Analyzing Web Server Performance Models with the Probabilistic Model Checker PRISM*

Tamás Bérczes[†]
tberczes@inf.unideb.hu

Gábor Guta[‡]
Gabor.Guta@risc.uni-linz.ac.at

Gábor Kusper[§]
gkusper@aries.ektf.hu

Wolfgang Schreiner[‡]
Wolfgang.Schreiner@risc.uni-linz.ac.at

János Sztrik[†]
jsztrik@inf.unideb.hu

November 14, 2008

[†]Faculty of Informatics, University of Debrecen, Hungary, http://www.inf.unideb.hu

[‡]Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at

[§]Esterházy Károly College, Eger, Hungary, http://www.ektf.hu

*Supported by the Austrian-Hungarian Scientific/Technical Cooperation Contract HU 13/2007.

Abstract

We report our experience with formulating and analyzing in the probabilistic model checker PRISM various closely related web server performance models that were previously described in literature in terms of classical queuing theory. By our work various ambiguities and deficiencies (also errors) are revealed; in particular, the PRISM models which combine state machines descriptions with performance characteristics show that the original descriptions used slightly differed assumptions for their analysis.

Furthermore, while the queuing models are typically based on infinite queues, the state spaces of the PRISM models have to be finite for an automated analysis. While this forces us to explicitly deal with buffer overflows, it also gives us the possibility to investigate appropriate buffer sizes for concrete implementations of the models. Although also one of the previously reported models used a finite queue in some place, our investigations reveal that the size of that queue is actually not critical, while another (previously not constrained) one is.

Based on these observations, we argue that nowadays performance modeling should make use of (at least be accompanied by) state machine descriptions such as those used by PRISM. On the one hand, this helps to more accurately describe the systems whose performance are to be modeled (by making hidden assumptions explicit) and give more useful information for the concrete implementation of these models (appropriate buffer sizes). On the other hand, since probabilistic model checkers such as PRISM are furthermore able to analyze such models automatically, analytical models can be validated by corresponding experiments which helps to increase the trust into the adequacy of these models and their real-world interpretation.

Contents

1 Introduction

The two originally distinct areas of the qualitative analysis (verification) and quantitative analysis (performance modeling) of computing systems have in the last decade started to converge by the arise of stochastic/probabilistic model checking [6]. In [1], we have shown how the probabilistic model checker PRISM [7, 5] compares favorably with a classical performance modeling environment for modeling and analyzing retrial queueing systems, especially with respect to the expressiveness of the models and the queries that can be performed on them. In the present paper, we are making one step forward by applying PRISM to re-assess various web server performance models that have been previously described and analyzed in literature.

The starting point of our work is the seminal paper [8] which for the first time presented a performance model for a system of a web server and a web client and analyzed the model with respect to various parameters. This has stimulated further research: e.g. in [3], the model is generalized to an environment where a "proxy cache server" receives all the requests from the clients of a local network; with a certain probability the data requested by a client are already cached on the proxy server and can be returned without contacting the web server from which the data originate. In [2], two of the authors of the present paper have further generalized this model by allowing the proxy cache server to receive also requests from external sources. All these models were first informally sketched in the corresponding papers (referring to their respective predecessors) and then manually analyzed on the basis of classical queueing theory [4].

In this paper, we have from each of the informal model sketches constructed a formal model in the language of PRISM [7]. This language essentially allows to construct in a modular manner a finite state transition system (thus modeling the qualitative aspects of the system) and to associate rates to the individual state transitions (thus modeling the quantitative aspects); the mathematical core of such a system is a Continuous Time Markov Chain (CTMC) which can be analyzed by the PRISM tool with respect to queries that are expressed in the language of Continuous Stochastic Logic (CSL) [6].

On the one hand, this work shows again that PRISM is an effective tool for modeling and analyzing systems of the type investigated by the performance evaluation community. Moreover, while constructing the PRISM models requires some effort, the results are much more specific than the informal model sketches given in the reported papers. Most important, by this effort we have revealed various ambiguities, deficiencies, and even plain errors in the previously published models (and the corresponding analysis results). The use of a tool like PRISM therefore

is able to increase the confidence in a performance model much beyond what has been previously possible by classical means.

The remainder of this paper is structured according to the papers mentioned above: in Section 2, we investigate the model described in [8]; in Section 3, we address the model of [3]; in Section 4, we handle the model published in [2]. Section 5 summarizes our findings and conclusions. Appendices A–C list all the PRISM models and the corresponding CSL queries used in this paper.

2 Performance Model of a Web Server

In this section, we investigate the web server performance model reported in [8] on which (directly or indirectly) the models in [3] and [2] are based. Referring to an illustration redrawn in Figure 1, the core description in that paper reads as follows (we have renamed the constants for consistency with the constant names used in the other papers):

> This network consists of four nodes (i.e. single-server queues): two model the Web server itself, and two model the Internet communication network. File requests (i.e. "jobs") arrive at the web server with frequency λ. All one-time "initialization" processing is performed at node S_I. The job then proceeds to node S_R where a single buffer's worth of data is read from the file, processed, and passed on to the network. At node S_S this block of data is transmitted to the Internet at the server's transfer rate (e.g. 1.5 Mbits on a T1 line). This data travels via the Internet and is received by the client's browser, represented by node S_C. If the file has not been fully transmitted, the "job" branches and returns back to node S_R for further processing. Otherwise, the job is complete, and exits the network.
>
> Notice that the branch is a probabilistic one; given an average file size of F and buffer size B_s, the probability that the file has been fully transmitted is $q = B_s/F$. Also, the arrival rate at node S_R (λ') is the sum of the network's arrival rate (λ), and the rate of the jobs flowing from S_C back to S_R. λ' is derived using an operational law of queuing theory: the rate of jobs leaving any stable node must equal its arrival rate.

The complete list of parameters defining the performance of the model (with default values stated in parameters) is given as

- Network Arrival Rate (λ)
- Average File Size ($F = 5275$)
- Buffer Size ($B_s = 2000$)
- Initialization Time ($I_s = 0.004$) (*)
- Static Server Time ($Y_s = 0.000016$) (*)
- Dynamic Server Rate ($R_s = 1310720$) (*)
- Server Network Bandwidth ($N_s = 193000$)
- Client Network Bandwidth ($N_c = 88375$)

Unfortunately, in [8] no values for the parameters marked as (*) are given for the calculations reported in that paper; we substitute in our experiments the corresponding values that were used in [3] and [2] for their models.

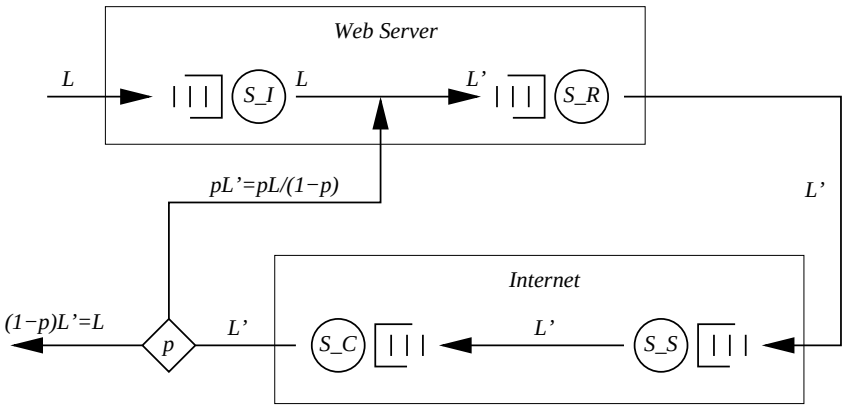

Figure 1: Queueing Network Model of a Web Server (redrawn from [8])

2.1 A First PRISM Model

The verbal descriptions above gives rise to the following (core of) the PRISM code which introduces by the keyword `stochastic` a continuous time Markov chain (CTMC) model [6] (the full code is given in Appendix A.1):

```
stochastic
...

module jobs
  [accept] true -> lambda : true ;
endmodule

module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;

  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

module S_R
  irwaiting: [0..IR] init 0;
  iraccepted: bool init false;
```

```
  [forward] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [forward] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  [repeat] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [repeat] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  [isforward] irwaiting > 0 -> 1/(Ys+Bs/Rs):
    (irwaiting' = irwaiting-1) ;
endmodule

module S_S
  iswaiting: [0..IS] init 0;
  isaccepted: bool init false;

  [isforward] iswaiting = IS -> 1 :
    (isaccepted' = false) ;
  [isforward] iswaiting < IS -> 1 :
    (isaccepted' = true) &
    (iswaiting' = iswaiting+1) ;

  [icforward] iswaiting > 0 -> (Ns/Bs) :
    (iswaiting' = iswaiting-1) ;
endmodule

module S_C
  icwaiting: [0..IC] init 0;
  icaccepted: bool init false;

  [icforward] icwaiting = IC -> 1 :
    (icaccepted' = false) ;
  [icforward] icwaiting < IC -> 1 :
    (icaccepted' = true) &
    (icwaiting' = icwaiting+1) ;

  [repeat] (icwaiting > 0) & (1-q > 0) -> (Nc/Bs)*(1-q) :
    (icwaiting' = icwaiting-1) ;

  [done] (icwaiting > 0) & (q > 0) -> (Nc/Bs)*q :
    (icwaiting' = icwaiting-1) ;
endmodule
```

The model consists of one process ("module") *jobs* generating requests and four

processes S_I, S_R, S_S, S_C as described above. Each process contains declarations of its state variables (bounded integers or booleans) and state transitions of form

```
[label] guard -> rate : update ;
```

A transition is enabled to execute if its *guard* condition evaluates to true; it executes with a certain (exponentially distributed) *rate* and performs an *update* on its state variables. Transitions in different processes with the same *label* execute synchronously as a single combined transition whose rate is the product of the rates of the individual transitions. Since a product of rates rarely makes sense in a model, it is a common technique to give all but one of the individual transitions the rate 1 and let the remaining transition alone determine the combined rate (we follow this practice in all our PRISM models).

Each node has a counter (*waiting*, *irwaiting*, *iswaiting*, *icwaiting*) that denotes the number of jobs waiting in the corresponding queue. If a new job arrives and the queue is not full (the counter has not yet reached its upper bound), this counter is increased and an "acceptance" flag (*accepted*, *iraccepted*, *isaccepted*, *icaccepted*) is set to "true"; otherwise, the job is dropped and the flag is set to "false".

As indicated by the verbal description, jobs may "branch back" with a probability q. As indicated by the illustration, the branching occurs after a server block has been transferred to the client; i.e. the client requests from the server the transfer of the next block. This "branching back" is modeled above by a transition *repeat* in S_C that forwards with probability $1-q$ a request back to S_R while the transition *done* completes the request with probability q. Since CTMC models use rates rather than probabilities, we scale the basic rate (N_c/B_s) of the transitions (which models the forwarding of a block of size B_s over the client network with bandwidth N_c) with factors q respectively $1-q$. Since both transitions are simultaneously enabled, they "compete" for execution, and are consequently executed in proportion to their respective rates, i.e. in proportion $q/(1-q)$, as intended.

We can define in PRISM the "reward structure"

```
rewards "allaccepted"
  accepted   : 1;
  iraccepted : 1;
  isaccepted : 1;
  icaccepted : 1;
endrewards
```

which assigns to every state of a system run the number of acceptance flags that are set to true. Using the CSL query

```
R{"allaccepted"}=? [ S ]
```

we can compute the long term average of this value and use it as an "indicator" (not a reliable approximation) for the probability P that a request is "rejected" (i.e. dropped from the system because it encounters some full buffer).

Likewise, we can define the reward structure

```
rewards "pending"
  true : waiting + irwaiting + iswaiting + icwaiting;
endrewards
```

which assigns to every state the number N of requests "pending" in the system and can compute by a CSL query its long term average.

