Chapter 6

Case study

6.1 Introduction

In this chapter we show how the theory developed in previous chapters, 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 chapter are the following:

1. a stochastic Reo model of the ASK system[1]. The distributions in this model were obtained by statistical analysis of real values filtered out of the logs of an actual running ASK system.

2. analysis of several interesting properties using the probabilistic model checker PRISM [57, 76] 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.

[1]Details available at `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/CaseStudies/SimulatoronASK/Reception`.

3. analysis of the system using a simulator which enables the study of properties involving non-exponential distributions (CTMCs can deal only with exponential distributions).

6.2 The ASK system

The "Access Society's Knowledge" (ASK) system [83] is an industrial software developed by the Dutch company Almende [1], and marketed by their daughter company ASK Community Systems [10]. 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 the following aspects:

- 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 requests. 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.

Some representative applications of the ASK system include:

- *Workforce deployment.* To offer deployment of temporary workers by finding those who are available for an assignment. For instance, this can the setup in an employment agency.
- *Customer services.* To directly connect customers to proper, available service providers, according to the customers' requirements. For instance, the Dutch housing corporation Vestia increases its tenant satisfaction by using the ASK system to put its tenants in direct contact with a repairman in case some house facilities need to be repaired.
- *Emergency response.* To collect the status information of emergent situations and provide safety by utilizing all possible means of communications. For instance, this can be used to find available volunteers with the best accessibility for emergency situations.
- *Flexible resource allocation.* To increase the flexibility of workforce and to decrease scheduling workload. For instance, this can be used to provide the best matches between working schedules and private lives of employees. In fact, the European mail distribution company TNT Post uses the ASK system for this type of flexible resource allocation.
- *Knowledge sharing.* To collect, share, and distribute information, experiences, skills, and preferences of users in order to provide high quality of service. For instance, in patient care applications that involve multiple care giver professionals or institutions, such as individuals, hospitals, and/or pharmaceutical companies, the ASK system can be used to update and share patient information to provide proper services in synchronization.

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 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 models describe random system behavior, leading to more realistic models of behavior than their deterministic counterparts. CTMCs, one of stochastic models, are frequently used to model randomized behavior in various systems and their features, and efficient closed-form and numerical techniques [85] exist for analysis. Traditionally, such models are constructed by human experts whose experience and insight constitute the only link between an 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 form 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 [34] within the context of the EU project Credo [33] for verification of its functional properties. In the work we report in this chapter, 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 the Almende company, we use this model to analyze and study the QoS properties of the ASK system in various settings. For instance, using the approach in [68], 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 6.4.1. The following sections describe the architecture of the ASK system in some detail. The figures and the descriptions we use here are based on [84].

6.2.1 Overview of the ASK system

The top-level architecture of the ASK System is shown in Figure 6.1. 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* (Domain Data in Figure 6.1), 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.

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 Figure 6.1.

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. According to the determined steps, the result of the Reception component is sent to either the *Matcher* or the *Executer* component. The *Matcher* component determines proper participants for fulfilling a request. The *Executer* component determines the best means of connection

