include this multithreads communication mechanism in our model (see place *Thread_Pool* in Figure 6 and Figure 8) and consider it as server windows with number equal to n in queueing analytical model; as a result, the message handling process can be modeled as an M/M/N queueing model as discussed in Section 5. What is worth mentioning here is that, due to the scalability mechanism supported by the cloud, we assume there are n CPUs in a server node given the number of threads equal to n, so as to ensure that these threads can

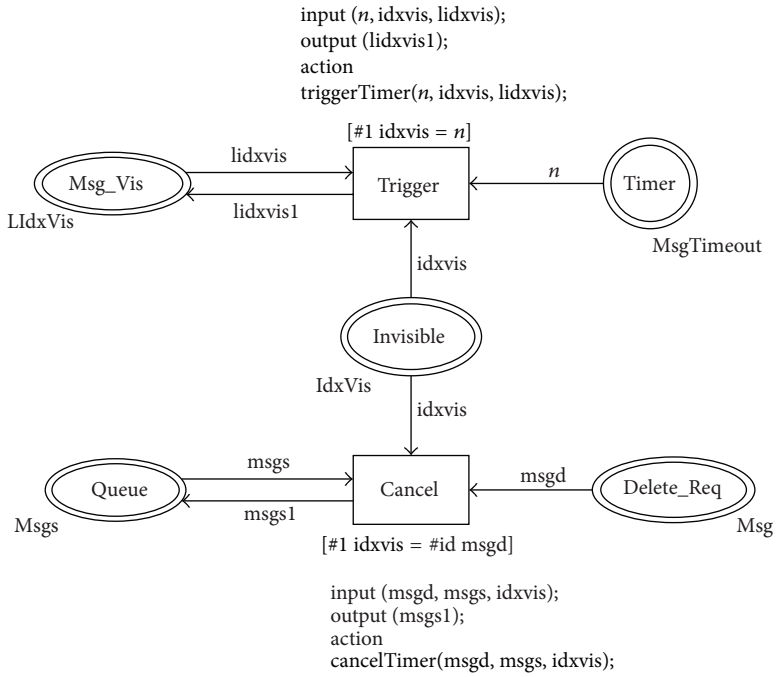

Figure 9: Visibility-timeout model.

work in real parallel rather than alternation in different time slice.

Consider n_1 and n_2 are the number of threads in sender and broker, respectively. If n_1 or n_2 is equal to 1 or a very small number, there will be a long time of response delay for each concurrent message processing because of waiting for idle thread. So not only the message waiting time and the delivery time will be relatively large but also the actual interval processing time does not meet the Poisson process, which is obeyed by the interval message production time. On the contrary, if n_1 or n_2 is relatively very large compared with system workload, we can assure that every time when a message needs to be sent or received, an idle thread is always available for processing immediately. Consequently, the waiting and delivery time of message processing is shorter. However, if $n_2 \ll n_1$, the queue length in broker is possible to increase sharply. Moreover, more idle threads would occupy more system resources without improving system throughput, thus decreasing system utilization. So the simulation and performance results of our model can help to analyze as well as provide suggestions for the optimal system configurations, for example, optimal threads number, in order to achieve relatively minimum mean message waiting time, minimum mean delivery delay, and the maximum system resource utilization rate.

6.4. Modeling Visibility-Timeout Mechanism. Visibility-timeout mechanism is adopted in commercial CMQSs such as Amazon SQS and Windows Azure Queue to manipulate the message lifecycle and it is included in our model as a unique characteristic which guarantees the reliability of system (i.e., ensuring every message is delivered despite client failure or network failure) and distinguishes our model from others.

The model of visibility-timeout mechanism is illustrated in Figure 9; it is a lower layer submodel that details the subtransition *Visibility_Timeout* which is contained in the *Broker* submodel. Every time when the broker is ready to process a message and send it to a receiver, the real content of this message is still stored in queue rather than be deleted immediately; instead, the visibility of message acts as an indication of whether a message in queue is being processed or not.

The default message visibility is visible, which means this message is waiting in queue and is available to be processed and delivered. As shown in Figure 9, the place *Msg_Vis* indicates these visible messages. The color of place *Msg_Vis* is *LIdxVis*, which means the tokens in the place are in the form of (*msgid, visible*), and notice that this place plays as an interface to connect with the *Send_Proc* submodel shown in Figure 6. Every time when a newly produced message has been inserted into the queue, a new token emerges in the place *Msg_Vis*, in which the "msgid" field of the token accords with the unique sequence number of this message. Then every time when the broker delivers a message, it retrieves a message whose visibility is visible from the queue. Once the message is being processed, the visibility of this message turns into invisible, which being reflected in our model is the fire of the transition *Get_Msg* in *Broker* submodel (see Figure 7), and the token (*msgid, visible*) is deleted and a new token (*msgid, invisible*) is added in place *Invisible*.

Additionally, when a message *A* is being processed, a timer is set to this message. If the broker receives the acknowledgment message for *A* (i.e., the *ACK* message instructs

Function description: cancel the timer on the message and delete its real content in queue
Source code:

```
(1) fun cancelTimer (msgd: Msg, msgs: Msgs, indexxvis: IndexxVis, lindexxvis: LIndexxVis) =
(2)     let
(3)         val msgd0 = setOTimeZero (msgd)
(4)     in
(5)         if #1 indexxvis = #id msgd
(6)         then (listsub msgs [msgd0], lindexxvis)
(7)         else (msgs, indexxvis::lindexxvis)
(8)     end;
```

Algorithm 1: Function cancelTimer.

