

Stochastic Reo: a Case Study

Y.-J. Moon¹, F. Arbab, A. Silva, C. Verhoef A. Stam

CWI, Amsterdam & Almende BV, Rotterdam, The Netherlands

Abstract

QoS analysis of coordinated distributed autonomous services is currently of interest in the area of service-oriented computing and calls for new technologies and supporting tools. In previous work, the first three authors have proposed a compositional automata model to provide semantics for stochastic Reo, a channel based coordination language that supports the specification of QoS values (such as request arrivals or processing rates). Furthermore, translations from this automata model into stochastic models, such as continuous-time Markov chains (CTMCs) and interactive Markov chains (IMCs) have also been presented.

Based on those results, we describe in this paper a case study of QoS analysis. We analyze a certain instance of the ASK system, an industrial software system for connecting people offering professional services to clients requiring those services. We develop a model of the ASK system using stochastic Reo. The distributions used in this model were obtained by applying statistical analysis techniques on the raw values that we obtained from the real logs of an actual running ASK system. These distributions are used for the derived CTMC model for the ASK system to analyze and to improve the performance of the system, under the assumption that the distributions are exponentially distributed. In practice, this is not always the case. Thus, we also carry out a simulation-based analysis by a Reo simulator that can deal with non-exponential distributions. Compared to the analysis on the derived CTMC model, the simulation is approximation-based analysis, but it reveals valuable insight in the behavior of the system. The outcome of both analyses helps both the developers and the installations of the ASK system to improve the performance of the system.

Keywords: Stochastic Reo, QoS analysis, case study, Extensible Coordination Tools

1 Introduction

The increasing complexity of software has motivated much research in order to develop techniques for the modular development of systems. Component-based software engineering and service-oriented computing aim at the development of reusable software components and/or services as building blocks that can be composed to build different applications. Research on software composition plays a key role in this quest, as it offers flexible ways of plugging components together. Connector based-languages, where channels or connectors are used to compose components and services into a system play a prominent role in the world of software composition. One of such languages is

¹ Email: yjm@cwi.nl

Reo [2,3], which offers a model of component and service coordination, wherein complex connectors are constructed by composing various types of primitive connectors called channels.

QoS analysis of composed software (intensive) systems has become popular in the last few years, with the goal of evaluating and improving performance and resource allocation in service-oriented applications.

Stochastic Reo [13] is an extension of Reo which allows for the specification of stochastic values for the channels (e.g., arrival and processing rates). A compositional automata model of Stochastic Reo was proposed in [13] and translations from this automata model to stochastic models such as CTMCs and IMCs were presented.

In this paper we show how the theory developed in previous papers, implemented as tools, can be used to model a part of a real industrial system, perform QoS analysis, and help the developers get an insight into the system behavior, which enables to improve the performance of the system. We model and analyze the ASK system, a software system developed by the Dutch company Almende, which provides efficient matching between service providers and clients. An example of the application of the ASK system consists of a service-based system running in a call center that matches calling clients with the appropriate representatives that can provide them with the specialized customer service that they need.

One challenge that arises when installing particular instances of the ASK system is how to allocate resources, which are typically scarce or expensive. For instance, in the particular example above, the call center wants to have an optimal distribution of its operators' schedules in order to reduce waiting time for the customers without increasing enormously its personnel costs. A stochastic model of the ASK system can be used to perform analysis and provide advice to solve such problems.

The main contributions of this paper are the following:

- (i) a stochastic Reo model of the ASK system². The distributions in this model were obtained by statistical analysis of real values filtered out of the logs of an actual running ASK system.
- (ii) analysis of several interesting properties using the probabilistic model checker PRISM [12,15] which allowed to produce suggestions for the performance improvement of the ASK system. This analysis is done on a CTMC obtained from the Reo model.
- (iii) analysis of the system using a simulator which enables the study of properties involving non-exponential distributions (CTMCs can deal only with exponential distributions).

² Details available at <http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/CaseStudies/SimulatoronASK/Reception>.

2 Preliminaries

Overview of Reo

Reo is a channel-based coordination model wherein so-called *connectors* are used to coordinate (i.e., control the communication among) components or services *exogenously* (from outside of those components and services). In Reo, complex connectors are compositionally built out of primitive channels. Channels are atomic connectors with exactly two ends, which can be either *source* or *sink* ends. Source ends accept data into, and sink ends dispense data out of their respective channels. Reo allows channels to be undirected, i.e., to have respectively two source or two sink ends.



Fig. 1. Some basic Reo channels

Fig. 1 shows the graphical representations of some basic channel types. The **Sync** channel is a directed, unbuffered channel that synchronously reads data items from its source end and writes them to its sink end. The **LossySync** channel behaves similarly, except that it does not block if the party at the sink end is not ready to receive data. Instead, it just loses the data item. **FIFO1** is an asynchronous channel with a buffer of size one. The **SyncDrain** channel differs from the other channels in that it has two source ends (and no sink end). If there is data available at both ends, this channel consumes (and loses) both data items synchronously.