Having determined N and P, we can apply "Little's Law" from queueing theory [4] to determine the average response time T for a request

$$T = \frac{N}{(1-P)\lambda}$$

i.e. $T \simeq \frac{N}{\lambda}$ for $P \simeq 0$.

In the following, we present the results of the corresponding experiments performed with PRISM (choosing the Jacobi method for the solution of the equation systems and a relative termination epsilon of 10^{-3}).

Using the default values for the model parameters described above, we detect that the capacities of the queues of S_I, S_R, S_S have only little influence on the rejection rate; thus we choose the smallest queue capacities ($IA = IR = IS = 3$) for which the acceptance ratios of the corresponding queues is very close to 1.

However, the situation is different with the capacity IC of the queue in S_C modeling the client network. As depicted in Figure 2, even if we increase IC very much, the acceptance ratio of this queue drops significantly for arrival rates $\lambda \geq 20$. The number of pending requested in Figure 3 is therefore only for $\lambda < 20$ (where $P \simeq 0$) linearly related to the average response time of a request; for $\lambda \geq 20$, it has to be adjusted by the factor $\frac{1}{1-P}$ to give the estimated response time. Nevertheless, we give in Figure 4 the ratio N/λ without adjustments; the curve for $IC = 35$ represents in the range $\lambda \leq 20$ time rather accurately.

We see that response times increase exponentially with the request arrival rate in accordance with the predictions of [8]; beyond a certain bound, the server becomes "saturated" and cannot serve requests in an acceptable time any more.

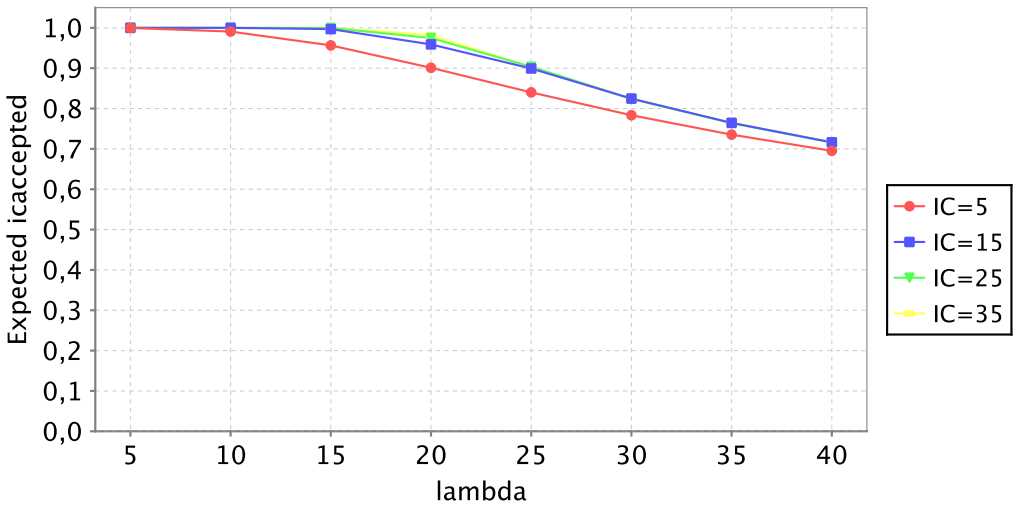

Figure 2: Estimated Acceptance Ratio for S_C

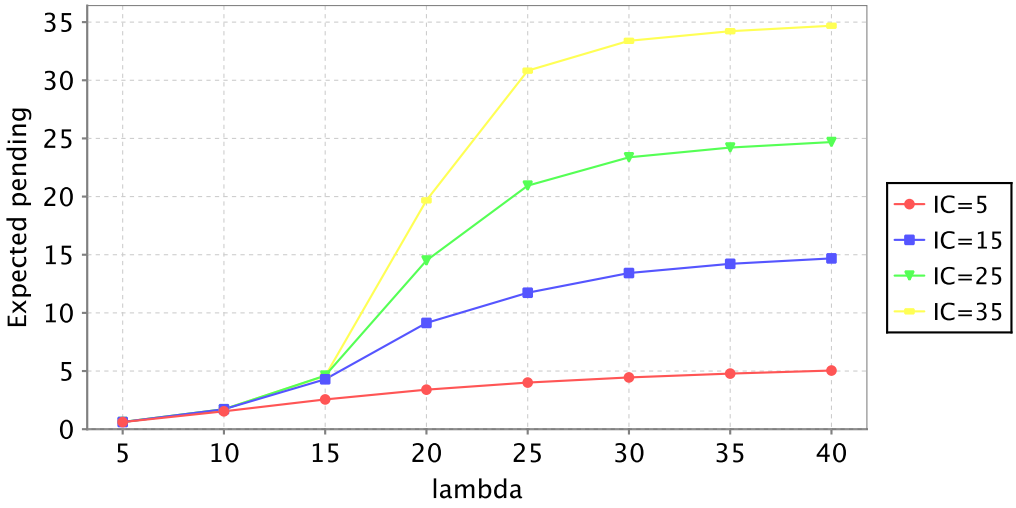

Figure 3: Average Number of Pending Requests

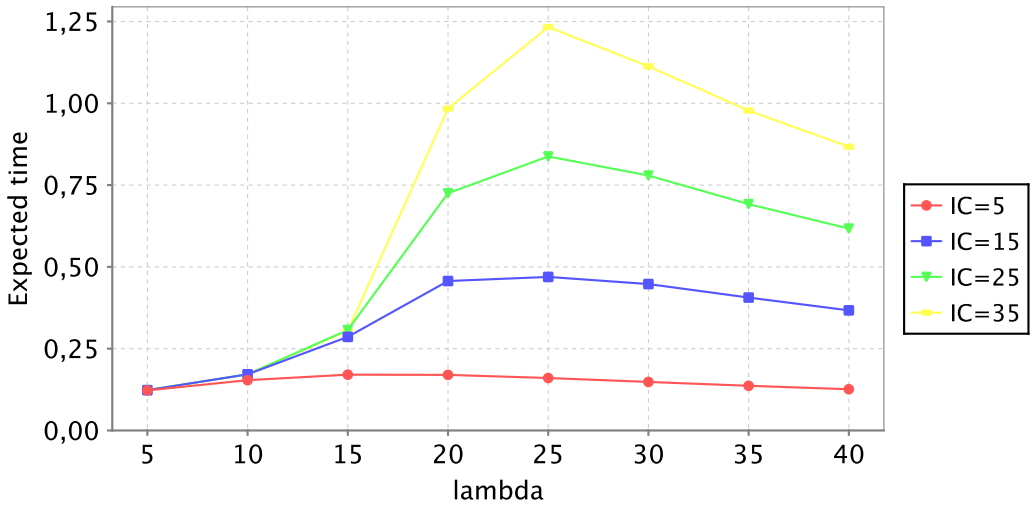

Figure 4: Estimated Response Time (N/λ)

2.2 The PRISM Model Corrected

While the response times derived in the previous section show the overall expected behavior, the absolute numbers are much larger than predicted in [8] by the formula

$$T = \frac{F}{N_c} + \frac{I_s}{1-\lambda I_s} + \frac{F}{N_s - \lambda F} + \frac{F(B_s + R_s Y_s)}{B_s R_s - \lambda F(B_s + R_s Y_s)}$$

for the average response time T of a request; see Figure 5 for the graph depicting T for the default values of the constants given in the previous section and for growing values of λ. Since [8] does not give all parameter values, we cannot accurately reproduce the figures given there; however, the curve in our Figure 5 closely resembles the curve in Figure 4 in that paper: for $\lambda = 30$, we have a response time of $T \simeq 0.2$ s; the time becomes prohibitively large for $\lambda \geq 35$.

The major culprit of the discrepancy is that the client network bandwidth N_c (determining the processing rate of S_C) only shows up in the term $\frac{F}{N_c}$ i.e. it is only used to contribute to the total response time the time for the transfer of the file over the client network. If indeed, as suggested by the verbal description, after the transfer of every block the client would with probability q request the transfer of another block, the maximum transfer rate of blocks $(N_c/B \simeq 44)$ should also impose a limit on the number of "repetition" requests.

A little inspection reveals the problem: while the PRISM model above is a faithful translation of the diagram and of the verbal description, it does *not* match the

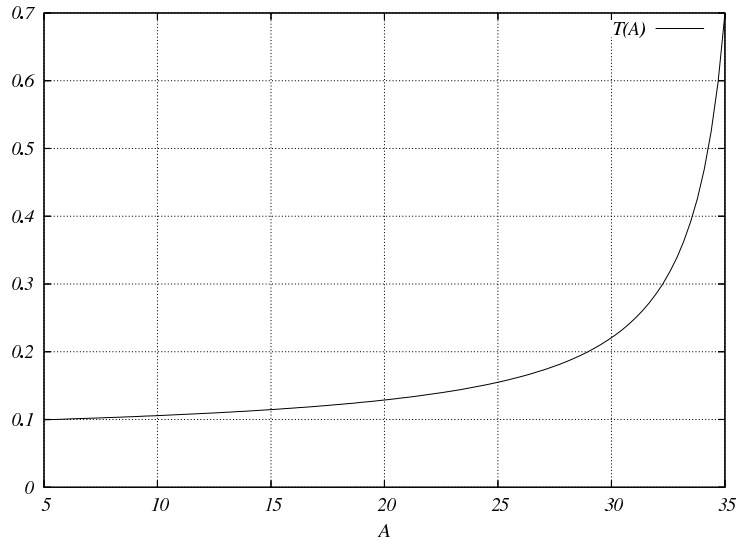

Figure 5: Response Time (Analytical Model)

model that was actually analyzed in [8]. In fact, the client network is *not* considered part of the queueing model but just used to contribute the file transfer time to the formula for T given above. The actually analyzed queuing model thus consists only of three nodes S_I, S_R, and S_S where the "branching back" of jobs occurs at/after S_S (rather than at/after S_C)! Figure 6 presents an updated illustration that describes the model analyzed in [8] is a more accurate way.

A little introspection reveals that we should actually have anticipated this. The node S_C does actually not represent a single client with network bandwidth N_c but a whole class of clients each of which is connected to the network with an average bandwidth N_c; thus also the rate for the generation of repetition requests is not constrained by a single client queue since the requests are generated from multiple clients simultaneously.