Function description: trigger the timer on the message and make it ready to be sent in queue
Source code:

```
(1) fun triggerTimer (n: int, indexxvis: IndexxVis, lindexxvis: LIndexxVis, lindexxvis2: LIndexxVis) =
(2)     if #1 indexxvis = n
(3)     then ([(#1 indexxvis, visible)] ^^ lindexxvis, lindexxvis2)
(4)     else (lindexxvis, indexxvis::lindexxvis2);
```

Algorithm 2: Function triggerTimer.

the broker to delete the original message A in queue after successfully delivering A) in time, the system will cancel the timer and delete all replications of A stored in cloud. Otherwise, the timer will trigger and the broker will send A again assuming that the last sending failed. In Figure 9, the place *Timer* models the timers set for every message and the place *Delete_Req* models the acknowledgment messages. The transitions *Cancel* and *Trigger* model the actions of system under two opposite message delivery situations, respectively, while the detailed action codes are shown in Algorithms 1 and 2, respectively. Specially, when the timer is triggered, the token (*msgid, invisible*) is removed from place *Invisible* and the token (*msgid, visible*) is added again in place *Msg_Vis*; that is, the message in queue is available to be sent again.

6.5. Modeling Queue Operations. The queue operations are contained in *Broker* submodel. As mentioned above, the place *Queue* models the logical queue of messages in distributed storage cluster in the cloud and the type of colorset of the place is *List*, so queue operations like enqueue (i.e., inserting a message into queue) and dequeue (i.e., retrieving a message from queue) are implemented using list-related functions. The enqueue operation is simply realized using list connection function while the dequeue operation is more complex. This is because the real content of message is still stored in queue while the message is being processed, so we have to retrieve an available message according to the visibility indication, which is referred to as the place *Msg_Vis* in our model. The implementation code of dequeue operation uses a recursive procedure, as shown in Algorithm 3.

Function description: find the oldest message in queue
Source code:

```
(1) fun getIndexMsg (msgs: Msgs, index: int) =
(2)     if #id (hd msgs) = index
(3)     then (hd msgs)
(4)     else getIndexMsg ((tl msgs), index)
```

Algorithm 3: Function getIndexMsg.

As discussed in previous section, we model two message consistency situations in CMQSs in this paper, that is, the in-order option and the out-of-order option. Notice that all the figures displayed above are suitable for both options except the *Broker* submodel (see Figure 7, which shows the graph under the out-of-order option), for the reason that there is a difference in the dequeue operation between them.

In [29], the authors suggest using another queue named queue-index to store the sequence order of messages. This queue-index queue exists in the distributed storage clusters just like the message queue except that the former is not stored in replication but rather in only one partition. So every time when delivering a message, the system visits the queue-index queue first to get the message sequence number of the oldest one before retrieving the corresponding message from the queue. Because the authors use IBM WebSphere eXtreme Scale [30] as an in-memory cache storage as well as the FIFO ObjectMap API provided by it to realize the oldest message access process, we can assume that the searching time of the oldest message through multiple server nodes is negligible.

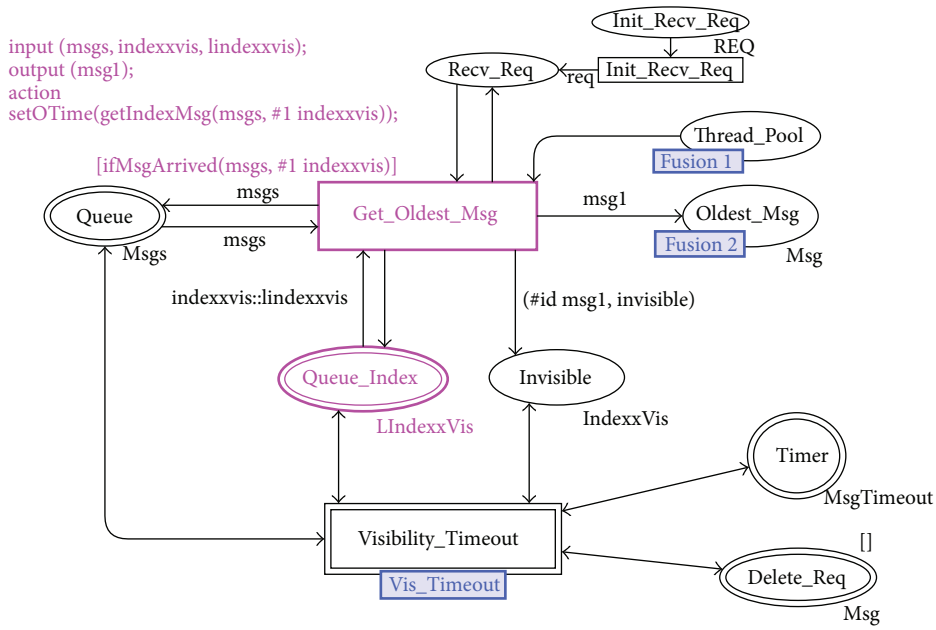

Figure 10: Detailed *Broker* submodel with consideration of message ordering.

Function description: determine whether the corresponding message has already exists in queue
Source code:

```
(1) fun ifMsgArrived ([], index: int) = false
(2)   | ifMsgArrived (msgs: Msgs, index: int) =
(3)     if #id (hd msgs) = index
(4)     then true
(5)     else ifMsgArrived ((tl msgs), index)
```

Algorithm 4: Function ifMsgArrived.

The *Broker* submodel under the in-order option is shown in Figure 10, in which we use the place *Queue_Index* instead of *Msg_Vis* to distinguish the two different models (the differences are emphasized in bold frame), and the tokens in place *Queue_Index* are used to indicate the oldest message ID implicitly.

Under the in-order option, the dequeue operation is modeled by the transition *Get_Oldest_Msg*. Additionally, we set an occurring condition for dequeue operation, that is, a guard function for the transition, as shown in Algorithm 4. This function is set to handle the case that message with larger sequence number (newer message) arrives earlier in queue than the message with smaller sequence number (older message). In such situation, the operation has to wait for a while until the corresponding message indicated by queue-index is sent to the queue. As a result, we can infer that the message delivery time under the in-order option is certainly longer than the opposite option. We can quantitatively measure this tradeoff between message consistency and system performance (i.e., extra time delay) according to the simulation results illustrated in the next section.