Channels can be joined together using nodes. A node can have one of three types: *source*, *sink* or *mixed* node, depending on whether all ends that coincide on the node are source ends, sink ends or a combination of both. Source and sink nodes, collectively called *boundary nodes*, form the boundary of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, and sink nodes as mergers. A mixed node combines both behaviors by atomically consuming a data item from one sink end and replicating it to all of its source ends.

An example connector is depicted in Fig. 2. It reads a data item from a , buffers it in a **FIFO1** and writes it to c . The connector loses data items from a if and only if the **FIFO1** buffer is already full. This construct, therefore, behaves as a connector called (overflow) **LossyFIFO1**.

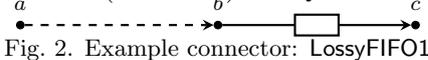


Fig. 2. Example connector: LossyFIFO1

Stochastic Reo

Stochastic Reo is an extension of Reo where channel ends and channels are annotated with stochastic values for *data arrival rates* at channel ends and *processing delay rates* on channels. Such rates are, e.g., non-negative real values that describe how the probability that an event occurs varies with time. Fig. 3 shows the stochastic versions of the primitive Reo channels in Fig. 1.

Here and throughout, for simplicity, we omit the node names, since they can be inferred from the names of their respective arrival rates: for instance, γ_a is the arrival rate of node a .

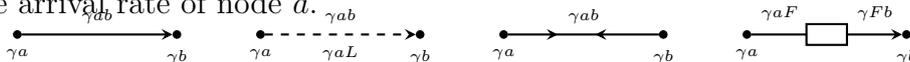


Fig. 3. Stochastic Reo channels corresponding to the channels in Fig. 1

A processing delay rate represents how long it takes for a channel to perform a certain activity, such as data-flow. For instance, a **LossySync** has two associated rates γ_{ab} and γ_{aL} for, respectively, successful data-flow from node a to node b , and losing the data item from node a . In a **FIFO1** γ_{aF} represents the delay for data-flow from its source a into the buffer, and γ_{Fb} for sending the data from the buffer to the sink b .

Arrival rates describe the time between consecutive arrivals of I/O requests at the source and sink nodes of Reo connectors. For instance, γ_a and γ_b in Fig. 3 are the associated arrival rates of write/take requests at the nodes a and b .

Since arrival rates on nodes model their interaction with the environment only, mixed nodes have no associated arrival rates. This is justified by the fact that a mixed node delivers data items instantaneously to the source end(s) of its connected channel(s). Hence, when joining a source with a sink node into a mixed node, their arrival rates are discarded. A more precise description of Stochastic Reo appears in [4,13]. A stochastic version of the **LossyFIFO1** is depicted in Fig. 4, including its arrival and processing delay rates.

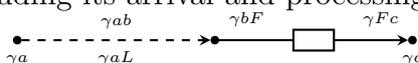


Fig. 4. Stochastic LossyFIFO1

3 ASK system

The “Access Society’s Knowledge” (ASK) system [16] is an industrial software developed by the Dutch company Almende [1], and marketed by their daughter company ASK Community Systems [5]. The ASK system is a communication software product that acts as a mediator between service consumers and service providers, for instance, connecting rescue institutions (e.g., fire departments) and professional volunteers. The connection established by the ASK system is provided by mechanisms for matching users requiring information or services with potential suppliers. For this purpose, the matching mechanisms use the profiles and availability offered by people who provide or require services.

The main goal of the ASK system is to do the matching in an efficient way. To achieve that, the system collects feedback on the quality of services after the connection. Such feedback is used to decide better connections for the subsequent requests of the same type. In addition, the system uses self-learning and self-organizing mechanisms by continuously updating to users’ preferences and available resources. Moreover, the ASK system enables users to inform

others about their status, their availability, and how they can be contacted best. This information is used to select the right people for a communication session as well as the feedback.

To offer efficient connections, the ASK system considers:

- human knowledge and skills of service providers
- time schedules of the provision of services
- communication media such as telephones, SMS, and emails

When people request a certain service from specialists or service providers, the ASK system attempts to select the best possible service provider. This selection is based on the rating of the knowledge and the skills of service providers who are available at that moment. This rating, in turn, is based on the feedback on the quality of services offered by the service providers. The occurrences of events can follow either regular schedules or ad-hoc schedules. The ASK system deals with both of these situations while satisfying the constraints and the purposes of users' requests.

The ASK system generally considers the telephone as a primary communication medium, but other means of communication, such as email or SMS, are also supported. These types of media must be considered according to the reachability and the preferences of the users. For example, people can have more than one email address and telephone number, with different associated usage constraints and user preferences. Such information must be indicated in the system to allow for efficient connections.