The PRISM version of this updated model is given below (the full code is given in Appendix A.2):

```
stochastic
...

module jobs
  [accept] true -> lambda : true ;
endmodule

module S_I
```

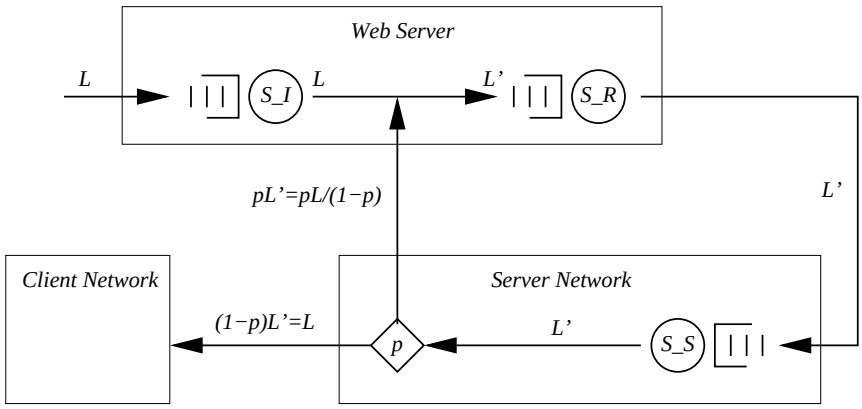

Figure 6: Corrected Network Model of a Web Server

```
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;

  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

module S_R
  irwaiting: [0..IR] init 0;
  iraccepted: bool init false;

  [forward] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [forward] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  [repeat] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [repeat] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  [isforward] irwaiting > 0 -> 1/(Ys+B/Rs):
    (irwaiting' = irwaiting-1) ;
```

```
endmodule

module S_S
  iswaiting: [0..IS] init 0;
  isaccepted: bool init false;

  [isforward] iswaiting = IS -> 1 :
    (isaccepted' = false) ;
  [isforward] iswaiting < IS -> 1 :
    (isaccepted' = true) &
    (iswaiting' = iswaiting+1) ;

  [repeat] (iswaiting > 0) & (1-q > 0) -> (Ns/Bs)*(1-q) :
    (iswaiting' = iswaiting-1) ;

  [done] (iswaiting > 0) & (q > 0) -> (Ns/Bs)*q :
    (iswaiting' = iswaiting-1) ;
endmodule
```

Performing the same analysis of acceptance ratios as above, we can determine that now the capacity of S_S becomes the critical factor. However, as can be seen in Figure 7 (computed with PRISM's JOR Jacobi Overrelaxation method and a relative termination epsilon of 10^{-4}), even for $IS = 3$, the acceptance rate remains larger than 87% up to a request rate of $\lambda = 40$; by increasing the capacity to $IQ = 33$, we get an acceptance of 99% up to $\lambda = 35$.

The corresponding numbers of pending requests are shown in Figure 8; in Figure 9, the curve for $IS = 33$ represents in the range $\lambda \leq 30$ the average response time rather accurately (compare with Figure 5); for $\lambda \geq 35$, the values become unreliable (they underestimate the request time by the factor $\frac{1}{1-P}$).

2.3 The PRISM Model Simplified

While the model of the previous section gives adequate results, its analysis by PRISM starts to take some non-negligible amount of computation time (a couple of seconds for each parameter set), which might cause a problem for larger models as those considered later in this paper. In this section, we therefore investigate whether we can streamline the model to a simpler one that gives the same results at lower costs by reducing the number of states and the number of transitions. According to our intuition, it should suffice to model only the initial node S_I by a queue that, if full, rejects further requests and monitor the acceptance ratio for this queue; however, we simplify the behavior of every other nodes such that it blocks incoming requests that find the queue full until some request leaves the queue.

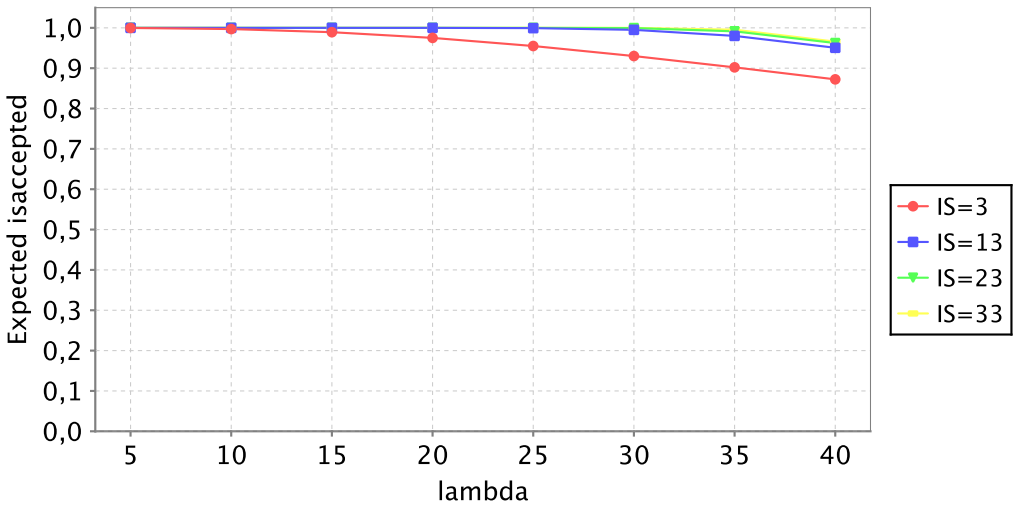

Figure 7: Estimated Acceptance Ratio for S_S (Corrected Model)

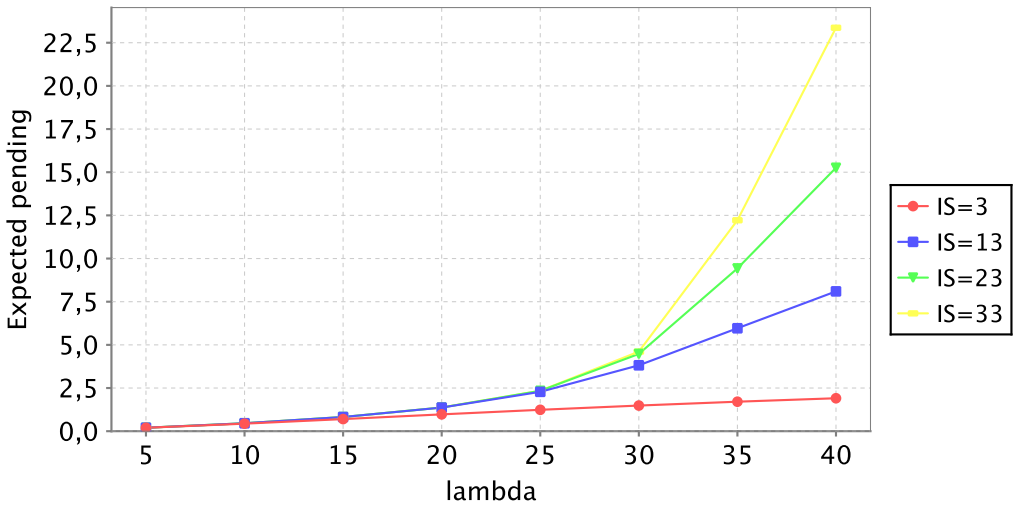

Figure 8: Number of Pending Requests N (Corrected Model)

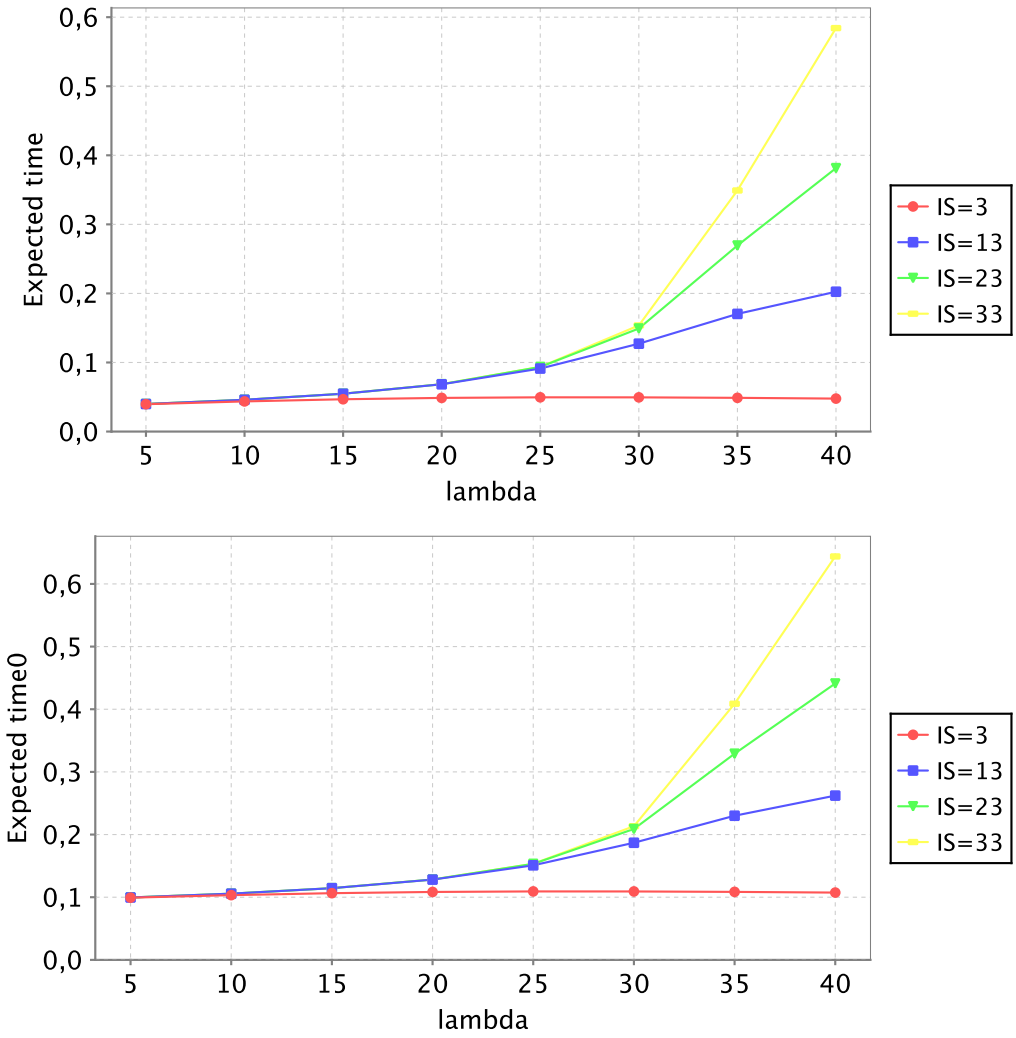

Figure 9: Estimated Response Time $N/\lambda\,(+F/N_c)$ (Corrected Model)

Thus we can save transitions and states (for the handling of acceptance flags); furthermore, it might be possible to reduce the capacities of the queues in S_R and S_S compared to those in the original model without harming the adequacy of the analysis. We then have to test the adequacy of our idea by investigating (for varying arrival rates of requests) the capacities of the queues, start with small initial capacities and increasing them until the number of pending requests (and thus the average response times) remain invariant.

The simplified model essentially looks as follows (the full code is given in Appendix A.3):

```
module jobs
  [accept] true -> lambda : true ;
endmodule

module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/I_s) :
    (waiting' = waiting-1) ;
endmodule

module S_R
  irwaiting: [0..IR] init 0;

  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [repeat] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [isforward] (irwaiting > 0) -> 1/(Y_s+B/R_s) :
    (irwaiting' = irwaiting-1) ;
endmodule

module S_S
  iswaiting: [0..IS] init 0;

  [isforward] iswaiting < IS -> 1 :
    (iswaiting' = iswaiting+1) ;
  [repeat] (iswaiting > 0) & (1-q > 0) -> (N_s/B_s)*(1-q) :
    (iswaiting' = iswaiting-1) ;
  [done] (iswaiting > 0) & (q > 0) -> (N_s/B_s)*q :
    (iswaiting' = iswaiting-1) ;
```

```
endmodule
```

Our original assumption was that it should suffice to choose *IR* and *IS* as 1, i.e. each node can only have states "empty" and "full"; any further increase should have no further effect on the average number of pending requests (and thus the average response time). We tested this hypothesis for $IA = 50$ and varying values of *IR* and *IS*. The results depicted in Figure 10 show that the assumption was not completely correct: we actually need for *IR* and *IS* a minimum value of 2 to keep the number of pending requests invariant for arrival rates $\lambda > 25$; the reason is probably the repetition "loop" between I_R and I_S which makes it essential that when sending a request in one direction, a node is still able to receive a request from the other direction.

Thus we use in the following the simplified model with $IR = IS = 2$ and varying values of *IA*. Figures 11, 12, and 13 give the corresponding estimated acceptance ratio, the number of pending requests, and the estimated response time (computed with PRISM's JOR Jacobi Overrelaxation method and a relative termination epsilon of 10^{-4}). We see that in the range $\lambda \leq 30$, the results are virtually identical to the original model (an average response time of less than 0.2 s) and get unreliable only for $\lambda \geq 35$.