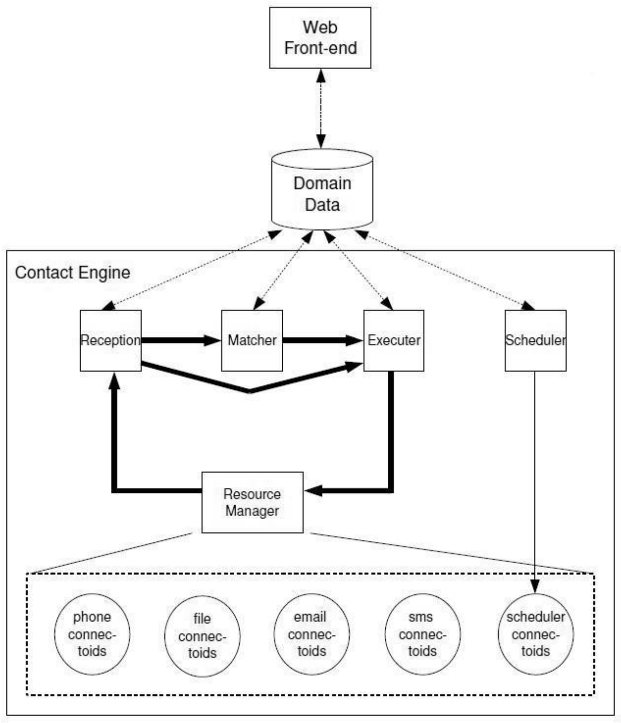


Figure 6.1: Overview of ASK system

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. For example, an incoming call arrives from the outside. First, the Resource Manager component handles this call. If the request of the call is simple and trivial, then the Resource Manager component establishes the connection between users immediately. Otherwise, the request is sent to the Reception component. The Reception component gathers some information from users and stores the information in the database, and also determines if it needs to decide either the proper service participants or the efficient way of the connection between users. For determining the proper service providers, the request is sent to the Matcher component; for the efficient ways of connections, the request is sent to the Executer component. The Resource Manager provides the connection between the determined service participants using the determined means of connection. Actual connections occur based on the schedule by the Scheduler component.

6.3 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.

6.3.1 The Reception component

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

- **HostessTask (HT)** which converts incoming requests into tasks that will be put into the task queue.
- **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 the HT function and the other for the HRT function. In general, the HRT function

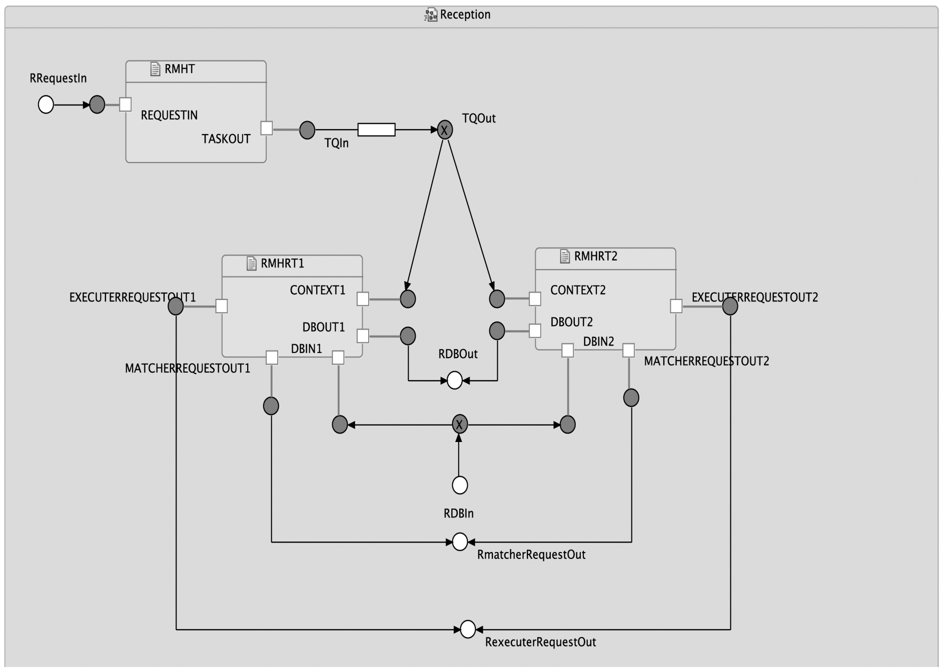
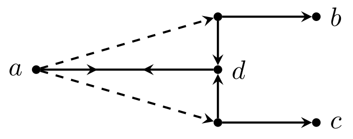


Figure 6.2: Reception component in ECT

takes more time than the HT function, since it actually deals with incoming tasks. Thus, the Reception component needs more threads running the HRT function. For simplicity of modeling, we assume that every thread in the Reception component has only one function, e.g., either the HT or the HRT function. Reflecting this simplification, Figure 6.2 shows the Reception model drawn in the Extensible Coordination Tools (ECT) [35]. This figure shows a Reception component with three RM threads, one with only the HT function and the other two with only the HRT function.

The inside boxes of the RMHT and the indexed RMHRTs in Figure 6.2 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 the 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 Figure 6.2 (a sink node of the FIFO1 channel), 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, which will be explained in Section 6.3.2. The graphical notation $\otimes$ used for *TQOut* in Figure 6.2 is an abbreviation for an *exclusive router* [3] whose Reo circuit is depicted below.



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.