The ASK system acts as an agent that connects service providers and service consumers in an *efficient* way, handling multitudes of such connections simultaneously at any given time. The ASK system has a hierarchical modular architecture, i.e., it consists of a number of high-level components, which in turn consist of lower-level components, etc., running as threads. In order to handle massive numbers of connections concurrently, the components need to utilize multiple threads that provide the same functionalities. In this setting, allocation of system resources, e.g. the number of threads, to various components plays a critical role in the performance and responsiveness of an installed system in its actual deployment environment (e.g., properties of servers, available telephone lines, call traffic, available human operators, etc.), but determining the proper resource allocations to provide a good performance is far from trivial. Deriving and analyzing a stochastic model for an installed ASK system provides valuable input and insight for improving its performance. Among other possibilities, such a model allows system architects and installation operators to play what-if games by changing various resource and demand parameters and discover how a deployed system would perform under such scenarios, in order to adjust and fine-tune the system for cost-effective, optimal performance.

Various methods for performance evaluation have been suggested. Rigorous methods require mathematical models of a system involving variables that represent the parameters relevant to its behavior. Stochastic variables describe random system behavior, leading to more realistic models of behavior than their deterministic counterparts. CTMCs are frequently used to model such systems and their features and efficient closed-form and numerical techniques [18] exist for their analysis. Traditionally, such models are constructed by human experts whose experience and insight constitute the only link between the actual system and the resulting models.

Ideally, mathematical models for the analysis of the behavior of a system should be derived from the same (hopefully, verified correct) models used for its design and construction. Such automation makes the derivation of these models less error-prone, and ensures that a derived analytical model corresponds to its respective implemented system. An expressive modeling formalism that simultaneously reflects structural, functional, and QoS properties of a modeled system constitutes a prerequisite for this automation. Reo serves as an example of such a formalism: (1) it provides structural model elements whose composition reflects the composition of their counterpart system components with architectural fidelity; (2) it allows formal verification of functional and behavioral properties of a modeled system; (3) it supports derivation of executable code from its models; and (4) it supports derivation of mathematical models for the analysis of the QoS properties of systems.

A Reo model of the ASK System was developed as a case study [8] within the context of the EU project Credo [7] for verification of its functional properties. In the work we report in this paper, we refined and augmented this Reo model with stochastic delays extracted from actual system logs to derive a Stochastic Reo model for the ASK System. Together with Almende, we use this model to analyze and study the QoS properties of the ASK system in various settings. For instance, using the approach in [13], we derive CTMC models from the Stochastic Reo model of the interesting parts of the ASK System, and feed them into CTMC analysis tools, which enables us to do model checking of the stochastic behavior of the system. We will show the analysis of several such properties using PRISM in Section 5. The following sections describe the architecture of the ASK system in some detail. The figures and the descriptions we use here are based on [17].

3.1 Overview of the ASK system

The top-level architecture of the ASK System is shown in Fig. 5. Every component in this architecture has its own internal architecture, with several levels of hierarchical nesting. At its top-level, the ASK system consists of three parts: a *web front-end*, a *database*, and a *contact engine*. The *web front-end*

deals with typical domain data, such as users, groups, phone numbers, mail address, and so on. The *database* stores typical domain data, together with the feedback from users and knowledge from past experience. The *contact engine* handles the communication between the system and the outside world (e.g., by responding to or initiating telephone calls, SMS, emails, etc.) and provides appropriate matching and scheduling functionalities.

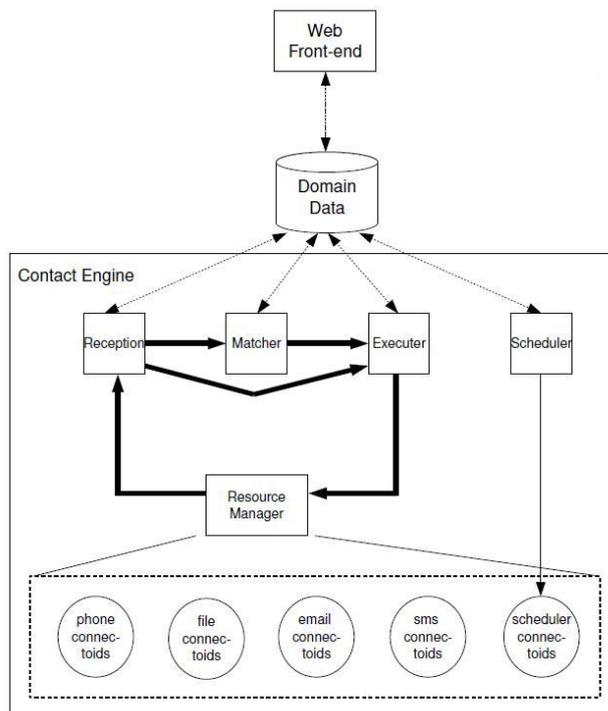


Fig. 5. Overview of ASK system

As mentioned above, the ASK system connects service providers and consumers for incoming requests. A connection is made when appropriate participants for a certain request are found. Until its proper connection is established, an incoming request loops through the system repeatedly as (sub-)tasks. This feature is called *Request loop* and it is represented by **thick arrows** in the contact engine in Fig. 5.