The time that PRISM needs for the analysis of the simplified model is significantly smaller than the one required for the original one; the simplification may thus serve also as a "blueprint" for the investigations performed in the following sections. We must, however, not forget to perform a careful analysis to show that the results are also adequate for higher arrival rates.

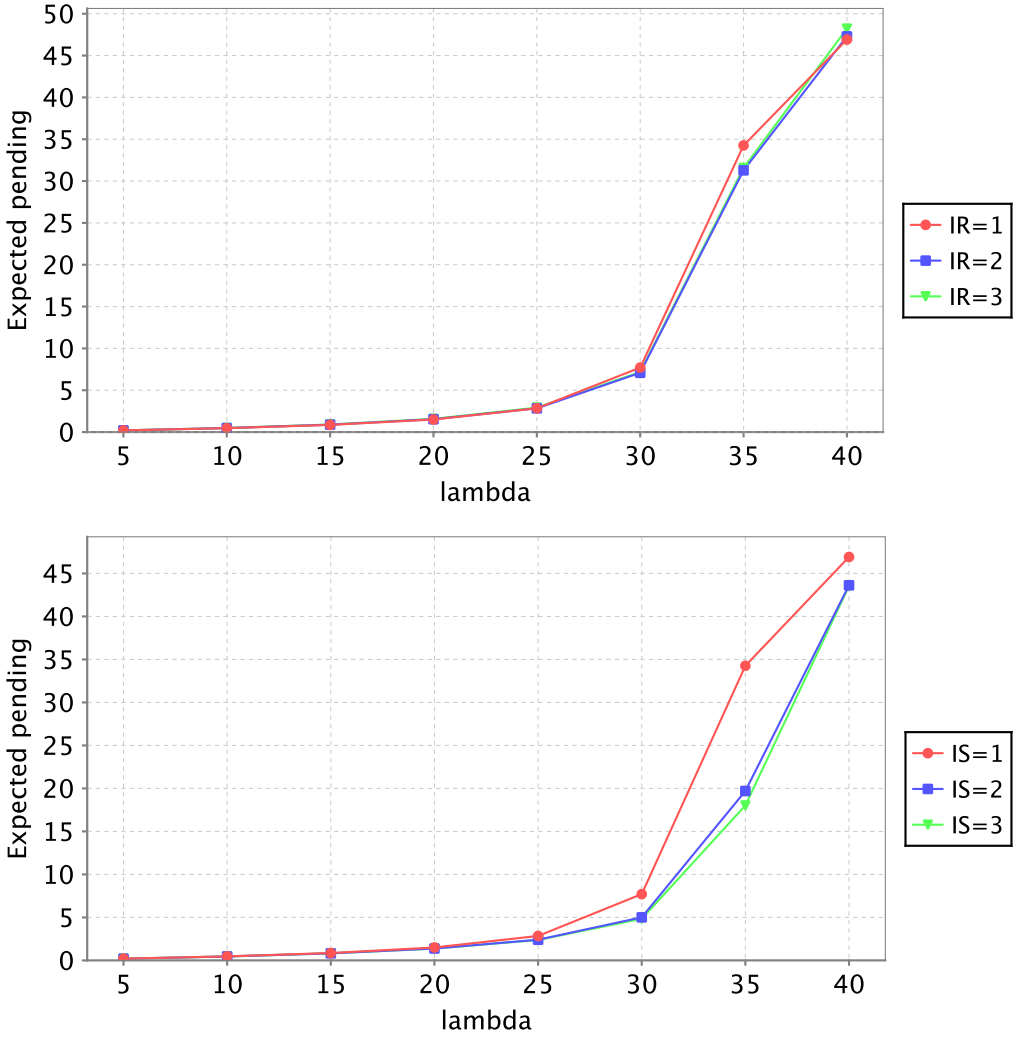

Figure 10: Number of Pending Requests N (Simplified Model)

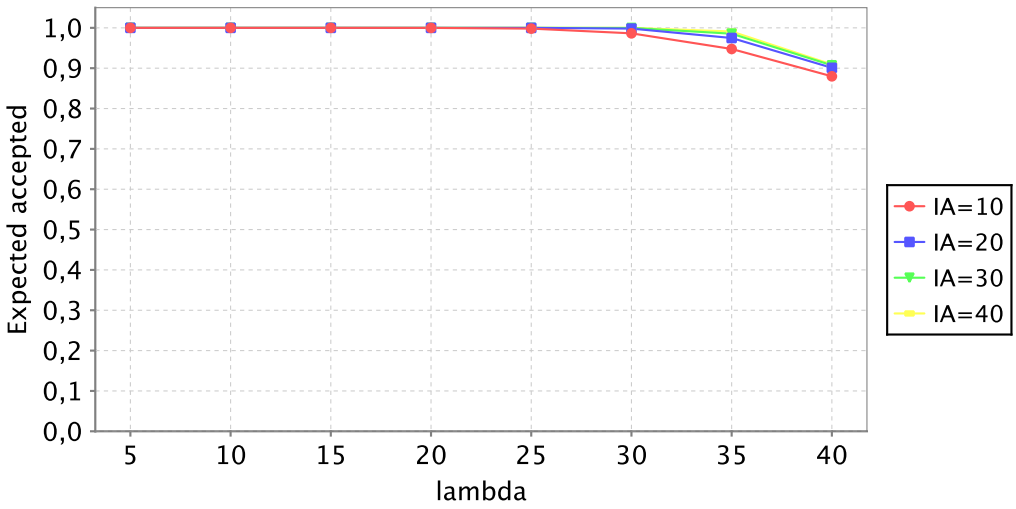

Figure 11: Estimated Acceptance Ratio for S_S (Simplified Model)

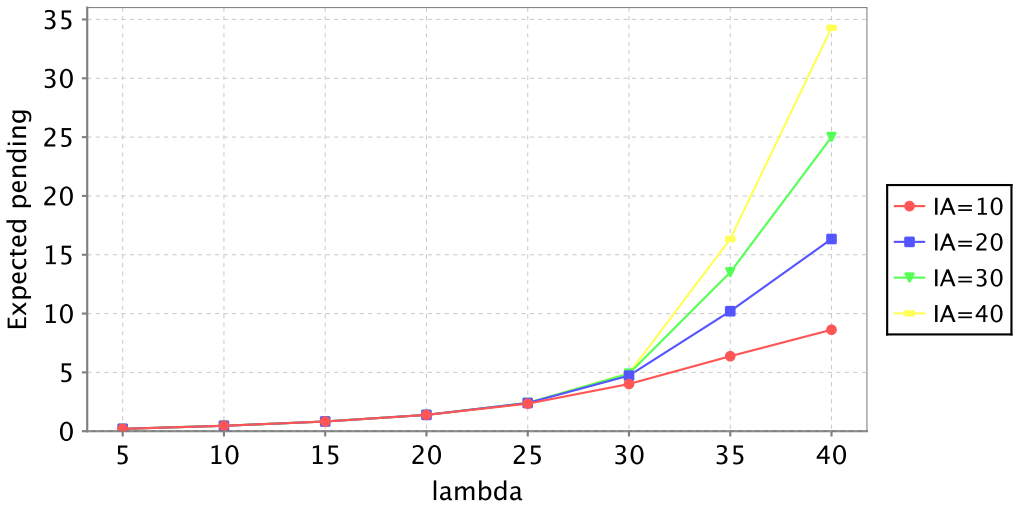

Figure 12: Number of Pending Requests N (Simplified Model)

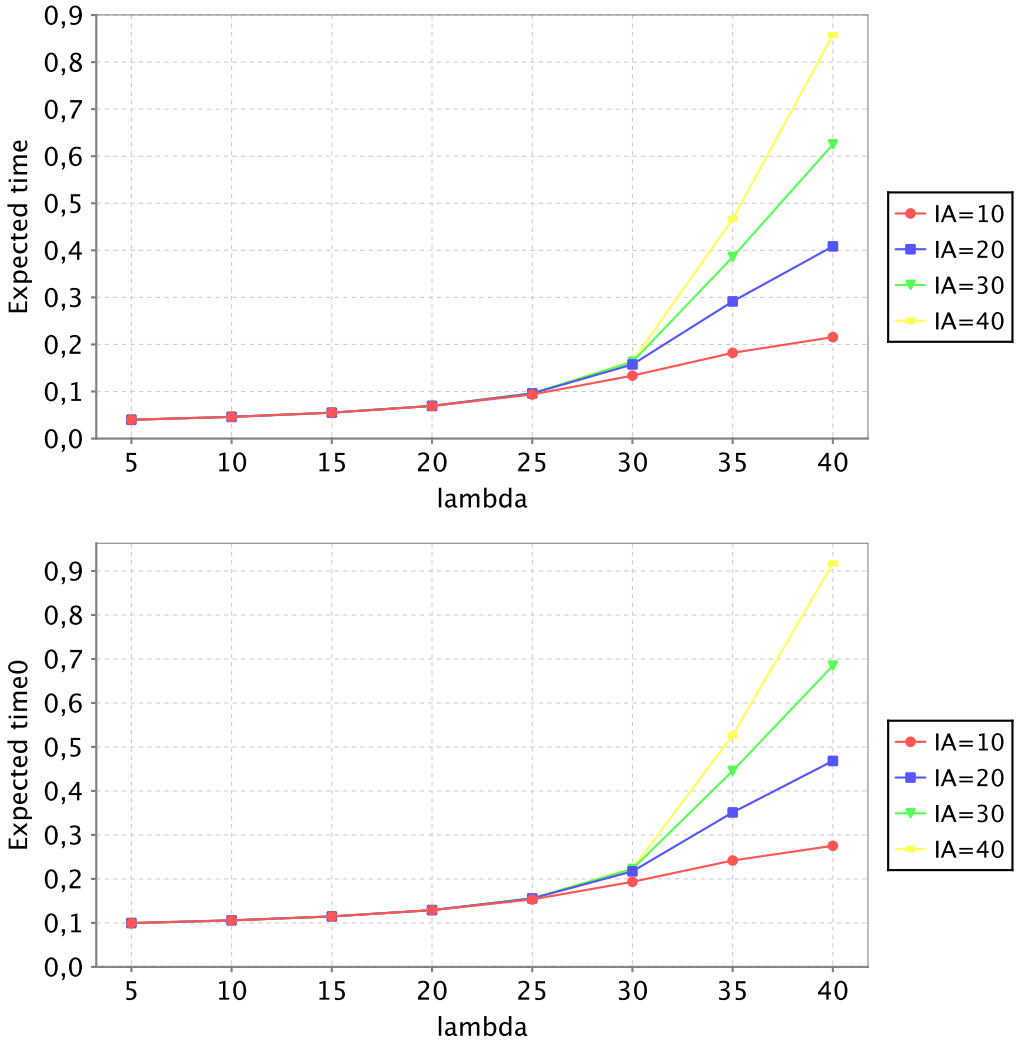

Figure 13: Estimated Response Time $N/\lambda\,(+F/N_c)$ (Simplified Model)

3 Performance Model of a Proxy Cache Server

The article [3] describes the model of a "proxy cache server" (PCS) to which the clients of a firm are connected such that web requests of the clients are first routed to the PCS. Referring to an illustration redrawn in Figure 14, the authors describe their model as follows:

> If the requested files are already stored in the PCS, the requested Web pages or files will be directly delivered to the user from the PCS. When the requested files cannot be found in the PCS, it initiates the process of fetching the desired files from the remote Web site. These new files will be stored in the firm's PCS, while a copy will be sent to the requesting user. ...
>
> The probability that the PCS can fulfill a request is p. ... We define λ_1 and λ_2 such that $\lambda = \lambda_1 + \lambda_2$ where $\lambda_1 = p\lambda$ and $\lambda_2 = (1-p)\lambda$. ...
>
> It is not unusual for the size of the requested file, F, to exceed the remote Web server's output buffer size, B_s. In this case, it may take several loops of retrieving and delivering smaller files to complete the PCS's request. This looping phenomenon is inherent in the Hyper Text Transfer Protocol (HTTP) where retrieval of the home page is followed by retrieval of embedded inline images. To model this looping, let q be the branching probability that a request from the PCS can be fulfilled at the first try; or $q = \min\{1, (B_s/F)\}$. Consequently, a $(1-q)$ proportion of the requests will loop back to the remote Web server for further processing. In equilibrium, the traffic coming out of the remote Web server toward the PCS after branching should equal the original incoming traffic, λ_2. Hence $q\lambda_2'$ equals λ_2 ... where λ_2' is the traffic leaving server network bandwidth *before* branching. ...