7. Performance Analysis

Performance is often a central issue in the design, development, and configuration of systems. It is not always enough to know that systems work properly, and they must also work effectively. Performance analysis can help to evaluate existing or planned systems, to compare alternative configurations, or to find an optimal configuration of a system. Notice that all parameters used in simulation are chosen for the purpose of illustration.

7.1. Parameters Setting and Simulation Design. In order to validate the performance of our model for CMQSs, we use MATLAB R2014a to test the experiments of the mathematical queueing model, and we compare the analytical results with the simulation outcomes of the CPN model, which is simulated using CPN Tools 4.0. We assume that the interval message production time is exponentially distributed with intensity parameter $\lambda = 30.3$, which means in average every 33 milliseconds (short as ms) a message is produced. Let the service time follow exponential distribution and let the mean service rate be $\{\mu_1 = 10, \mu_2 = 10\}$, where μ_1 and μ_2 are for

Function description: calculate and gather data related with message transmission
Source code:

```
(1) fun obs (bindelem) =
(2)     let
(3)         fun obsBindElem (Recv_Proc'Deliver (1, {msg, recvtime})) = Real.fromInt (recvtime − (#AT msg))
(4)           | obsBindElem_ = ~1.0
(5)     in
(6)         obsBindElem bindelem
(7)     end
```

Algorithm 5: Observer function of monitor *Msg_Delivery_Latency*.

senders and brokers, respectively. In other words, the mean service time for each message is 100 ms in both senders and brokers.

Speaking of CPN model, it is possible to investigate the behaviour of the modelled system using simulation, and to conduct simulation-based performance analysis [18]. Because of the uncertainty of the simulation output, it is very necessary to make detailed simulation solution before the experiments are carried out, and appropriate statistical techniques are also needed to guarantee the correctness and rationality of the simulation output data. For more flexibility, in our experimental scheme, a function is designed to control the variables used in simulation: *simulateConfigs (n: int)*, where the parameter n represents the repeat times of simulation. In this function, we can set system parameters such as mean service time and the number of threads, which facilitates the simulation process. In our simulation scheme, we design 26 monitors, which are the facilities used to collect simulation data or control the simulation execution process. For example, we use a monitor of breakpoint type to control the stop criteria of simulation, for example, setting the simulation time at 100 seconds.

Besides, to gain the performance metrics, some calculations have to be made using monitors. The monitors of data collection type are the most frequently used type. For instance, in order to calculate the whole message delivery time which is the most concerning metric for decision makers, we design the observer function for the monitor *Msg_Delivery_Latency*, as shown in Algorithm 5. This function will be executed every time when the transition *Deliver* (see the *Recv_Proc* submodel) fires; that is, a message is accepted by a receiver successfully, and then the delivery time of this message will be calculated using the current system time (variable *recvtime* in the function) minus the message generation time (variable *#AT msg* in the function). Each of the data for message delivery time will be collected as output of this monitor; using these raw data, we can get the final statistics that we are interested in.

7.2. Analysis of Numerical Results. Using the parameters set above, we present the reached numerical results in this section. Notice that we run 20 independent replications of simulation for each experiment and take their average value to make sure that the data collected during the simulation is independent and identically distributed (IID).

Figure 11 illustrates the average message delivery time with a different number of threads (i.e., server windows) $n_1 = n_2 = 4, 5, \ldots, 10$, where n_1 is the number in sender and n_2 is the number in broker. The reason $n_i \geq 4$, $i = 1, 2$, is to satisfy the constraint condition of system steady state: $\rho_i < 1, i = 1, 2$. The results of analytical model and CPN model are shown in the graph as symbols and lines, respectively. Two different values of reliability probabilities, $q = 99\%$, $q = 88\%$ (i.e., the probability of message loss is $p = 1\%$, $p = 12\%$, resp.), are used. As can be seen in Figures 11(a) and 11(b), the results obtained by solving the analytical model agree very well with those obtained by simulation under out-of-order option.

For the same input variables and the same values of reliability probabilities, Figure 12 shows the mean number of messages waiting in queue in case n_1 is not equal to n_2. More specifically, Figure 12(a) shows the average waiting number when n_1 is fixed with the value of 6 and n_2 is increasing by 1 starting with 4; Figure 12(b) shows the results with different n_1 while n_2 is unchanged with the value of 6. Similar to above, the solid lines relate to different message consistency options for the simulation results while the symbols relate to the analytical results. As the number of threads increases, the mean number of messages waiting in queue decreases sharply at first and then smoothly up to the threads number of around $n_1 = 7$ or $n_2 = 7$, when it begins to stabilize. As could be expected, the agreement between the calculation results of mathematical model and the simulation results of CPN model is very well under out-of-order option, which validates our modeling method, that is, the VMA approach for message queueing services in the cloud with reliability guarantee using CPN.