The contact engine consists of five components: *Reception*, *Matcher*, *Executer*, *ResourceManager*, and *Scheduler*. The *Reception* component determines which steps must be taken by the ASK system to fulfill a request. The *Matcher* component determines proper participants for fulfilling a request. The *Executer* component determines the best means of connection between the participants. The *Resource Manager* component either uses the Request loop for complicated requests or establishes direct connections between users for trivial requests. The *Scheduler* component, separated from the components within the request loop, schedules requests based on the time constraints of the requests in the database.

4 Modeling the ASK system

In this section, we consider the contact engine, which contains the Request loop, and focus specifically on the Reception component. The components in the contact engine have very similar architectures, thus, the analysis carried out here for the Reception component can be used for the other ones, as well.

4.1 The Reception component

The Reception component consists of multiple threads, the so-called *ReceptionMonks* (*RM*), which handle incoming requests using two different functions:

- **HostessTask (HT)** which converts incoming requests into tasks that will be put into the task queue outside of the Reception component;
- **HandleRequestTask (HRT)** which takes care of the communication flow, interacts with the database, and possibly generates new requests which are dealt with by the Matcher or the Executer component. For example, given an incoming request, HRT may ask questions from users by playing pre-recorded messages, obtain information such as menu item choices, account number, etc., punched in by the users, and store this information into the database. During this communication, new requests can be generated and sent to other components.

Each thread runs one of these two different functions/tasks exclusively. That is, if an RM thread runs the HT function, then it is forbidden to run the HRT function. This implies that the Reception component needs to have at least two threads, one for HT and the other for HRT. In general, HRT takes more time than HT, since it actually deals with incoming tasks. Thus, the Reception component needs more threads running HRT. For simplicity of modeling, we assume that every thread in the Reception component has only one function, e.g., either HT or HRT. Reflecting this simplification, Fig. 6 shows the Reception model drawn in the Eclipse Coordination Tools (ECT) [9]. This figure shows a Reception component with three RM threads, one with only HT and the other two with only HRT.

The RMHT and the indexed RMHRTs in Fig. 6 correspond to RM threads for a HT and HRT functions, respectively. Incoming requests are converted into tasks by the RMHT, and the converted tasks are stored in the task queue which is represented as a FIFO1 laid between RMHT and the indexed RMHRTs. The converted tasks are selected and handled by the RMHRTs. We model task selection as a non-deterministic choice at the *TQOut* node in Fig. 6, which will turn into a random process once we associate the distributions of the stochastic variables that describe the actual task mix of a running system, as extracted from its logs.

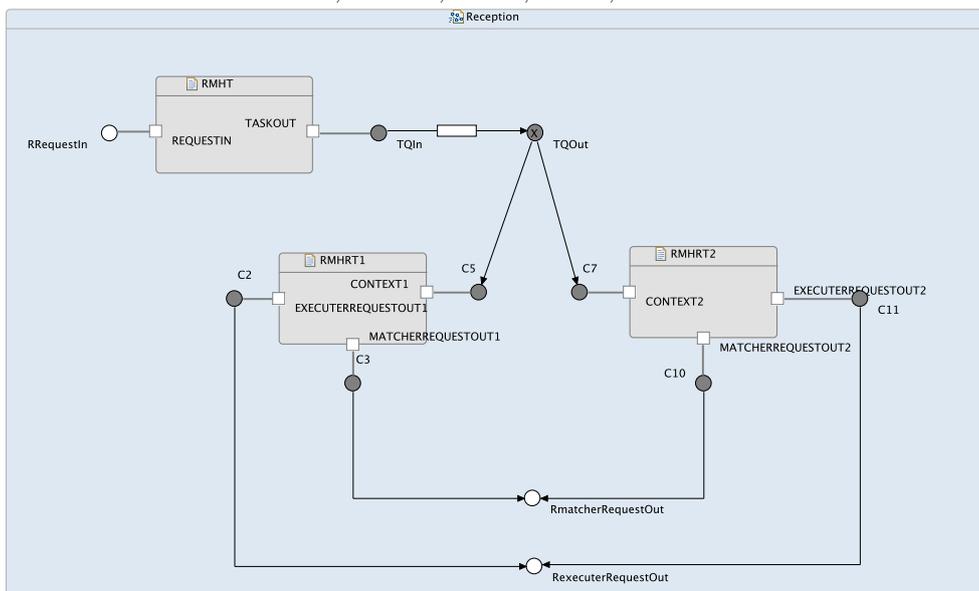


Fig. 6. Reception component in ECT

The graphical notation \otimes used for $TQOut$ in Fig. 6 is an abbreviation for an *exclusive router* [3] whose Reo circuit is depicted on the right. This circuit delivers an incoming data item at node a to either node b or node c , whichever one can accept it, and non-deterministically selects one when both can.