Figure 6.2 serves as a basic template model for the Reception component. Depending on the specific properties of interest in each analysis, we slightly adapt this basic template. For example, for the analysis of the properties of the task queue, we may substitute a LossyFIFO1 connector for the FIFO1 channel, as shown in Section 6.4.1. It should be noted that more than 3 RM threads can be used for modeling the Reception component, but here we use only 3 threads since the CTMC corresponding to the Stochastic Reo model of the Reception component with 3 RM threads is already big to handle.

6.3.2 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 (Rexecuter-RequestOut). 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.

6.4 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 6.4.2). The simulator was also used when we could not obtain any proper distribution from the logs at all. In this case, we used bootstrapping [69] in the simulator with the original data as special inputs in the simulator for the rates.

6.4.1 Analysis on 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.

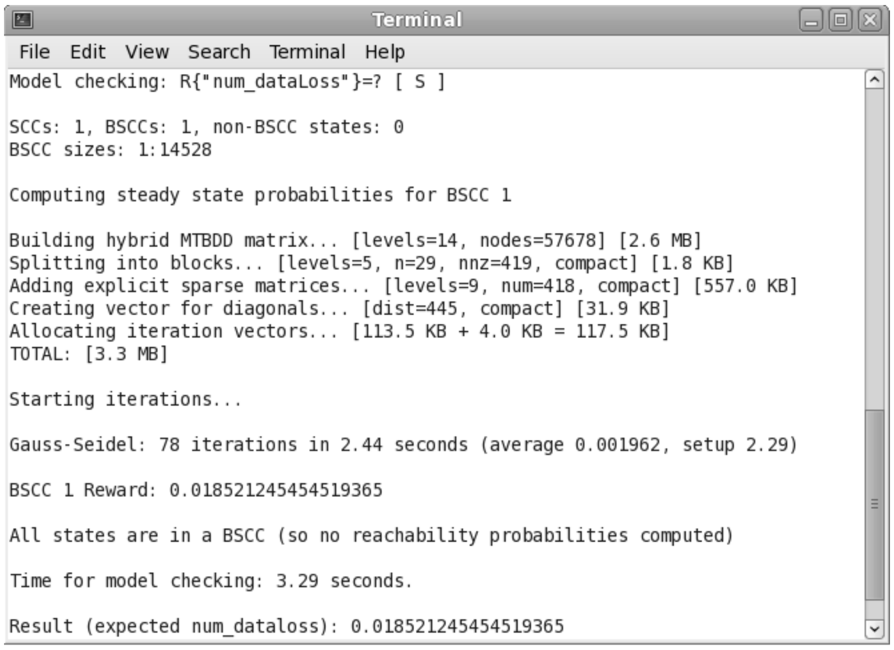
- `num_dataLoss` represents the number of task-loss in the task queue.
- `run` represents the running status of the RMHRT thread.

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 Figure 6.2, the resources of interest include:

1. the minimum capacity of the task queue
2. the utilization and/or the performance of the RMHRT threads that handle tasks

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 (the least sufficient) size for the task queue to make the ASK system efficient. We can check the probability of RMHT

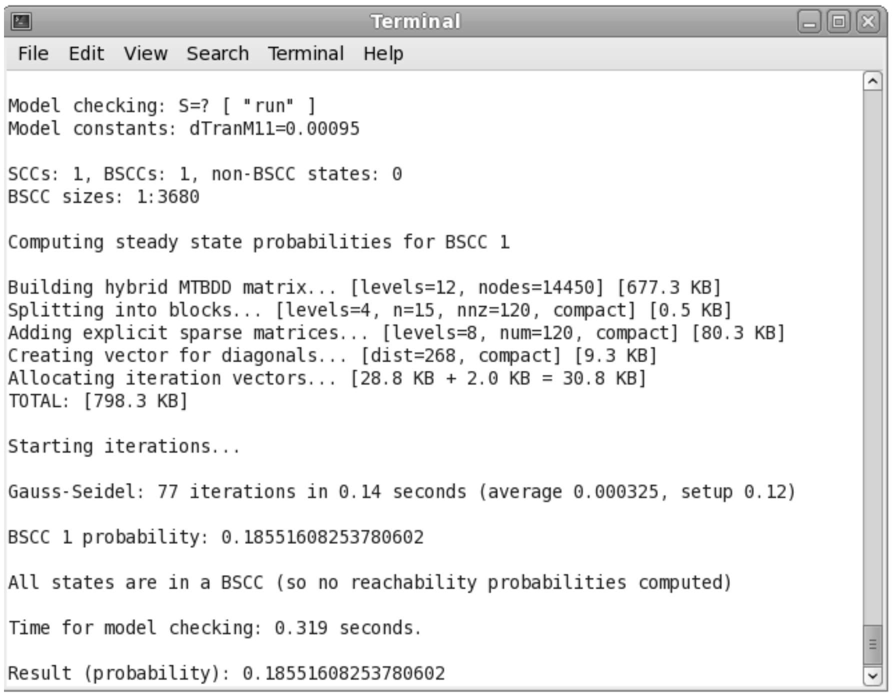


```
Model checking: R{"num_dataLoss"}=? [ S ]

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
```