In addition, we compare the results of performance metrics obtained between two different message consistency options, as shown in Figures 11 and 12. In both figures, the red line corresponds to the results under in-order option and the blue line corresponds to the results under out-of-order option. As can be confirmed from the graphs, either the mean message delivery time or the mean number of messages waiting in queue is larger in the case of in-order option than in the case of out-of-order option. This is because, under in-order option, the broker has to send messages according to their sequence numbers; thus, if some messages with larger sequence numbers arrive earlier than those with

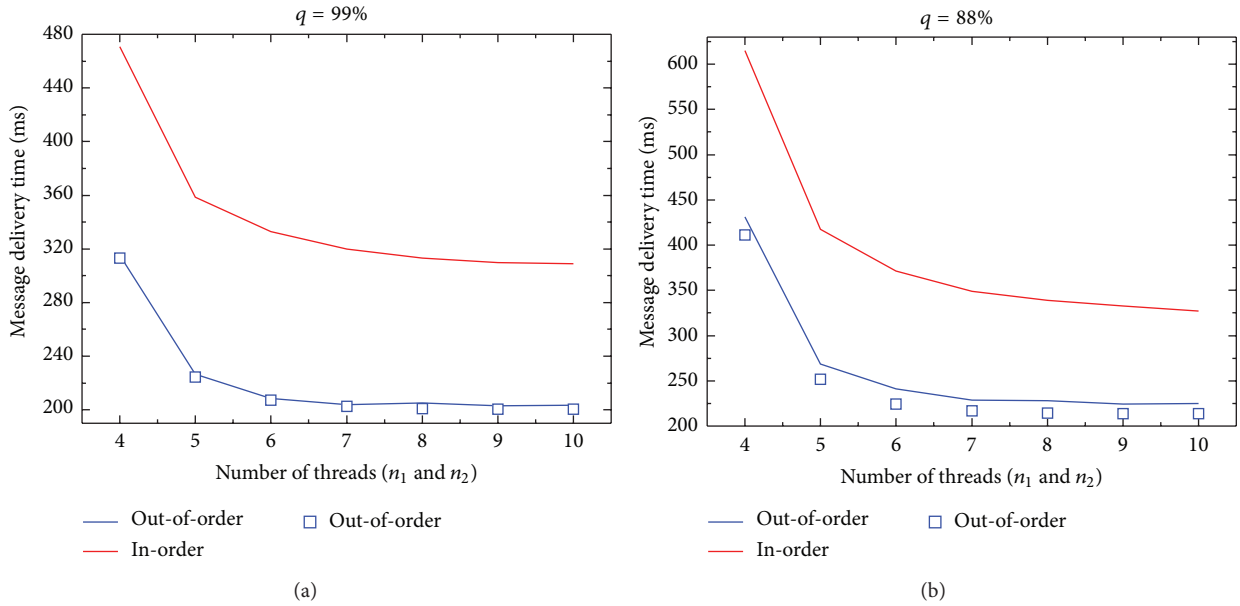

FIGURE 11: Mean message delivery time with a different number of threads.

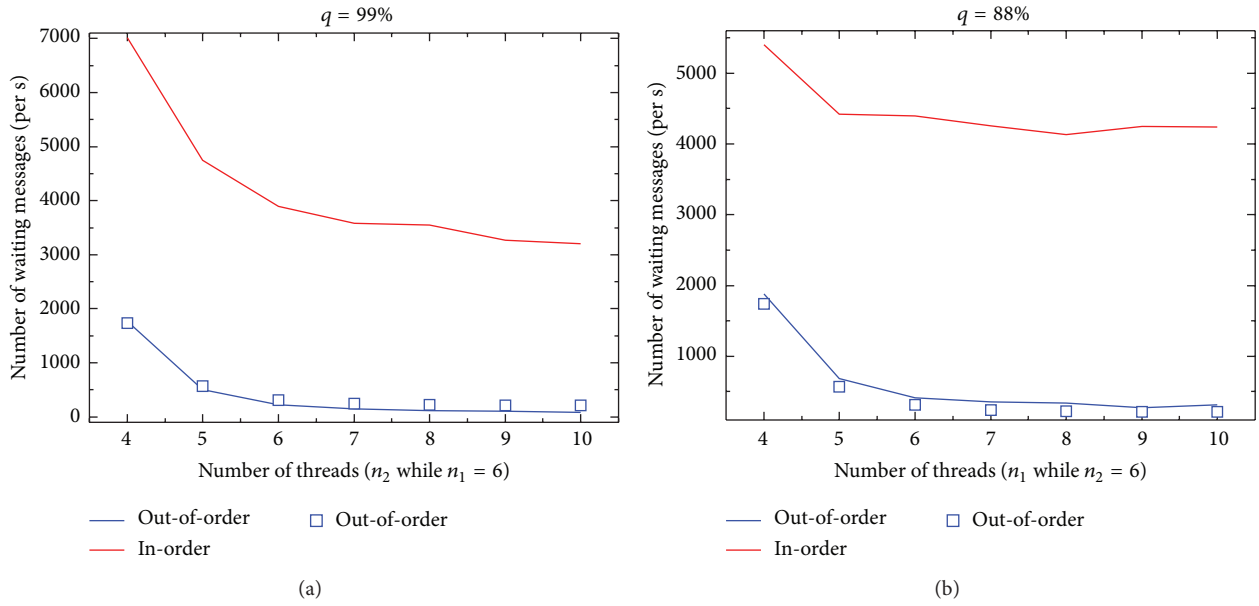

FIGURE 12: Mean number of messages waiting in queue with a different number of threads.

smaller sequence numbers, they would have to wait for a long time before they can be handled. Moreover, once some messages happen to be lost due to failure, the delivery latency of the following messages tends to be even longer. Also, it can be significantly noticed that the curves for in-order line and out-of-order line are nearly equidistant in each graph, which reveals that there is an approximate linear dependence among them. The results help us to do quantitative analysis of the tradeoff between system performance and message consistency, in which the performance of out-of-order option is about 50% higher than that of in-order option. In addition, by comparing the two graphs in Figure 12, we can notice that, in the case of in-order option (i.e., the red line), the mean number of waiting messages is stabilized at around 3000 per second in the upper graph while this number is stabilized at around 4000 per second in the bottom graph, which is relatively larger than the previous one. Thus, we can infer that, under in-order option, increasing the number of threads