The performance of the model is characterized by the parameters (in addition to those already listed in Section 2)

- PCS buffer size ($B_{xc} = \alpha B_s$)
- Static PCS time ($Y_{xc} = \beta Y_s$)
- Dynamic PCS rate ($R_{xc} = \beta R_s$)
- PCS initialization time ($I_{xc} = \gamma I_s$)

with default values $\alpha = \beta = \gamma = 1$ (and, different from the values listed in Section 2, $F = 5000$ and $N_c = 16000$).

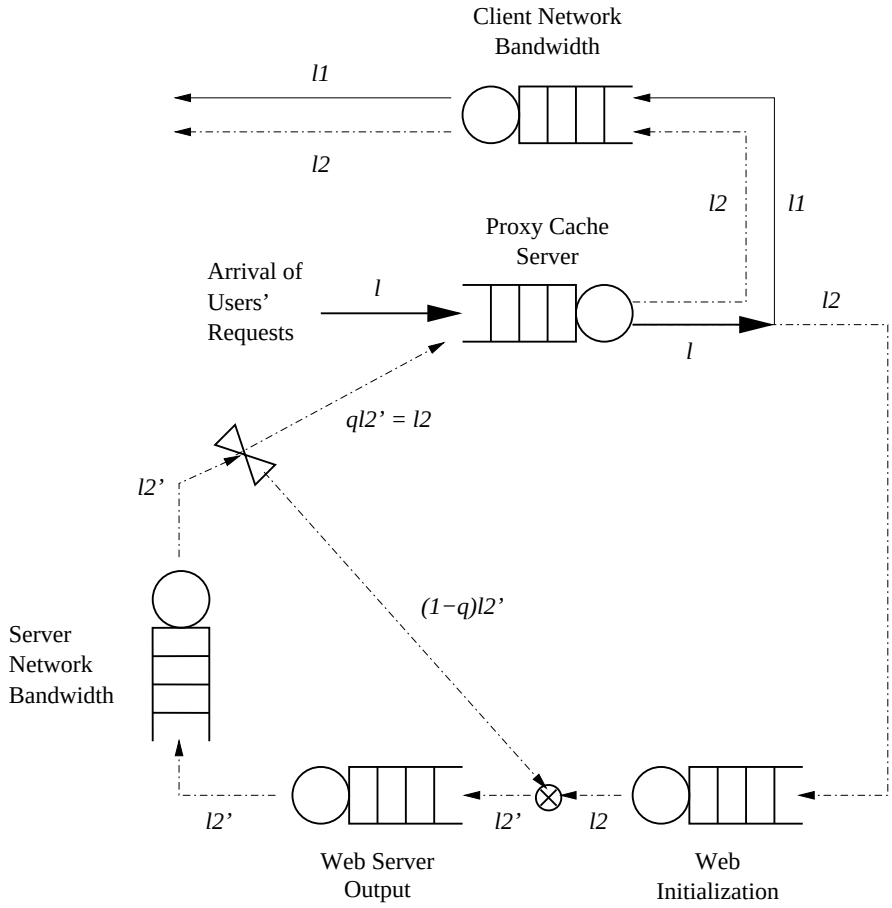

Figure 14: Queueing Network Model of a Proxy Cache Server (redrawn from [3])

The overall response time in the presence of the PCS is given as

$$\begin{aligned} T_{xc} &= \frac{1}{\frac{1}{I_{xc}}-\lambda} + p\left\{ \frac{1}{\frac{1}{\frac{F}{B_{xc}}[Y_{xc}+\frac{B_{xc}}{R_{xc}}]}-\lambda_1} + \frac{F}{N_c} \right\} \\ &\quad +(1-p)\left\{ \frac{1}{\frac{1}{I_s}-\lambda_2} + \frac{1}{\frac{1}{\frac{F}{B_s}[Y_s+\frac{B_s}{R_s}]}-\lambda_2/q} + \frac{F}{N_s} + \frac{1}{\frac{1}{\frac{F}{B_{xc}}[Y_{xc}+\frac{B_{xc}}{R_{xc}}]}-\lambda_2} + \frac{F}{N_c} \right\} \end{aligned}$$

In this formula, the first term denotes the lookup time to see if the desired files are available from the PCS, the second term (with factor p) describes the time for the content to be delivered to the requesting user, and the third term (with factor $1-p$) indicates the time required from the time the PCS initiates the fetching of the desired files to the time the PCS delivers a copy to the requesting user.

Furthermore, it is stated that without a PCS the model reduces to the special case

$$T = \frac{1}{\frac{1}{I_s}-\lambda} + \frac{1}{\frac{1}{\frac{F}{B_s}[Y_s+\frac{B_s}{R_s}]}-\lambda/q} + \frac{F}{N_s} + \frac{F}{N_c}$$

The response times for the PCS model with various arrival rates λ and probabilities p as well as the response time for the model without PCS, are depicted in Figure 15.

3.1 The Model without PCS

It is claimed in [3] that the equation for T given above represents the special case reported in [8], but this is actually *not* the case. In [3], the only term where the server bandwidth N_s plays a role is

$$\frac{F}{N_s}$$

which indicates the time for the transfer of the file over the server network. In [8], instead the term

$$\frac{F}{N_s - \lambda F}$$

is used which can be transformed to

$$\frac{1}{\frac{N_s}{F}-\lambda}$$

which indicates the time that a request spends in a queue with arrival rate λ and departure rate $\frac{N_s}{F}$. In other words, while [8] did not treat the client network as a

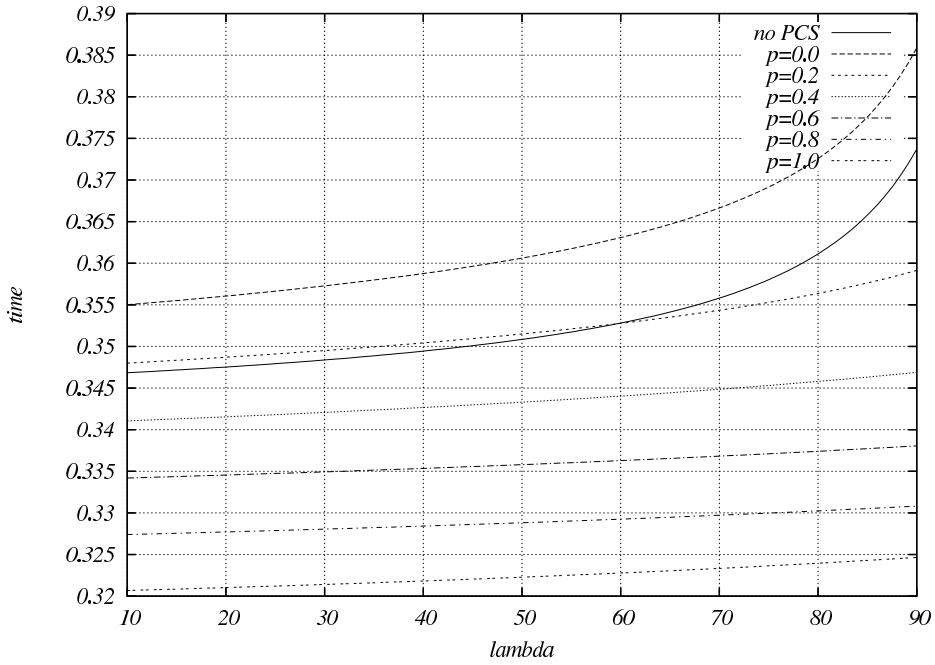

Figure 15: Response Times With and Without PCS (Analytical Model)

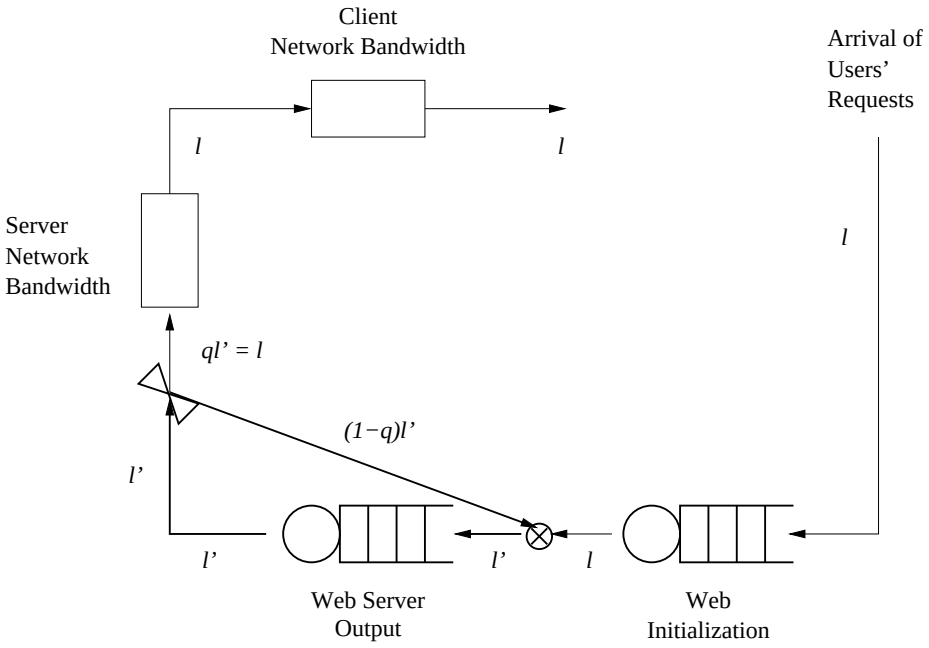

Figure 16: Corrected Queueing Network Model of Web Server

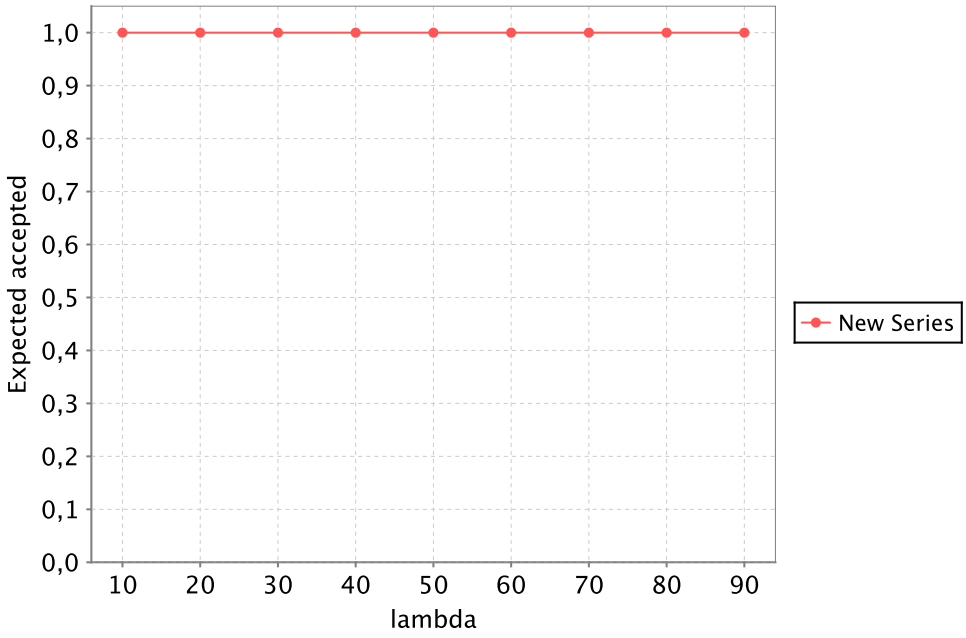

Figure 17: Estimated Acceptance Ratio

queue, it nevertheless treated the server network as such. However, in [3], neither the client network nor the server network are treated as queues; they are just used to give additional time constants for file transfers.