Figure 6.3: Long-run expected number of task-loss

blocking by iteratively increasing the queue capacity 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{"num_dataLoss"}=?[S]`. The result is shown in the screen-shot in Figure 6.3.

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 by this construct. Replacing the FIFO1 queue in Figure 6.2 by the LossyFIFO1 connector in Figure 4.1 provides such a pseudo-infinite task queue for this analysis. According to this result, around 18.5[2] 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.



```
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
```

Figure 6.4: Probability that RMHRT1 is running

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 Figure 6.4, 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 infinity, the steady-state probability

[2] The result 0.0185 was derived with millisecond as time unit.

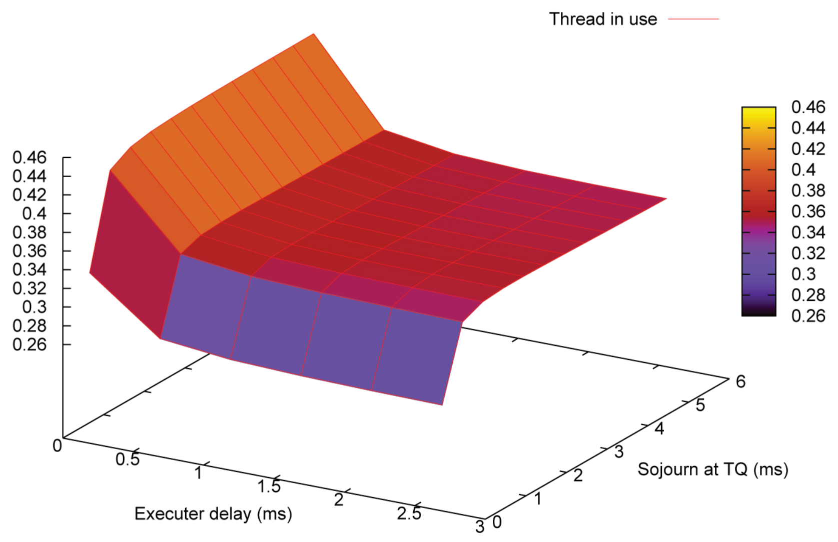


Figure 6.5: Steady-state probability `S=?["run"]`

`S=?[“run”]` from this rate value was 14%. Compared to the huge differences between these two values for the delay rate of the RMHRT1 thread[3], 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. Figure 6.5 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 Figure 6.5 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 Figure 6.5 when one

[3]The original rate value derived from the statistical analysis is 0.095, and we used the value $2^{31}-1$ as an infinity for this comparison.

projects this graph onto the (`Prob.`, `Exe.`) plane. To obtain meaningful utilization, we must increase `Executer delay`.

Based on the graph in Figure 6.5, 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 Figure 6.5 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 1.3 as the values of the respective rates for the best cost-effective utilization of the thread in this system.