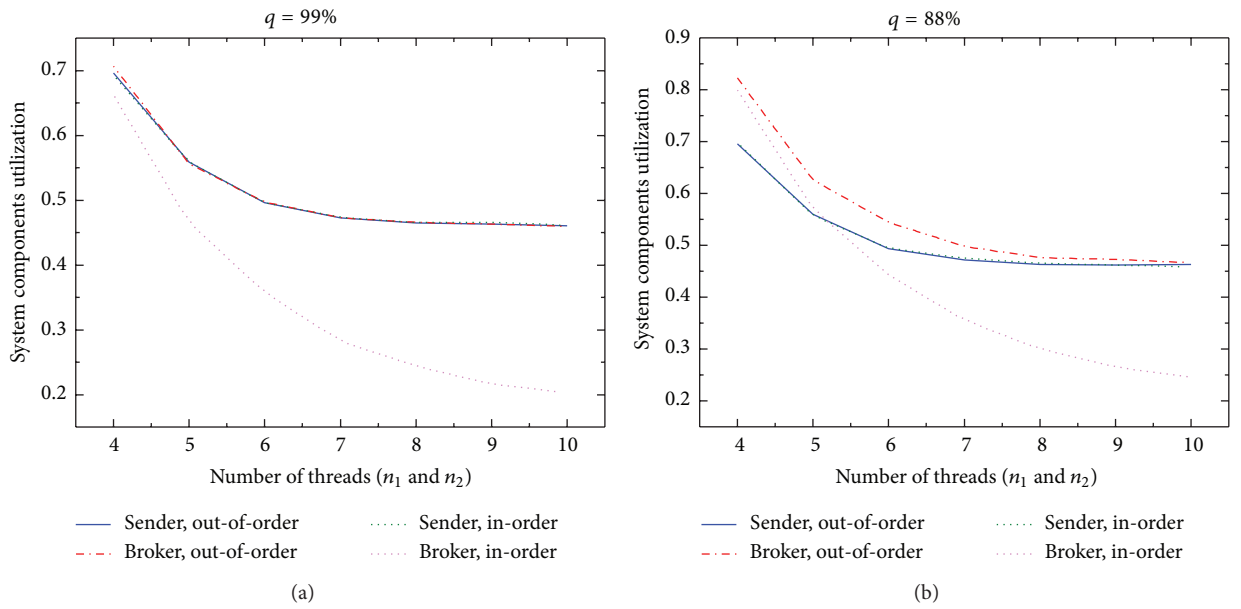

FIGURE 13: System components utilization with a different number of threads.

in broker (i.e., the parameter n_2) is more helpful than that in sender in decreasing the waiting number of messages as well as the message delivery time. This finding can ensure the service providers to put emphasis on the improvement of broker capability rather than sender.

We also analyze and compare the system component utilization in both senders and brokers through simulation, as shown in Figure 13. The graphs show, at a fixed number of threads, the sender utilization tends to remain the same regardless of the message consistency options or the values of reliability probabilities (i.e., the lines related to sender utilization are overlapped). However, the broker utilization rate varies a lot according to different simulation configurations. With $q = 99\%$, increasing the number of threads in senders and brokers reduces the broker utilization, and the decreased amplitude is nearly the same as that of the sender utilization under out-of-order option (i.e., the thinner lines), while the amplitude is much larger under in-order option (i.e., the thicker lines). This is because when the broker sends messages according to their sequence numbers, the broker has to wait for an extra period of time in the case that newer messages (i.e., messages with larger sequence numbers) arrive earlier in the queue than older messages, which leads to an increased idle time. Once the older message has arrived, several messages will begin to be processed in the same time, which leads to an increase in the parallel degree of processing. As the utilization is an estimation of the percentage of system busy time, with the increased idle time and reduced busy time, the broker utilization is much lower under the in-order options. With the reliability probability $q = 88\%$, more messages are triggered and sent again by the broker; thus the utilization of broker is higher than sender. However, under in-order option, for the same reason, the broker utilization turns to be lower than sender when the number of threads increases to a certain point, that is, $n_i > 5$, $i = 1, 2$. These observations can help the cloud message queueing service providers to regulate the limited resources more effectively to minimize their costs as well as maximize their commercial interests.

Moreover, we also calculate the three performance metrics under different service times while fixing the average arrival rate at 30 messages per second. The reliability probability is set as $q = 99\%$ under out-of-order consistency option and the results are shown in Figure 14. According to Figure 14(a), it can be observed that longer service time leads to longer delivery time, and for each n_i (i.e., the number of threads), the delivery time is linearly increasing when mean service time is less than a particular value; for example, when the number of threads is 5, the value is 100 ms while, for a number of 7, the value is 140 ms. Also, we can see that larger n_i can result in an improvement in the performance of delivery time when mean service time is larger than 100 ms. Figure 14(b) shows, before certain points (e.g., 80 ms for service time when $n_i = 5$, or 120 ms for service time when $n_i = 7$), increasing service time almost has no effect on the number of waiting messages, while afterwards, the number increases sharply. These turning points can help to warn the cloud service providers to pay attention to the system resource consumption and replan the resource allocation when necessary. Figure 14(c) illustrates that the sender utilization and the broker utilization both grow sharply at first and then rise smoothly as service time gets longer. Also, the relatively steady period (i.e., the utilization rate remains at around 47%) stays longer as the number of threads n_i increases.