The non-deterministic choice is actually conducted by the merger d . Thus, the rates for the random selection apply to the merger d .

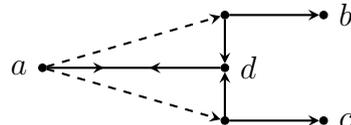


Fig. 6 serves as a basic template model for the Reception component. Depending on the specific properties of interest in each analysis, we adapt and vary this basic template slightly. For example, for the analysis of the properties of the task queue, we may substitute a **LossyFIFO1** connector for the FIFO1 channel, as shown below.

4.2 Extracting distributions from logs

A stochastic model of the ASK system requires the distributions for all activities in the system. To obtain these distributions, we applied statistical data-analysis techniques on the raw values extracted from the real logs of a running ASK system. The logs contained the data of 100 incoming calls. Those calls simultaneously resulted in 369 requests sent to the Reception component. The trace holds exact timings of all actions performed related to each process.

We need to determine the rates for request arrivals (RRequestIn) and processing delay at the Reception component, reading request arrivals from the Matcher (RmatcherRequestOut) and the Executer (RexecuterRequestOut). For this purpose, after a cleanup of the raw data by removing outliers and erroneous data, we determined the appropriate distributions, using statistical

tests (like the chi-square goodness-of-fit test).

For the Reo model, it is not important which type of distributions we obtain. However, to perform analysis using PRISM, which takes a CTMC as input, only exponential distributions can be used. In the case of request arrival rates, we may indeed assume that the inter-arrival times of the requests are exponentially distributed. This is reasonable since incoming calls to the ASK system are independent from each other, and the inter-arrival times are memoryless. However, in the case of processing delay rates, we were not able to conclude that the rates are exponentially distributed. The statistical tests showed that we may assume that the processing times follow a log-normal distribution.

5 QoS analysis

In this section, we show how to analyze the ASK system using both the CTMC and Reo Simulator approach. As mentioned in the previous section, the arrival or service times for some activities are not exponentially distributed. This is one of the reasons to analyze the Reo model with the Reo Simulator (see Section 5.2). The simulator was also used when we could not obtain any proper distribution from the logs at all. In this case, we used bootstrapping [14] in the simulator with the original data as special inputs in the simulator for the rates.

5.1 Analysis on the derived CTMC

In this section, we analyze the ASK system to reveal some of its interesting properties in order to both evaluate and obtain clues for improving its performance. We carry out our analysis on the CTMC model derived from the Stochastic Reo model of the ASK system. We then feed the derived CTMC model as input to PRISM. In PRISM, properties of models are expressed using operations such as P , S , and R operators: the P operator is used to reason about the probability of the occurrence of a certain event; the S operator is used to reason about the steady-state behavior of a model; the R operator is used to analyze reward-based properties. In addition, labels are used to concisely express the formulas representing the properties of a model. Specifically, we use the following labels to express some properties later.

- `num_dataLoss` represents the number of task-loss in the task queue.
- `run` represents the running status of the RMHRT thread.

In general, resources are neither infinite nor free. Thus, one needs to balance cost-effective resource utilization against most efficient performance, i.e., obtaining the best performance taking into account the limited resource. In the Reception component in Fig. 6, the resources of interest include:

- (i) the minimum capacity of the task queue
- (ii) the utilization and/or the performance of the RMHRT threads that handle tasks

5.1.1 Task queue

As mentioned above, RMHT merely converts incoming requests into tasks, but it does not actually handles the requests. In general, the conversion into tasks does not take long, whereas handling a request may take considerable time. Thus, if the task queue has a small capacity, then RMHT frequently waits as it is blocked until task queue capacity becomes available. On the other hand, if the task queue has a large capacity, RMHT remains idle most of the time and some queue capacity goes to waste. Therefore, we want to determine a reasonable size for the task queue to make the ASK system efficient. We can check the probability of RMHT blocking by iteratively increasing the queue capacity by 1 in subsequent runs, but this laborious approach is too time consuming. Alternatively, we can assume that the task queue has infinite capacity and try to find how much of it is actually used. With this task queue, we obtained the long-run expected number of task-loss due to unavailable buffer capacity or the unbalanced performance of RMHT and RMHRT threads. For this purpose, we use the following PRISM property $R\{\text{"num_dataLoss"}\}=? [S]$. The result is shown in Fig. 5.1.2.