6.4.2 Simulation

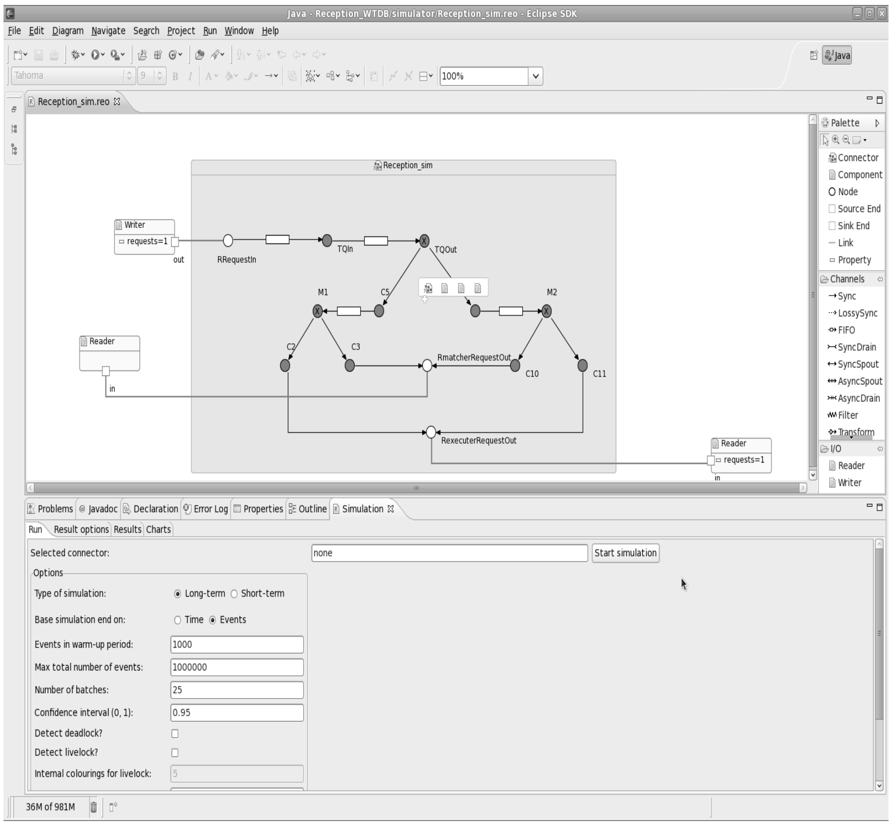
The Stochastic Reo simulator [51, 89] supports performance evaluation of Reo models through simulation. It allows arbitrary distributions for describing stochastic properties of channels and components. The method used by this tool combines simulation techniques and specific stochastic automata models to conduct automated performance analysis of both steady-state and transient properties of the model. The tool uses the coloring semantics [30] of Reo to properly model context-dependent behavior, i.e. to express the availability of requests. The Stochastic Reo simulator tool is developed as a plug-in within the ECT by Oscar Kanters. 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.

For the simulation of the ASK system, we also focus on the Reception component. Since the other components in the ASK system have very similar architectures, we can use some of the results from this simulation for their simulation as well. The Stochastic Reo simulator tool does not support hierarchical models yet. Therefore, to run our simulation we had to flatten the original Reo model for the ASK system (shown in Figure 6.2), abstract its nested components into FIFO1 channels, and somewhat simplify it to reflect some restrictions. The resulting Reo circuit that we use for this simulation analysis appears in Figure 6.6 as an ECT screen-shot.

As a plug-in within the ECT, the simulator is used and accessed using the `Simulation` tab under the Reo editor (See Figure 6.6). In the `Run` and `Result options` sub-tabs under the `Simulation` tab, users can set some variables such as `Type of simulation`[4], `Max total number of events`[5], and so on. Having clicked the Reo model to be analyzed and pressed the `Start simulation` button in the `Simulation`

[4]Long- or short-term simulation.

[5]When the simulation is based on events, the length of the simulation is determined by the given event numbers.



Selected connector: none

Options

Type of simulation: [✓ Long-term] Short-term

Base simulation end on: Time [✓ Events]