We also examine the effects of message arrival rate on total delivery time, mean number of waiting messages, and components utilization, given that the mean service time is unchanged at 100 ms; that is, service rate per second is $\mu_1 = \mu_2 = 10$. Unlike the above experiment, we set the reliability probability at $q = 88\%$ for a change. Figure 15 presents the

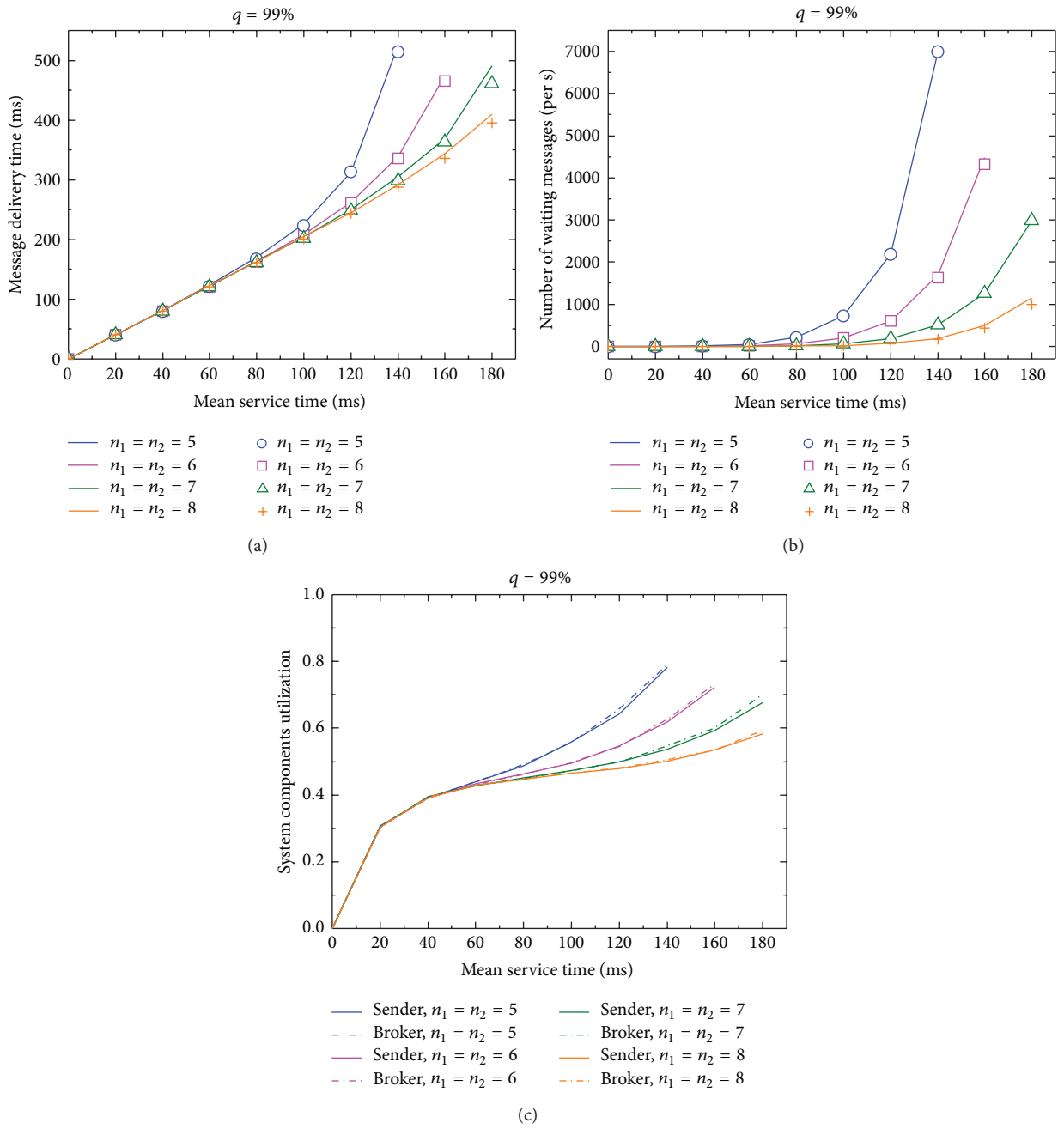

FIGURE 14: Performance metrics with different service time, for $\lambda = 30$ and $q = 99\%$.

performance metrics. From Figure 15(a), it can be seen that the message delivery time increases slowly before a steep rise for any n_i; this is because when the arrival rate increases, the system workload traffic (i.e., the parameter ρ_i) increases as well. The closer ρ_i is to 1, the greater the traffic intensity will be, which leads to a surprising congestion in queue, thus further resulting in the abrupt rise of delivery time. The congestion also means a larger number of messages waiting to be processed in system with respect to increased arrival rate, as shown in Figure 15(b). Similar to the above experiment, the utilization of sender and broker which are depicted in Figure 15(c) increases as the arrival rate gets larger, except that the utilization of broker is always a little higher than the sender because of low reliability probability.

8. Conclusions

Cloud has been identified to have many advantages and realizing the full benefits of this new paradigm will require rethinking the way we build applications. As the progressive improvement of cloud-based message queueing services,