In Figure 16, we therefore depict the model without PCS by using only two queues rather than three. The PRISM formulation of this model is given below (for the full code see Appendix B.1):

```
module jobs
  [accept] true -> lambda : true ;
endmodule
module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
```

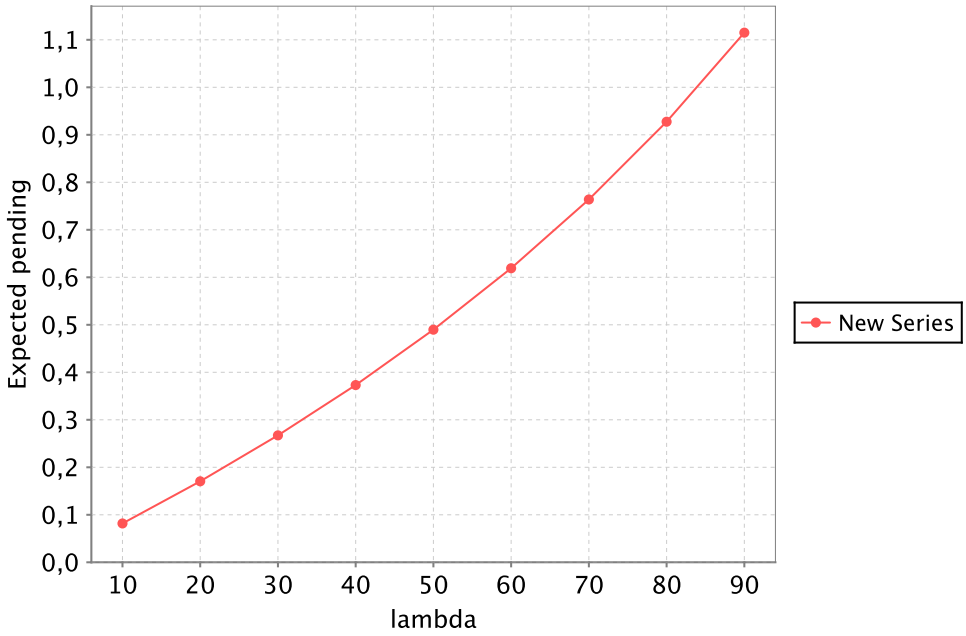

Figure 18: Number of Pending Requests (N)

```
endmodule

module S_R
  irwaiting: [0..IR] init 0;

  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [done] (irwaiting > 0) & (q > 0) ->  1/(Ys+Bs/Rs)*q :
    (irwaiting' = irwaiting-1) ;
endmodule
```

For this model, it suffices to take $IA = 10$ and $IR = 3$ to get stable results for $\lambda \leq 90$. Figure 17 shows the acceptance ratio, Figure 18 shows the average number of requests N pending in the system, and Figure 19 gives the derived estimated response times.

3.2 The Analytical Model Corrected

As it turns out, the numerical results produced by the analysis in PRISM do not accurately correspond to those depicted as "No PCS" in Figure 15, in particular

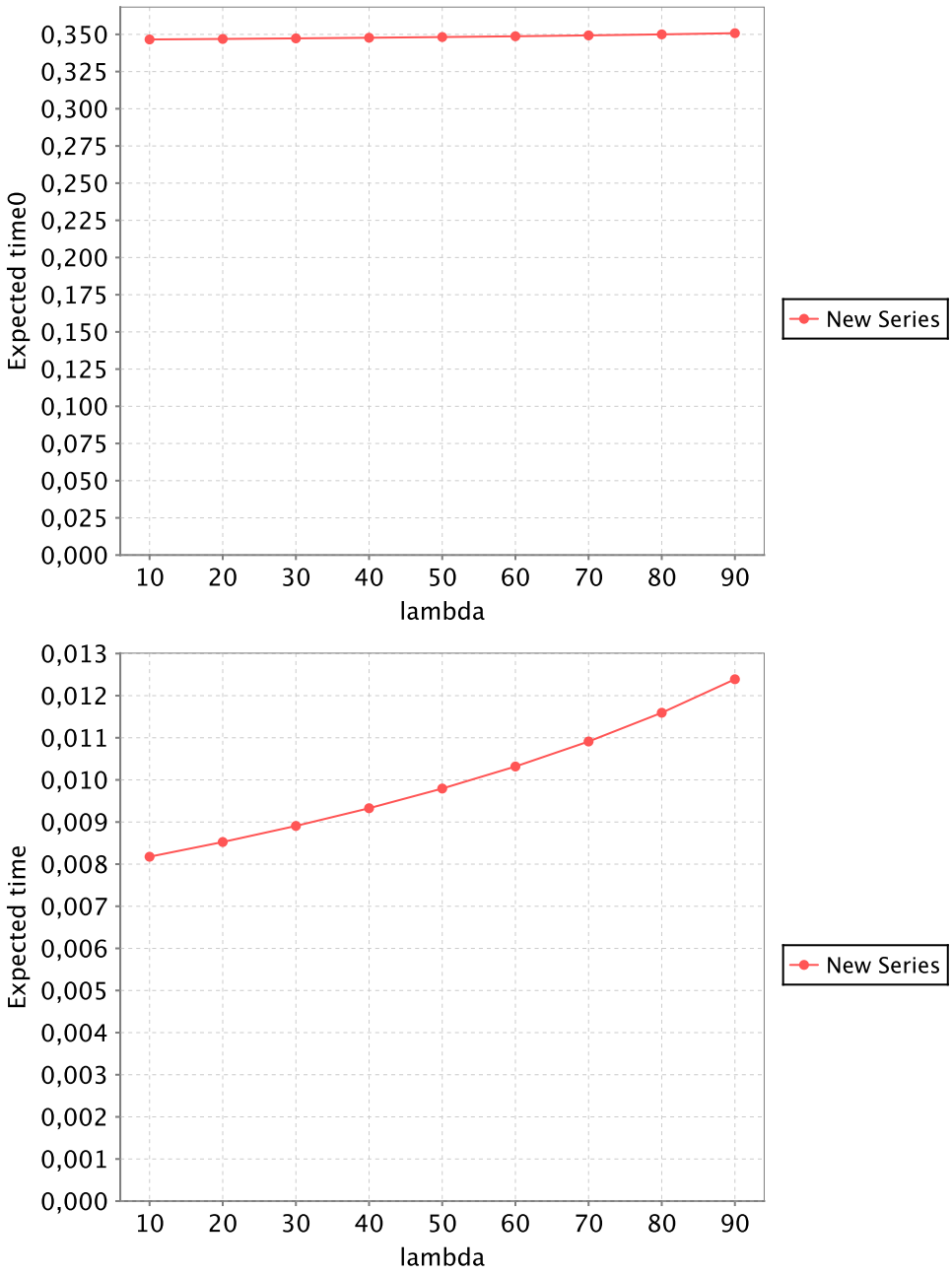

Figure 19: Estimated Response Time $N/\lambda(+\frac{F}{N_c}+\frac{F}{N_s})$

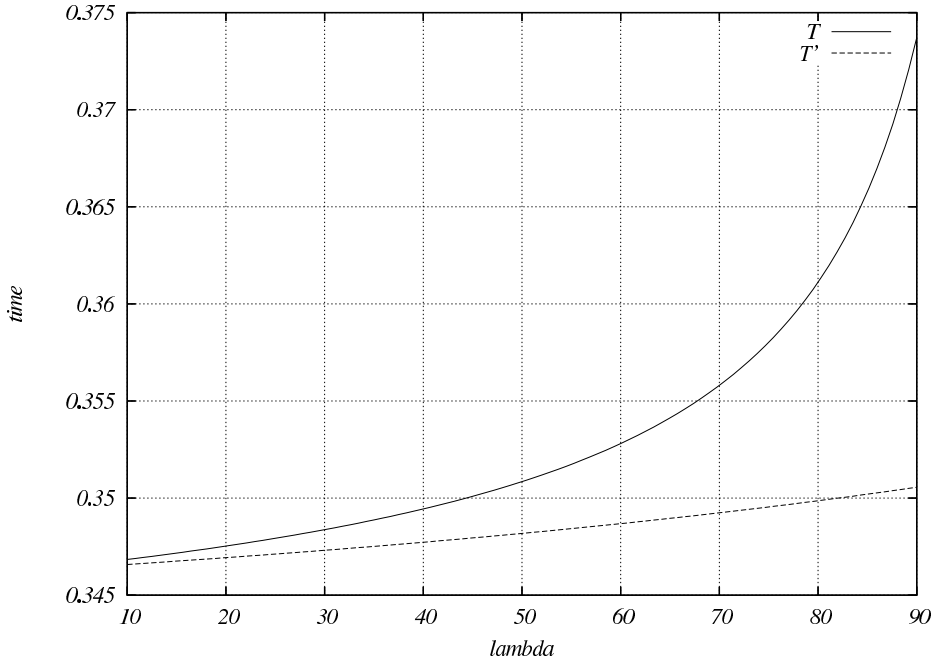

Figure 20: Response Time Without PCS (Modified Analytical Model)

for $\lambda \geq 50$. Actually the results are better described by the equation

$$T' = \frac{1}{\frac{1}{I_s} - \lambda} + \left(\frac{F}{B_s}\right) \frac{1}{\frac{1}{Y_s + \frac{B_s}{R_s}} - \lambda/q} + \frac{F}{N_s} + \frac{F}{N_c}$$

depicted in Figure 20 where the second term (modeling the "repetition loop" in the generation of the web server output) has been modified. Indeed, a closer inspection substantiates the correctness of this formulation: F/B_s represents the number of "iterations" of the corresponding queue which has arrival rate λ/q and departure rate $1/(Y_s + \frac{B_s}{R_s})$; this term now also equals the last term of the equation for T of [8] given in Section 2.2 (taking $q = \frac{B_s}{F}$).