Events in warm-up period: 1000

Max total number of events: 1000000

Number of batches: 25

Confidence interval (0, 1): 0.95

Detect deadlock? ☐

Detect livelock? ☐

Internal colourings for livelock: 5

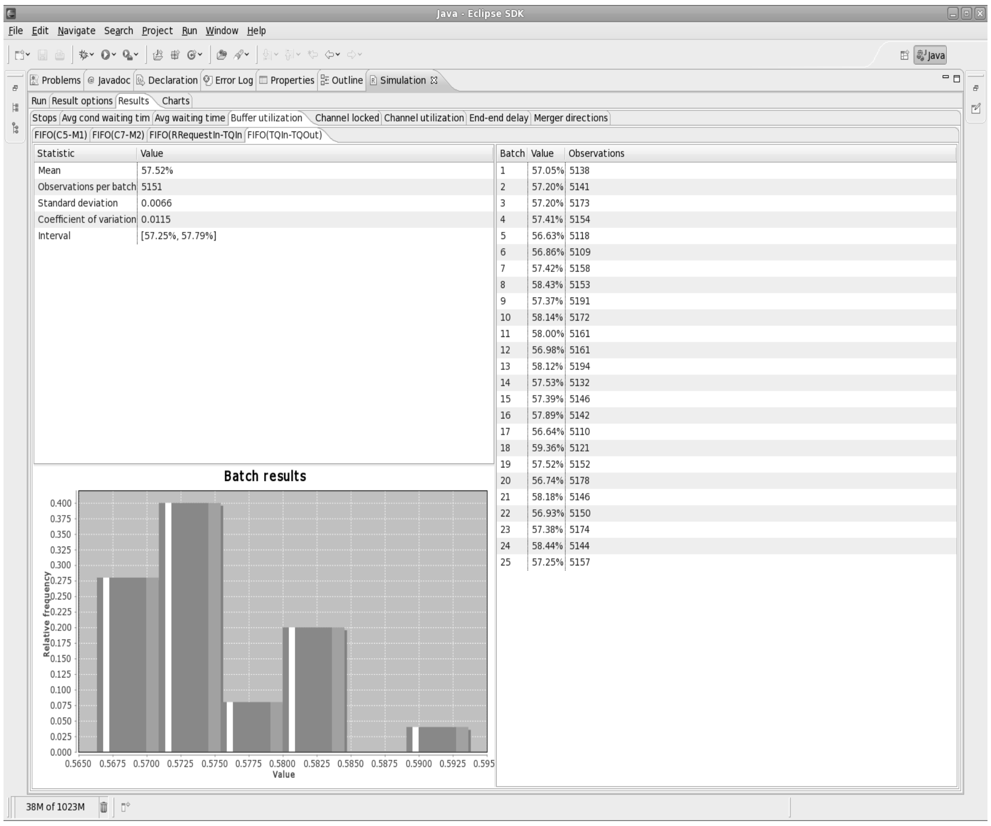
Figure 6.6: Flattened and simplified Reception model

tab, the simulation on the model is carried out. The simulation results are presented in other sub-tabs of `Results` and `Charts`. More detailed explanation on the usage and the underlying methods of this Reo simulator is given by [51].

The simulator provides information about

1. buffer utilization
2. end-to-end delays
3. the average waiting times of I/O requests at boundary nodes
4. channel utilization

The following examples show this information applying the simulator on the ASK system. Because we are not intended in channel utilization in this section, we assumed


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Java
Problems Javadoc Declaration Error Log Properties Outline Simulation
Run Result options Results Charts
Stops Avg cond waiting tim Avg waiting time Buffer utilization Channel locked Channel utilization End-end delay Merger directions
FIFO(C5-M1) FIFO(C7-M2) FIFO(RRequestIn-TQIn FIFO(TQIn-TQOut)
Batch results
Relative frequency
Value
38M of 1023M

Statistic	Value
Mean	57.52%
Observations per batch	5151
Standard deviation	0.0066
Coefficient of variation	0.0115
Interval	[57.25%, 57.79%]

Batch	Value	Observations
1	57.05%	5138
2	57.20%	5141
3	57.20%	5173
4	57.41%	5154
5	56.63%	5118
6	56.86%	5109
7	57.42%	5158
8	58.43%	5153
9	57.37%	5191
10	58.14%	5172
11	58.00%	5161
12	56.98%	5161
13	58.12%	5194
14	57.53%	5132
15	57.39%	5146
16	57.89%	5142
17	56.64%	5110
18	59.36%	5121
19	57.52%	5152
20	56.74%	5178
21	58.18%	5146
22	56.93%	5150
23	57.38%	5174
24	58.44%	5144
25	57.25%	5157

Figure 6.7: Buffer utilization of the task queue

most data-flows through channels are immediate actions. The distributions used for this simulation are also derived from the real logs of an actual running ASK system.

As an example of the simulation results we obtain, Figure 6.7 presents the ECT screen-shot that shows the utilization of the buffer between the TQIn and TQOut nodes. Actually, this buffer corresponds to the task queue in the Reception component, whose average utilization, according to this analysis, is 57.52%. That is, on the average, this buffer is used during 57% of the running time of the ASK system.

In addition, the graph at the bottom of the result in Figure 6.7 shows how frequently this average value occurs during simulation. In general, this value follows a normal distribution, which in this case does not happen. This can be due to a low number of batches: if this number increases, our graph may tend to a normal distribution.