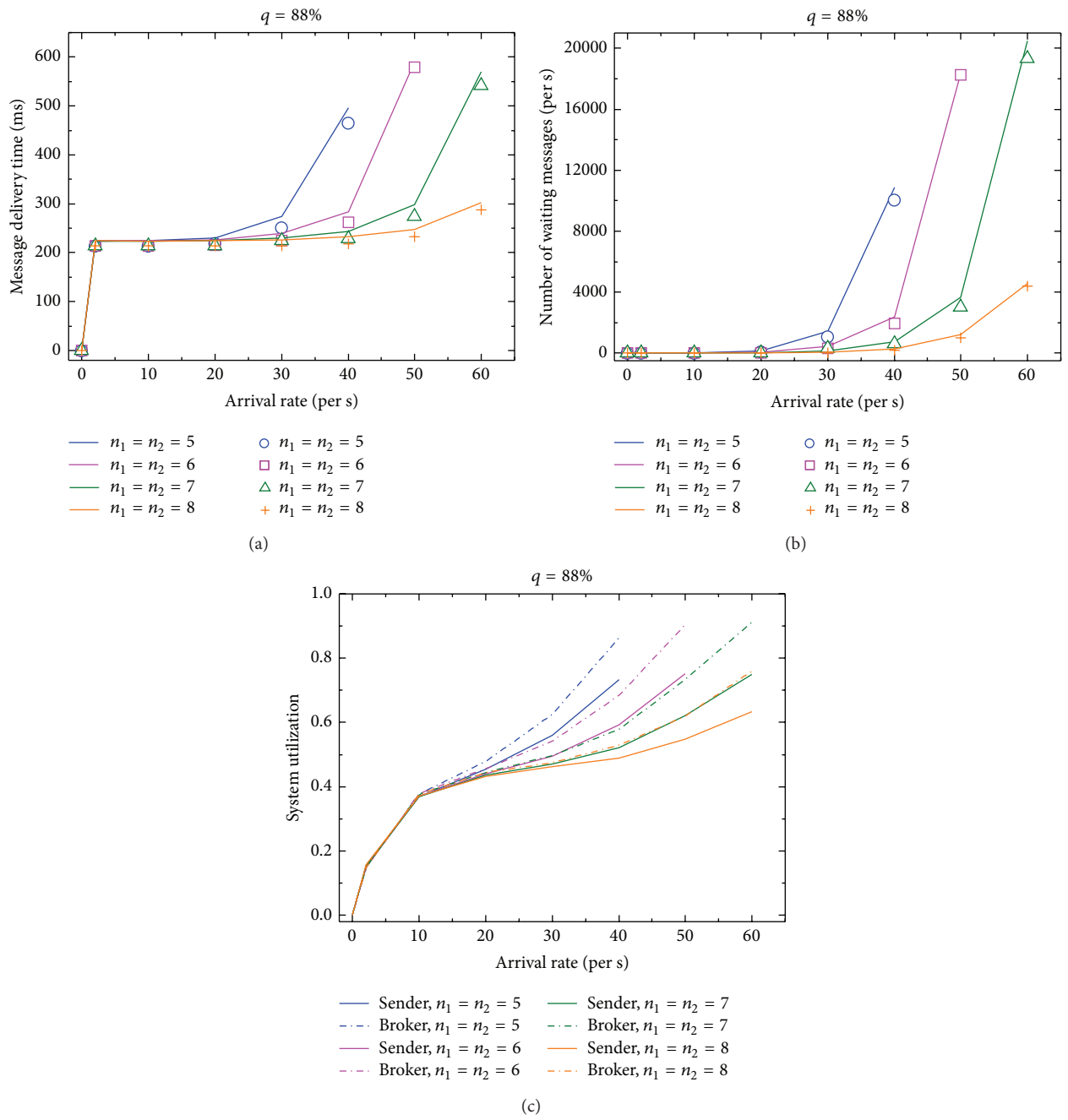

FIGURE 15: Performance metrics with different arrival rate, for $\mu_1 = \mu_2 = 10$ and $q = 88\%$.

performance analysis has become a crucial interest for both cloud providers and cloud consumers before its universal usage in the industrial area. In this paper, we present both analytical and simulation modeling methods to analyze the performance of CMQSs with reliability guarantee. The mathematical model is presented using queueing theory and Markov process, while the simulation model is created employing colored Petri nets. Considering different message consistency requests, we develop two simulation models dealing with in-order message delivery and out-of-order delivery, respectively. We name our novel modeling approach as VMA in virtue of the visibility-timeout mechanism which is adopted in system to enhance reliability. We examine the effects of various parameters such as message arrival rate, broker service rate, message consistency options, and number of resources on the system performance metrics of interest, for example, message delivery time, waiting number of messages, and system components utilization. We also quantitatively analyze the tradeoff between system performance and message consistency. Numerical and simulation

results show that the presented models are able to model the performance of CMQSs with reliability guarantee, which is not emphasized in other researches.

Future work will consider other service time distributions such as general distribution or Erlang distribution, which makes the model more flexible and realistic in cloud environment, as well as increasing the diversity of service time. We would also like to model differentiated quality of service requirements for classified messages, aiming at improving the performance of CMQSs in multiple levels.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is sponsored by the National Basic Research Program of China (2009CB320505), China 863 Program (2011AA01A102), the Fundamental Research Funds for the Central Universities (2013RC0502), and the Scientific Research Foundation of Qiongzhou University (QYQN201331).

References

[1] W. Emmerich, M. Aoyama, and J. Sventek, "The impact of research on the development of middleware technology," *ACM Transactions on Software Engineering and Methodology*, vol. 17, p. 48, 2008.

[2] Apache ActiveMQ, June, http://activemq.apache.org/index.html.

[3] Rabbit MQ, http://www.rabbitmq.com/features.html.

[4] The Kafka Project, http://www.kafka.org/.

[5] Message Queuing (MSMQ), https://msdn.microsoft.com/en-us/library/ms711472%28v=vs.85%29.aspx.

[6] M. Armbrust, A. Fox, R. Griffith et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[7] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2009.

[8] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, "Cloud computing: distributed internet computing for IT and scientific research," *IEEE Internet Computing*, vol. 13, no. 5, pp. 10–11, 2009.

[9] Amazon Simple Queue Service, http://docs.aws.amazon.com/zh_cn/AWSSimpleQueueService/latest/SQSDeveloperGuide/Welcome.html.

[10] Windows Azure Queue, June, http://download.microsoft.com/download/5/2/D/52D36345-BB08-4518-A024-0AA24D47BD12/Windows%20Azure%20Queue%20-%20Dec%202008.docx.

[11] K. Sachs, S. Kounev, and A. Buchmann, "Performance modeling and analysis of message-oriented event-driven systems," *Software and Systems Modeling*, vol. 12, no. 4, pp. 705–729, 2013.

[12] L. L. Ferreira, M. Albano, and L. M. Pinho, "QoS enabled middleware for real-time industrial control systems," in *Proceedings of the IEEE 18th International Conference on Emerging Technologies & Factory Automation (ETFA '13)*, pp. 1–4, September 2013.

[13] C. U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, Reading, Mass, USA, 1990.