To mimic an infinite queue, we use a `LossySync` channel feeding into a queue with a fixed capacity. This construct always accepts arriving tasks, but arriving tasks are lost when the queue is full. We can approximate the minimum required queue capacity out of the expected number of losing tasks

```

Terminal
File Edit View Search Terminal Help
Model checking: R{"num_dataLoss"}=? [ S ]

SCCs: 1, BSCCs: 1, non-BSCC states: 0
BSCC sizes: 1:14528

Computing steady state probabilities for BSCC 1

Building hybrid MTBDD matrix... [levels=14, nodes=57678] [2.6 MB]
Splitting into blocks... [levels=5, n=29, nnz=419, compact] [1.8 KB]
Adding explicit sparse matrices... [levels=9, num=418, compact] [557.0 KB]
Creating vector for diagonals... [dist=445, compact] [31.9 KB]
Allocating iteration vectors... [113.5 KB + 4.0 KB = 117.5 KB]
TOTAL: [3.3 MB]

Starting iterations...

Gauss-Seidel: 78 iterations in 2.44 seconds (average 0.001962, setup 2.29)

BSCC 1 Reward: 0.018521245454519365

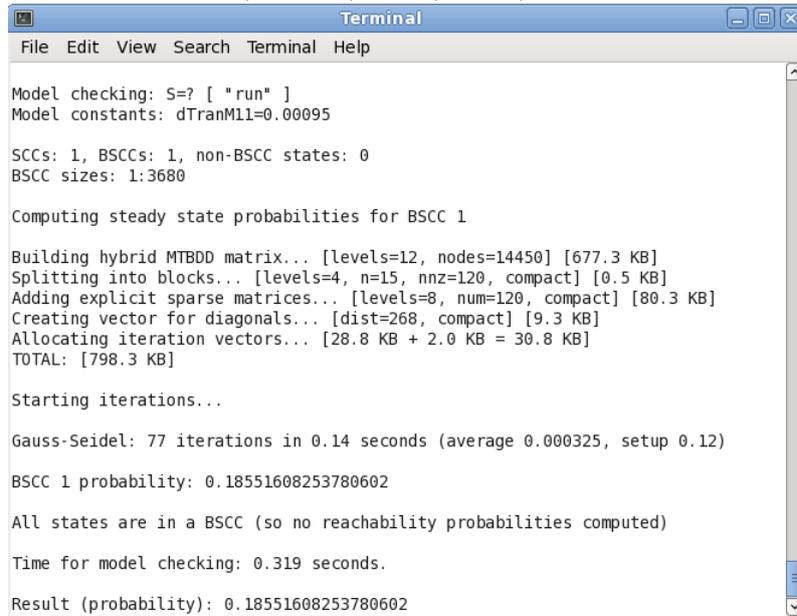
All states are in a BSCC (so no reachability probabilities computed)

Time for model checking: 3.29 seconds.

Result (expected num_dataLoss): 0.018521245454519365

```

Fig. 7. Expected number of task-loss in the task queue



```

Terminal
File Edit View Search Terminal Help

Model checking: S=? [ "run" ]
Model constants: dTranM11=0.00095

SCCs: 1, BSCCs: 1, non-BSCC states: 0
BSCC sizes: 1:3680

Computing steady state probabilities for BSCC 1

Building hybrid MTBDD matrix... [levels=12, nodes=14450] [677.3 KB]
Splitting into blocks... [levels=4, n=15, nnz=120, compact] [0.5 KB]
Adding explicit sparse matrices... [levels=8, num=120, compact] [80.3 KB]
Creating vector for diagonals... [dist=268, compact] [9.3 KB]
Allocating iteration vectors... [28.8 KB + 2.0 KB = 30.8 KB]
TOTAL: [798.3 KB]

Starting iterations...

Gauss-Seidel: 77 iterations in 0.14 seconds (average 0.000325, setup 0.12)

BSCC 1 probability: 0.18551608253780602

All states are in a BSCC (so no reachability probabilities computed)

Time for model checking: 0.319 seconds.

Result (probability): 0.18551608253780602

```

Fig. 8. steady-state probability of thread in use

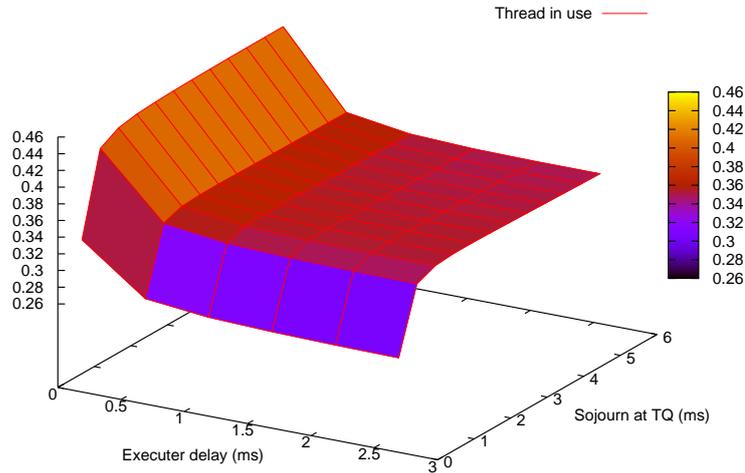
by this construct. Replacing the FIFO1 queue in Fig. 6 by the LossyFIFO1 connector in Fig. 4 provides such a pseudo-infinite task queue for this analysis. According to this result, around 18.5^3 requests are lost per second in front of the task queue. From this result, we can conclude that the minimum capacity of the task queue needs to be 20 to guarantee no task-loss.