Actually the same problem also affects the corresponding terms in the equation for T_{xc} modeling repetition loops; the correct formulation apparently is

$$\begin{aligned} T'_{xc} \quad = \quad & \frac{1}{\frac{1}{I_{xc}} - \lambda} + p \left\{ \left(\frac{F}{B_{xc}}\right) \frac{1}{\frac{1}{Y_{xc} + \frac{B_{xc}}{R_{xc}}} - \lambda/p_{xc}} + \frac{F}{N_c} \right\} \\ & + (1-p) \left\{ \frac{1}{\frac{1}{I_s} - \lambda_2} + \left(\frac{F}{B_s}\right) \frac{1}{\frac{1}{Y_s + \frac{B_s}{R_s}} - \lambda_2/q} + \frac{F}{N_s} + \left(\frac{F}{B_{xc}}\right) \frac{1}{\frac{1}{Y_{xc} + \frac{B_{xc}}{R_{xc}}} - \lambda/p_{xc}} + \frac{F}{N_c} \right\} \end{aligned}$$

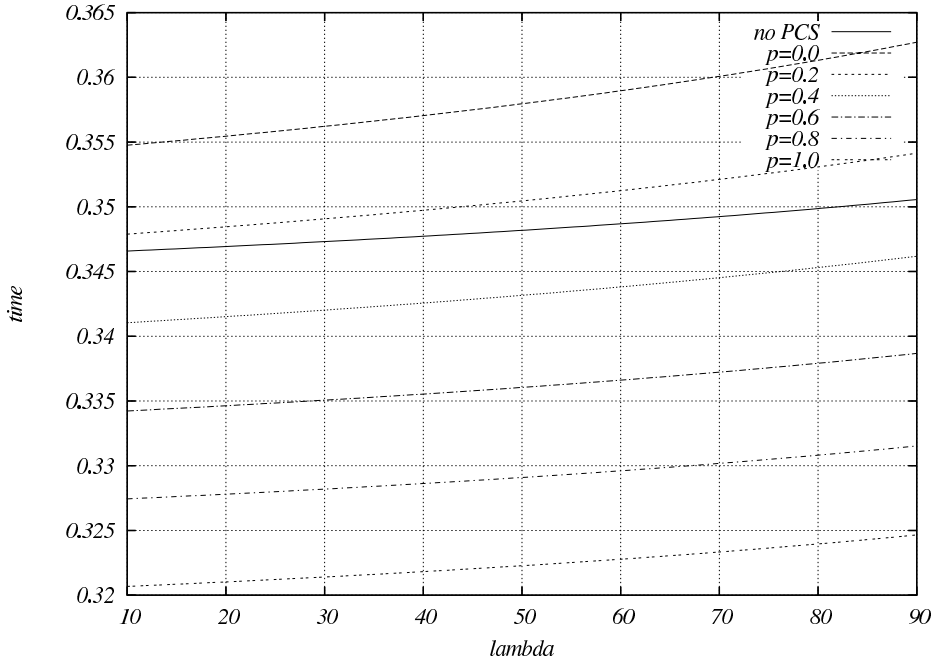

Figure 21: Response Times With and Without PCS (Modified Analytical Model)

where $p_{xc} = B_{xc}/F$ is the probability that the repetition loop is terminated (please note also the changes in the arrival rates of the corresponding terms). The corresponding numerical results are depicted in Figure 21, compare with the original results in Figure 15. However, here the difference plays only a minor role (for $p \geq 0.2$ only the third digit after the comma is affected).

3.3 The Model with PCS

Also in the model with PCS, the server network is not modelled by a queue but just by an additive constant for the transfer of the file over the network. This fact is made clear by rewriting the equation for the average response time as

$$\begin{aligned} T'_{xc} &= \frac{1}{\frac{1}{I_{xc}}-\lambda} + p\left\{\left(\frac{F}{B_{xc}}\right)\frac{1}{\frac{1}{Y_{xc}+\frac{B_{xc}}{R_{xc}}}-\lambda/p_{xc}}\right\} \\ &\quad +(1-p)\left\{\frac{1}{\frac{1}{I_s}-\lambda_2} + \left(\frac{F}{B_s}\right)\frac{1}{\frac{1}{Y_s+\frac{B_s}{R_s}}-\lambda_2/q} + \left(\frac{F}{B_{xc}}\right)\frac{1}{\frac{1}{Y_{xc}+\frac{B_{xc}}{R_{xc}}}-\lambda/p_{xc}}\right\} \\ &\quad +\left\{\frac{F}{N_c}+(1-p)\frac{F}{N_s}\right\} \end{aligned}$$

Here each fraction of form $\frac{1}{\mu-\lambda}$ indicates an occurrence of a queue with arrival rate λ and departure rate μ. We can see clearly that neither the server bandwidth N_s nor the client bandwidth N_c play a role in such fractions.

Figure 14 is therefore highly misleading; neither the server network bandwidth nor the client network bandwidth are in the model actually represented by queues; thus the queues labelled as "server network bandwidth" and "client network bandwidth" should be removed (i.e. replaced by other visual elements indicating simple delays). Furthermore, similar to the "branching" discussed in Section 2, the "branching" in this picture should not start after the "server network" but directly after the "web server output", because the repetition rate of requests is not bounded by the network bandwidth in the model.

However, on the other side actually a queue is missing (also from the description in the text); this is the one that models the repeated requests for blocks of size B_{xc} which are sent by the clients to the PCS (analogous to the repeated requests for blocks of size B_s sent by the client to the web server in the basic web server model); therefore the client indeed needs to be modeled by a queue (whose output is redirected with probability $1 - p_{xc}$ to its input), but because of the looping process, not because of the client bandwidth.

Furthermore, the dotted arrow pointing to the input of the PCS queue is actually wrong; the corresponding requests do not flow to the PCS queue (where, since the queue cannot distinguish its inputs, they might generate new requests for the web server) but directly to the client queue.

Summarizing, the actual queueing network modeled in [3] contains only four nodes in contrast to the five ones shown in Figure 14 (no queue for modeling the server bandwidth) and one of these queues does not model the "client network bandwidth" but the repetition of block requests (it could be labelled in the figure as "client output" because it plays for the repetition the same role as the queue labeled "web server output").

Figure 22 shows a revised picture that describes the model as outlined above. We implement this model in PRISM as shown below (for the full code see Appendix B.2):

```
module jobs
  [accept] true -> lambda : true ;
endmodule

module PCS
  pxwaiting: [0..IP] init 0;
  pxaccepted: bool init true;
```

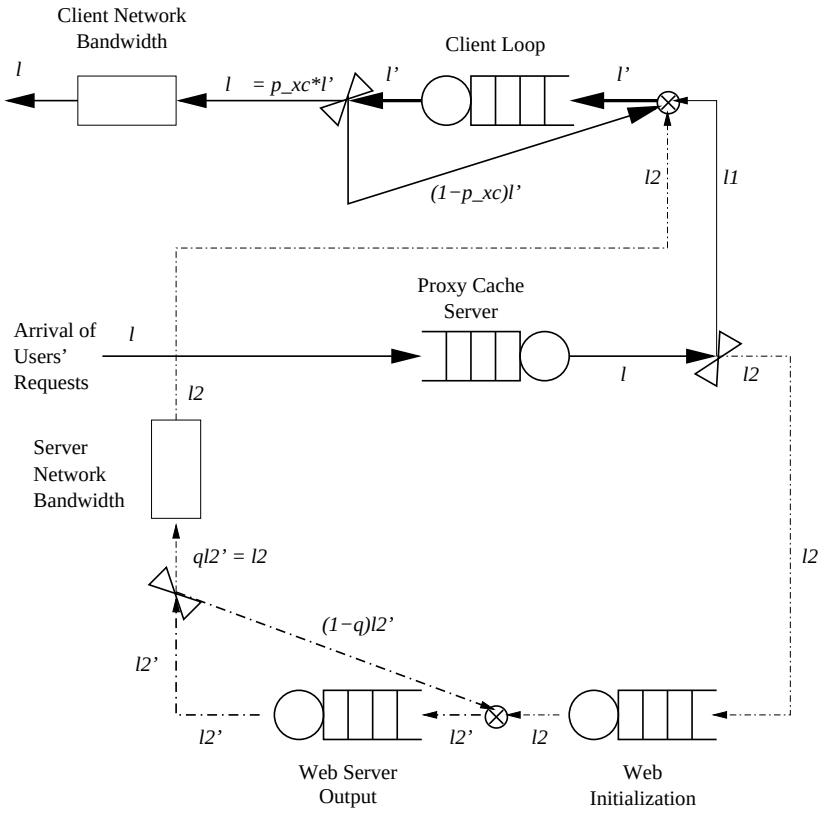

Figure 22: Corrected Queueing Network Model of Proxy Cache Server

```
  [accept] pxwaiting = IP -> 1 :
    (pxaccepted' = false) ;
  [accept] pxwaiting < IP -> 1 :
    (pxaccepted' = true) &
    (pxwaiting' = pxwaiting+1) ;
  [sforward] (pxwaiting > 0) & (1-p > 0) -> (1/Ixc)*(1-p) :
    (pxwaiting' = pxwaiting-1) ;
  [panswer] (pxwaiting > 0) & (p > 0) -> (1/Ixc)*p :
   (pxwaiting' = pxwaiting-1) ;
endmodule

module S_C
  icwaiting: [0..IC] init 0;

  [panswer] icwaiting < IC -> 1 :
    (icwaiting' = icwaiting+1) ;
  [sanswer] icwaiting < IC -> 1 :
    (icwaiting' = icwaiting+1) ;
  [done] (icwaiting > 0) & (pxc > 0) ->  1/(Yxc+Bxc/Rxc)*pxc :
    (icwaiting' = icwaiting-1) ;
endmodule

module S_I
  waiting: [0..IA] init 0;

  [sforward] waiting < IA -> 1 :
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

module S_R
  irwaiting: [0..IR] init 0;

  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [sanswer] (irwaiting > 0) & (q > 0) -> 1/(Ys+Bs/Rs)*q :
    (irwaiting' = irwaiting-1) ;
endmodule
```

Module *PCS* models the proxy cache server, module S_C the client, module S_I the initialization queue of the web server, module S_R the output queue of the web server with the following behavior:

- *PCS* returns with probability q an answer to the client (transition *canswer*) and forwards with probability $1-q$ the request to the server (transition *sforward*). The corresponding transitions "carry" the initialization time I_{xc} of the server.

- S_I buffers the incoming server request and forwards it after the initialization for further processing (transition *forward*); the transition carries the initialization time I_s of the server.

- S_R generates an output buffer with rate $1/(Y_s + \frac{B_s}{R_s})$ according to the model. However, since the request is repeated with probability $1-q$ (where $q = F/B_s$), the final result is only produced with probability q which contributes as a factor to the rate of the corresponding transition (transition *sanswer*).

- S_S models the repetition behavior of the client; a buffer of size B_{xc} is received from the PCS with rate $1/(Y_{px} + \frac{B_{xc}}{R_{xc}})$. However, the request for a buffer is repeated with probability $1-p_{xc}$ such that only with probability p_{xc} the final buffer is received and the request is completed (transition *done*).

While it would be tempting to model the repetition in S_C by generating a new request for *PCS*, this is actually wrong (as already discussed above for the model of [3]): since such a repetition request is only triggered after the PCS has already received the complete file from the web server, it is not to be treated like the incoming requests (that with probability $1-p$ generate requests for the web server); rather we just consider the probability p_{xc} with which the *final* block is received from the PCS in the rate of the termination transition *done*.

We describe in above PRISM model the actual queueing model analyzed in [3] where neither the client network nor the server network are actually modelled by queues; the response time determined of this model has to be correspondingly increased by the network transfer times $F/N_c + F/Ns$ to give the total response time. If we would, however, nevertheless like to model the server network by a queue, above model would have to be changed as follows:

```
module S_R
  irwaiting: [0..IR] init 0;
  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [repeat] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;
  [isforward] (irwaiting > 0) & (q > 0) -> 1/(Ys+Bs/Rs) :
    (irwaiting' = irwaiting-1) ;
endmodule

module S_S
  iswaiting: [0..IS] init 0;
  [isforward] iswaiting < IS -> 1 :
    (iswaiting' = iswaiting+1) ;
```

```
  [repeat] (iswaiting > 0) & (1-q > 0) -> (Ns/Bs)*(1-q) :
    (iswaiting' = iswaiting-1) ;
  [sanswer] (iswaiting > 0) & (q > 0) -> (Ns/Bs)*q :
    (iswaiting' = iswaiting-1) ;
endmodule
```

Here the module S_S represents the server network which receives file blocks of size B_s from the server such that after a delay B_s/N_s (with a probability $1-q$) either a new block can be requested (transition *repeat*) or (with a probability q) the total server answer is available to the PCS (transition *sanswer*); the rates of the transitions reflect this behavior.