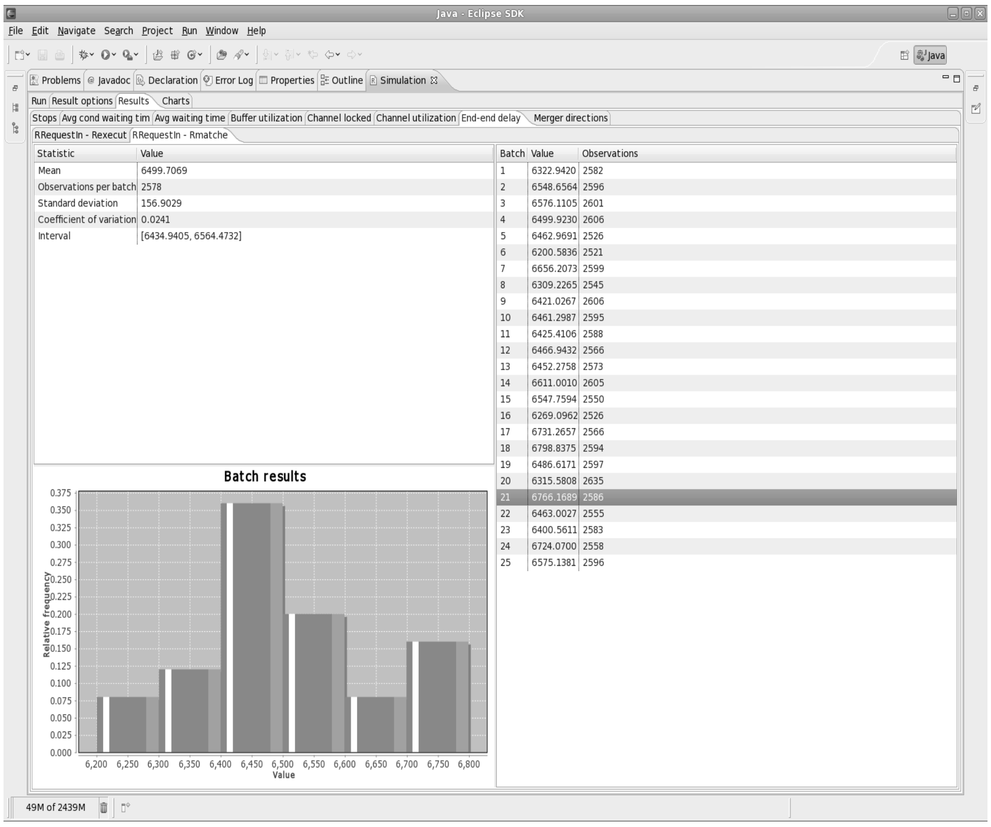
As another example of the simulation, Figure 6.8 presents the ECT screen-shot that shows one of the end-to-end delays from the *RRequestIn* node to the *RmatcherRe-*

questOut node in Figure 6.6. This delay implies how long it takes for the Reception component to handle an incoming request and to send its result to the Matcher component. The average end-to-end delay is around 6500 microseconds, i.e., 6.5 milliseconds.

As the last example, Figure 6.9 presents the screen-shot that shows the waiting time of I/O request at *RRequestIn* node in Figure 6.6. According to this result, on the average, I/O requests wait 1727 microseconds, i.e., around 1.7 milliseconds.

6.5 Discussion

In this chapter, 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



Statistic	Value
Mean	6499.7069
Observations per batch	2578
Standard deviation	156.9029
Coefficient of variation	0.0241
Interval	[6434.9405, 6564.4732]