5.1.2 Functions

The RMHRT threads are the primary task handling processes. Thus, the performance of the Reception component depends on the collective performance of its RMHRT threads. It is interesting to learn how many RMHRT threads are required to handle a task load, or what is the reasonable performance of RMHRT threads that can provide a satisfactory QoS. Instead of changing the number of RMHRT threads, here we fix their number at 2 and vary their performance by changing their processing delay rates. These two threads have the same architecture with the same performance, thus, the analysis on the utilization is carried out on the RMHRT1 thread, the result of which can be used for the other RMHRT thread. We first find the steady-state probability that the RMHRT1 thread is running, expressed as $S=?["run"]$ in PRISM. The result, shown in Fig. 8, implies that the utilization of the RMHRT1 is 18%.

In a series of analysis experiments on this property, we varied the processing delay rates for the RMHRT1 thread. However, the gaps between the experiment results are not significant. For example, when we considered the activity of the RMHRT1 as an immediate activity by setting its rate as 2,147,483,687, the steady-state probability $S=?["run"]$ from this rate value

³ The result 0.0185 was derived with millisecond as time unit.

Fig. 9. Steady-state probability $S=?["run"]$

was 14%. Compared to the huge differences between these two values, their resulting probabilities are barely changed. This implies that improving the performance of the RMHRT1 thread does not influence the overall performance of the Reception component that much, which suggests the presence of some bottlenecks in this system.

In order to figure out the bottlenecks, we experimented with the model by varying the rates relevant to other activities in the system. Fig. 9 shows the probability results of these experiments. The label *Sojourn at TQ* presents the exit rate from the task queue. As this rate decreases, incoming requests stay longer in the task queue, and the RMHRT threads become more idle, i.e., the probability of the thread utilization decreases, since the request arrive at the thread less frequently. The graph in Fig. 9 shows this tendency when one projects this graph onto the (Prob., Soj.) plane. This implies that increasing *Sojourn at TQ* value generates higher utilization of the thread.

The label *Executer delay* represents the frequency that the Executer component takes the output from the Reception component. As this rate decreases, the threads in the Reception component need to keep their results waiting longer and block incoming tasks. Thus, the thread becomes less idle, i.e., the utilization of the thread increases, but their throughput becomes low since the thread just waits without doing anything. This tendency is also observable in the graph in Fig. 9 when one projects this graph onto the (Prob., Exe.) plane. To obtain meaningful utilization, we must increase *Executer delay*.

Based on the graph in Fig. 9, we now determine bottlenecks in this system. In general, a small change in a bottleneck causes significant differences for the overall performance. The graph in Fig. 9 shows an instance of this: variations in the rates in the interval $[0.1, 0.6]$ for both *Executer delay* and *Sojourn at TQ* induce a big variation on the probability of utilization of the thread (represented in the vertical axis). Thus, these two rates can be assumed to

be bottlenecks, which limit the overall performance. In order to mitigate these bottlenecks, we need to increase both rates at least above 0.6. However, we cannot increase these rates enormously since their relevant resources are neither infinite nor free. As a criterion for this increase, we can consider the convergent disposition of this graph. Above the value 1.3 of the respective rates, the utilization of the thread converges. Thus, we can choose the third values of the respective rates for the best cost-effective utilization of the thread in this system.

5.2 Simulation-based analysis

The Stochastic Reo Simulator [10,19] supports performance evaluation of Reo models through simulation. It allows arbitrary distributions for describing stochastic properties of channels and components. The method used by this tool combines simulation techniques and specific stochastic automata models to conduct automated performance analysis of both steady-state and transient properties of the model. The Stochastic Reo Simulator tool uses the coloring semantics [6] of Reo to properly model context-dependent behavior, i.e. to express the availability of requests. The tool is developed as a plug-in within the Eclipse Coordination Tools (ECT) [9,11]. Through the GUI editor of the ECT, one can develop a model of a system as a Reo circuit in an intuitive way, annotate the circuit with rates, and then use the simulator to get insight into the behavior of the model.

The simulator provides information about (1) the average waiting times of I/O requests at boundary nodes, (2) buffer utilization, (3) end-to-end delays, and (4) channel utilization. Using this simulator on the Reception component, we used the distributions extracted in Section 4.2. Due to space limit, we do not show details of the use of the simulator here. We show a more detailed description on the ECT web-page⁴. As a few examples, the properties/facts we learned about the Reception component from the simulation include: the task queue is used 57% of the time; it takes, on the average, 6.5 milliseconds to handle a request; and the waiting time of I/O requests at the `RRequestIn` node in Fig. 6 is, on the average, 1.7 milliseconds.