[14] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: a survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[15] H. Koziolek, "Performance evaluation of component-based software systems: a survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[16] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FoSE '07)*, pp. 171–187, Minneapolis, Minn, USA, May 2007.

[17] K. Jensen, L. M. Kristensen, and L. L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, Springer, Berlin, Germany, 2009.

[18] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3-4, pp. 213–254, 2007.

[19] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Proceedings of the 6th International Workshop on Software and Performance (WOPS '07)*, pp. 54–65, ACM, Buenos Aires, Argentina, February 2007.

[20] J. Happe, H. Friedrich, S. Becker, and R. H. Reussner, "A pattern-based performance completion for message-oriented middleware," in *Proceedings of the 7th International Workshop on Software and Performance (WOSP '08)*, pp. 165–176, Princeton, NJ, USA, June 2008.

[21] M. Woodside, D. Petriu, and K. Siddiqui, "Performance-related completions for software specifications," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pp. 22–32, Orlando, Fla, USA, May 2002.

[22] C. Rathfelder and S. Kounev, "Modeling event-driven service-oriented systems using the palladio component model," in *Proceedings of the 1st International Workshop on Quality of Service-Oriented Software Systems*, pp. 33–38, Amsterdam, The Netherlands, 2009.

[23] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric performance completions for model-driven performance prediction," *Performance Evaluation*, vol. 67, no. 8, pp. 694–716, 2010.

[24] S. Kounev, S. Spinner, and P. Meier, "Introduction to queueing petri nets: modeling formalism, tool support and case studies," in *Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE '12)*, pp. 9–18, April 2012.

[25] A. Piórkowski and J. Werewka, "Minimization of the total completion time for asynchronous transmission in a packet data-transmission system," *International Journal of Applied Mathematics and Computer Science*, vol. 20, no. 2, pp. 391–400, 2010.

[26] N. L. Tran, S. Skhiri, and E. Zimányi, "EQS: an elastic and scalable message queue for the cloud," in *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom '11)*, pp. 391–398, Athens, Greece, December 2011.

[27] H. Chen, F. Ye, M. Kim, and H. Lei, "A scalable cloud-based queuing service with improved consistency levels," in *Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems (SRDS '11)*, pp. 229–234, October 2011.

[28] A. Lakshman and P. Malik, "Cassandra—a decentralized structured storage system," *Operating Systems Review (ACM)*, vol. 44, pp. 35–40, 2010.

[29] Z. Zhang, Y. Wang, H. Chen, M. Kim, J. M. Xu, and H. Lei, "A cloud queuing service with strong consistency and high availability," *IBM Journal of Research and Development*, vol. 55, no. 6, pp. 10:1–10:12, 2011.

[30] WebSphere eXtreme Scale, http://www-03.ibm.com/software/products/en/websphere-extreme-scale.

[31] R. C. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," *BMC Bioinformatics*, vol. 11, no. 12, article S1, 2010.

[32] C. Zhang and X. Liu, "HBaseMQ: a distributed message queuing system on clouds with HBase," in *Proceedings of the 32nd IEEE Conference on Computer Communications (INFOCOM '13)*, pp. 40–44, Turin, Italy, April 2013.

[33] D. Patel, F. Khasib, I. Sadooghi, and I. Raicu, "Towards in-order and exactly-once delivery using hierarchical distributed message queues," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14)*, pp. 883–892, IEEE, Chicago, Ill, USA, May 2014.

[34] I. Sadooghi, S. Palur, A. Anthony et al., "Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14)*, pp. 404–413, Chicago, Ill, USA, May 2014.

[35] P. Szilágyi, "Iris: a decentralized approach to backend messaging middlewares," *Computer Science and Information Systems*, vol. 11, no. 2, pp. 549–567, 2014.

[36] T. Murata, "Petri nets: properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[37] L. C. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1996.

[38] S. Choosang and S. Gordon, "A coloured Petri Net methodology and library for security analysis of network protocols," *Journal of Computers*, vol. 9, no. 2, pp. 243–256, 2014.

[39] L. He, C. Huang, K. Duan et al., "Modeling and analyzing the impact of authorization on workflow executions," *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1177–1193, 2012.

[40] S. H. Zegordi and H. Davarzani, "Developing a supply chain disruption analysis model: application of colored Petri-nets," *Expert Systems with Applications*, vol. 39, no. 2, pp. 2102–2111, 2012.

[41] K. M. Ng, M. B. I. Reaz, and M. A. M. Ali, "A review on the applications of Petri nets in modeling, analysis, and control of urban traffic," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 2, pp. 858–870, 2013.

[42] W. M. van der Aalst, C. Stahl, and M. Westergaard, "Strategies for modeling complex processes using colored Petri Nets," in *Transactions on Petri Nets and Other Models of Concurrency VII*, pp. 6–55, Springer, 2013.

[43] K. Jensen, "An introduction to the theoretical aspects of coloured Petri nets," in *A Decade of Concurrency Reflections and Perspective*, vol. 803 of *Lecture Notes in Computer Science*, pp. 230–272, Springe, Berlin, Germany, 1994.

[44] C. Lu, *Queueing Theory*, Beijing University of Posts and Telecommunications Press, 2nd edition, 2009.

[45] N. Tanabe and A. Ohta, "Network interface architecture with scalable low-latency message receiving mechanism," *IEICE Transactions on Information and Systems*, vol. E96D, no. 12, pp. 2536–2544, 2013.

[46] T. Pongthawornkamol, K. Nahrstedt, and G. Wang, "Probabilistic QoS modeling for reliability/timeliness prediction in distributed content-based publish/subscribe systems over best-effort networks," in *Proceedings of the 7th International Conference on Autonomic Computing*, Washington, DC, USA, 2010.

[47] S. Kounev, "Performance modeling and evaluation of distributed component-based systems using queueing Petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.