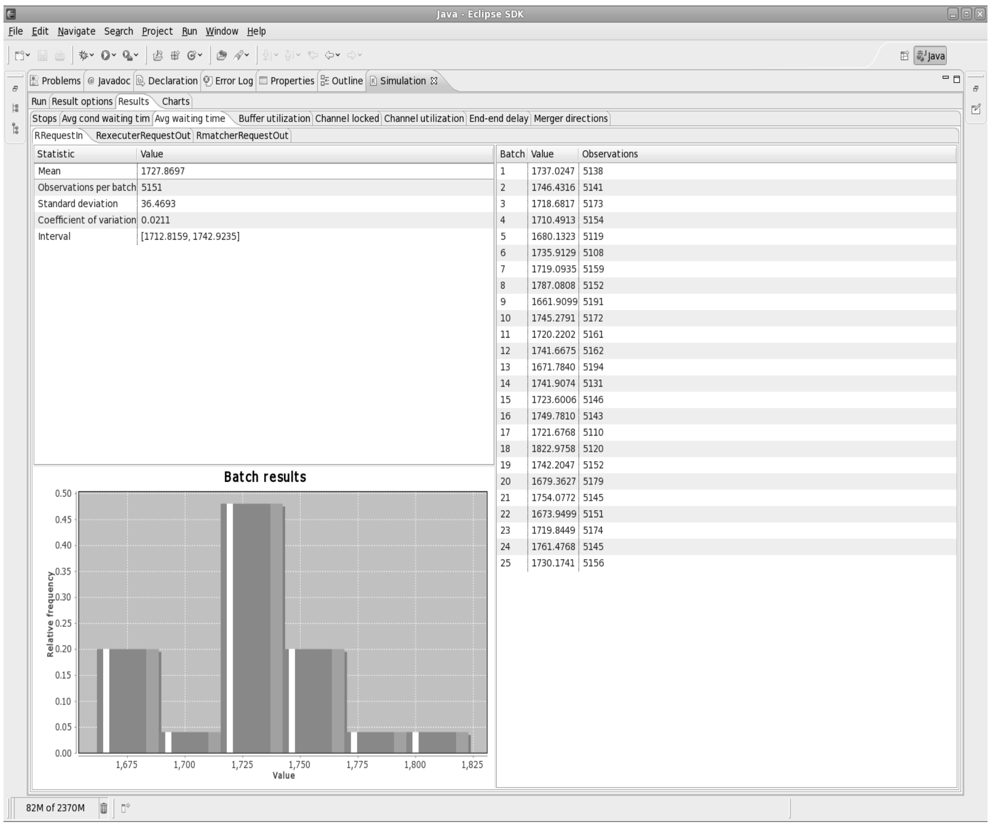
Batch	Value	Observations
1	6322.9420	2582
2	6548.6564	2596
3	6576.1105	2601
4	6499.9230	2606
5	6462.9691	2526
6	6200.5836	2521
7	6656.2073	2599
8	6309.2265	2545
9	6421.0267	2606
10	6461.2987	2595
11	6425.4106	2588
12	6466.9432	2566
13	6452.2758	2573
14	6611.0010	2605
15	6547.7594	2550
16	6269.0962	2526
17	6731.2657	2566
18	6798.8375	2594
19	6486.6171	2597
20	6315.5808	2635
21	6766.1689	2586
22	6463.0027	2555
23	6400.5611	2583
24	6724.0700	2558
25	6575.1381	2596

Figure 6.8: End-to-End delay from *RRequestIn* to *RmatcherRequestOut* in Figure 6.6

us to use the probabilistic model checker PRISM to verify some properties of interest, using the concrete stochastic distributions extracted from the logs 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 chapter 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 chapter 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 ex-


Java - Eclipse SDK
File Edit Navigate Search Project Run Window Help
Problems Javadoc Declaration Error Log Properties Outline Simulation
Run Result options Results Charts
Stops Avg cond waiting tim Avg waiting time Buffer utilization Channel locked Channel utilization End-end delay Merger directions
RRequestIn RexecuterRequestOut RmatcherRequestOut
Batch results
Relative frequency
Value
82M of 2370M

Statistic	Value
Mean	1727.8697
Observations per batch	5151
Standard deviation	36.4693
Coefficient of variation	0.0211
Interval	[1712.8159, 1742.9235]

Batch	Value	Observations
1	1737.0247	5138
2	1746.4316	5141
3	1718.6817	5173
4	1710.4913	5154
5	1680.1323	5119
6	1735.9129	5108
7	1719.0935	5159
8	1787.0808	5152
9	1661.9099	5191
10	1745.2791	5172
11	1720.2202	5161
12	1741.6675	5162
13	1671.7840	5194
14	1741.9074	5131
15	1723.6006	5146
16	1749.7810	5143
17	1721.6768	5110
18	1822.9758	5120
19	1742.2047	5152
20	1679.3627	5179
21	1754.0772	5145
22	1673.9499	5151
23	1719.8449	5174
24	1761.4768	5145
25	1730.1741	5156

Figure 6.9: Waiting time of I/O request at *RRequestIn*

ponential 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 [51, 89], 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 chapter 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.