6 Discussion

In this paper, we have presented a stochastic analysis of (a deployed installation of) the ASK system. We modeled the system using Stochastic Reo, from which we generated the CTMCs corresponding to some of the modules of the system. This enabled us to use the probabilistic model checker PRISM to verify some properties of interest, using the concrete data extracted from the logs

⁴ <http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/CaseStudies/SimulatoronASK>

of the running ASK installation. The results of this verification allowed us to draw conclusions about resource allocation and how the system installation can be adapted in order to improve its performance. CTMC models have the limitation of supporting only exponential distributions. To overcome this limitation, we also used a simulator. Even though the result from the simulation is approximation-based analysis, we can gain insight into the aspects of the behavior of the system that involve non-exponential distributions.

We have focused our analysis in this paper only on the Reception component of the ASK system. However, the other components have very similar architectures and, thus, all the techniques used in this paper can be easily applied to them as well.

The distributions used in this case study were obtained by statistical analysis based on the real logs of an actual running ASK system. Our analysis revealed exponential distributions for the arrivals and I/O requests. However, rates for the processing/service times of some components were not exponentially distributed. This made it necessary to do simulation for additional analysis. We used the Reo simulator [10,19], an integrated ECT tool, which enables the use of arbitrary distributions and predefined probabilistic behaviors. Using this simulator we can study a model which, for instance, has exponentially distributed data arrivals and log-normal distributed processing rates in some components.

In this analysis, we found two bottlenecks that were caused by (1) the low availability of the Executer component and (2) the long sojourn time at the task queue. In what concerns (1), we observe that we are modeling the connections between the Reception and other components (Executer and Matcher) synchronously (that is, using Sync channels), and that the observation that the consumption rates of the other two components become bottlenecks is not surprising. We have experimented with replacing the Sync channels with FIFOs to decouple the components and remove these bottlenecks. In the process of these experiments, we identified another bottleneck internal to the Executer component itself. In what concerns (2), the bottleneck is caused by congestion between the task queue and the threads. Thus, we can widen the bandwidth of this connection to obtain better performance for the system.

In earlier initiatives to improve the performance of the ASK system, the focus has been primarily on improving the execution times of request handling tasks, through extensive profiling. The work presented in this paper confirms and explains the observations from small experiments with ASK components in isolation, carried out by Almende last year. As a consequence of this, Almende decided to put additional effort into the optimization of queue sizes and bandwidth between the task queue and the threads in each of the ASK components. First attempts in this direction yield promising results.

Acknowledgments.

The authors are thankful to Christian Krause and Oscar Kanters for their help in using the Reo simulator.

References

- [1] Almende website. <http://www.almende.com>.
- [2] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
- [3] F. Arbab. Abstract Behavior Types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1-3):3–52, 2005.
- [4] F. Arbab, T. Chothia, R. van der Mei, S. Meng, Y.-J. Moon, and C. Verhoef. From Coordination to Stochastic Models of QoS. In *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 268–287. Springer, 2009.
- [5] ASK community systems website. <http://www.ask-cs.com>.
- [6] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, 2007.
- [7] Credo project. <http://projects.cwi.nl/credo/>.
- [8] F. S. de Boer, I. Grabe, M. M. Jaghoori, A. Stam, and W. Yi. Modeling and Analysis of Thread-Pools in an Industrial Communication Platform. In *Proc. ICFEM'09*, volume 5885 of *Lecture Notes in Computer Science*, pages 367–386. Springer, 2009.
- [9] Eclipse Coordination Tools. <http://reo.project.cwi.nl/>.
- [10] O. Kanters. QoS analysis by simulation in Reo. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, 2010.
- [11] C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Universiteit Leiden, 2011.
- [12] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Computer Performance Evaluation/TOOLS*, pages 200–204, 2002.
- [13] Y.-J. Moon, A. Silva, C. Krause, and F. Arbab. A Compositional Semantics for Stochastic Reo Connectors. In *FOCLASA*, volume 30 of *EPTCS*, pages 93–107, 2010.
- [14] C. Z. Mooney and R. D. Duval. *Bootstrapping: a nonparametric approach to statistical inference*. Sage Publications, 1993.
- [15] PRISM website. <http://www.prismmodelchecker.org/>.
- [16] A. Stam. The ASK System and the Challenge of Distributed Knowledge Discovery. In *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 663–668. Springer, 2008.
- [17] A. Stam, S. Klüppelholz, T. Blechmann, and J. Klein. ReASK Final Models. Technical Report To appear, Almende, The Netherlands and Technical University of Dresden, Germany, 2009.
- [18] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.
- [19] C. Verhoef, C. Krause, O. Kanters, and R. van der Mei. Simulation-based Performance Analysis of Channel-based Coordination Models. In *COORDINATION 2011*, volume 6721 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 2011.