In the following, we present the results of analyzing our (unmodified) model in PRISM (choosing the Jacobi method for the solution of the equation systems and a relative termination epsilon of 10^{-4}; the analysis only takes a couple of seconds). As it turns out, it suffices to take the queue capacities $IP = 5, IC = 3, IA = IR = 1$ to keep the response times essentially invariant.

Figure 23 gives the acceptance ratio for various arrival rates λ and proxy hit rates p; Figure 23 depicts the corresponding average number of requests N in the system. From this, we can estimate the time that a requests spends in the system as N/λ and the total time including the file transfer as $N/\lambda + \frac{F}{N_c} + (1-p)\frac{F}{N_s}$, see Figure 25 and compare with the curve given from the equation of T'_{xc} in Figure 21. The results are virtually identical; only for arrival rates $\lambda > 70$ and $p = 0$, we can see differences (because the web server gets saturated and the request rejection rate starts to get significant).

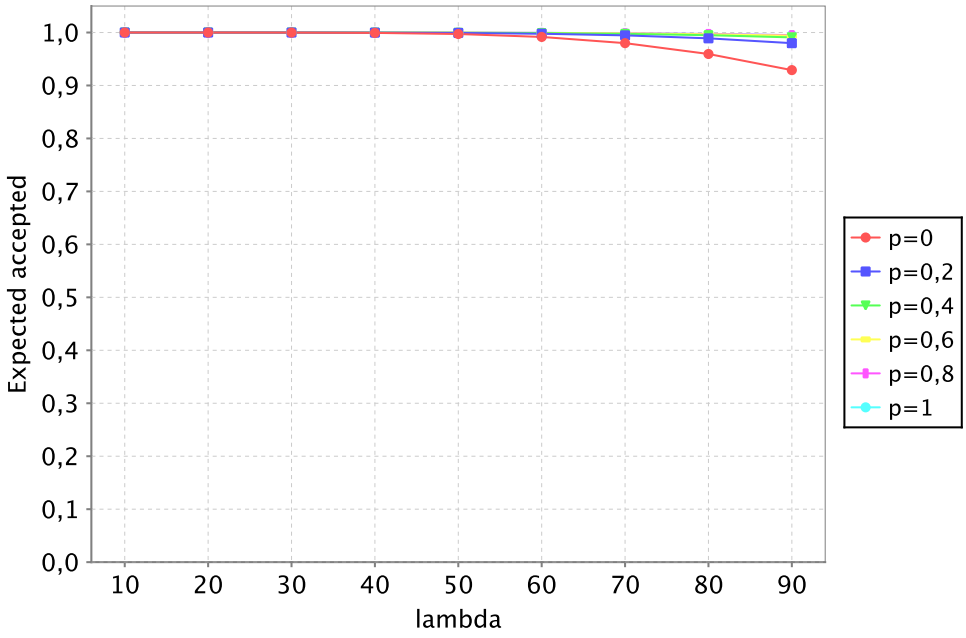

Figure 23: Estimated Acceptance Ratio

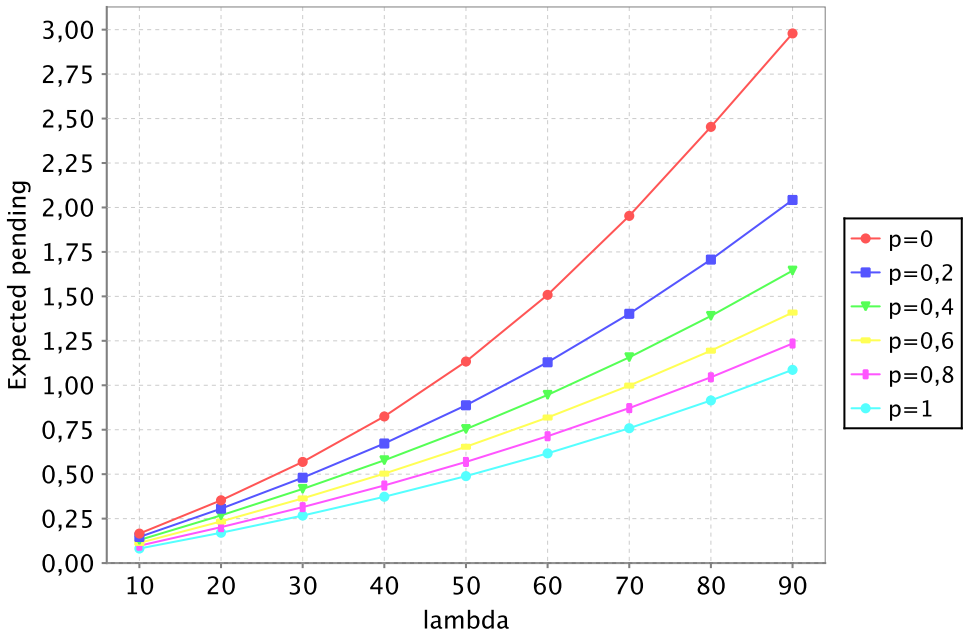

Figure 24: Number of Pending Requests (N)

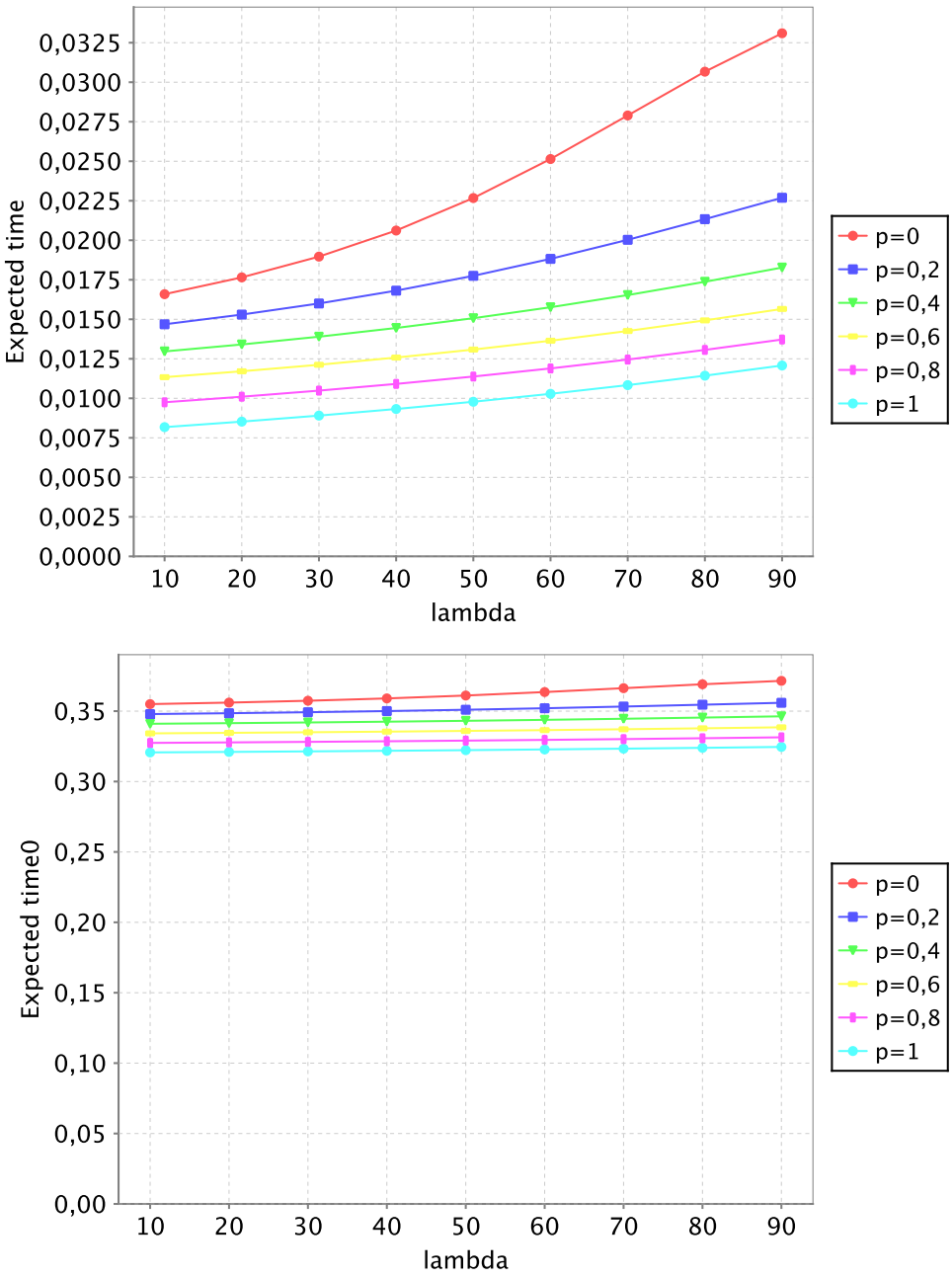

Figure 25: Estimated Response Time $N/\lambda(+\frac{F}{N_c}+(1-p)\frac{F}{N_s})$

4 Performance Model of a Proxy Cache Server with External Users

The article [3] describes the model of a "proxy cache server" (*PCS*) to which the clients of a firm are connected such that web requests of the clients are first routed to the *PCS*. If the requested file cannot be served by the *PCS*, then it downloads it from the remote Web servers and forwards to the clients. This model is refined in the article [2] by the following two issues: (1) external visits (from the rest of the Internet) are also allowed to the remote Web servers, (2) the Web servers have limited buffer. Referring to an illustration redrawn in Figure 26, the authors describe their model as follows:

> In this paper a modification of the performance model of Bose and Cheng [3] is given to deal with a more realistic case when external visitors are allowed to the remote Web servers and the Web servers have a limited buffer. For the easier understanding of the basic model and comparisons we follow the structure of the cited work.
>
> Using proxy cache server, if any information or file is requested to be downloaded, first it is checked whether the document exists on the proxy cache server. (We denote the probability of this existence by p). If the document can be found on the *PCS* then its copy is immediately transfered to the user. In the opposite case the request will be sent to the remote Web server. After the requested document arrived to the *PCS* then the copy of it is delivered to the user. ...
>
> We assume that the requests of the *PCS* users arrive according to a Poisson process with rate λ, and the external visits at the remote web server form a Poisson process with rate Λ
>
> Let F be the average of the requested file size. We define λ_1, λ_2, λ_3 and λ_5 such that: $\lambda_1 = p * \lambda$, $\lambda_2 = (1-p) * \lambda$, $\lambda_3 = \lambda_2 + \Lambda$, and $\lambda_5 = (1-P_b) * \lambda_2$ ...
>
> In our model we assume that the Web server has a buffer of capacity K. Let P_b be the probability that a request will be denied by the Web server. As it is well-known from basic queueing theory the blocking probability P_b for the $M/M/1/K$ queueing system: $P_b = P(N = K) = \frac{(1-\rho)*\rho^K}{1-\rho^{K+1}}$ ...
>
> Now we get $\rho = \frac{\lambda_3 F(Y_s R_s + B_s)}{R_s B_s}$
>
> Now we can see that the requests arrive to the buffer of the Web server according to a Poisson process with rate $\lambda_4 = (1-P_b) * \lambda_3$ ...
>
> If the size of the requested file is greater then the Web server's output buffer it will start a looping process until the delivery of all requested file's is completed. Let $q = \min\left(1, \frac{B_s}{F}\right)$ be the probability that the desired file can be

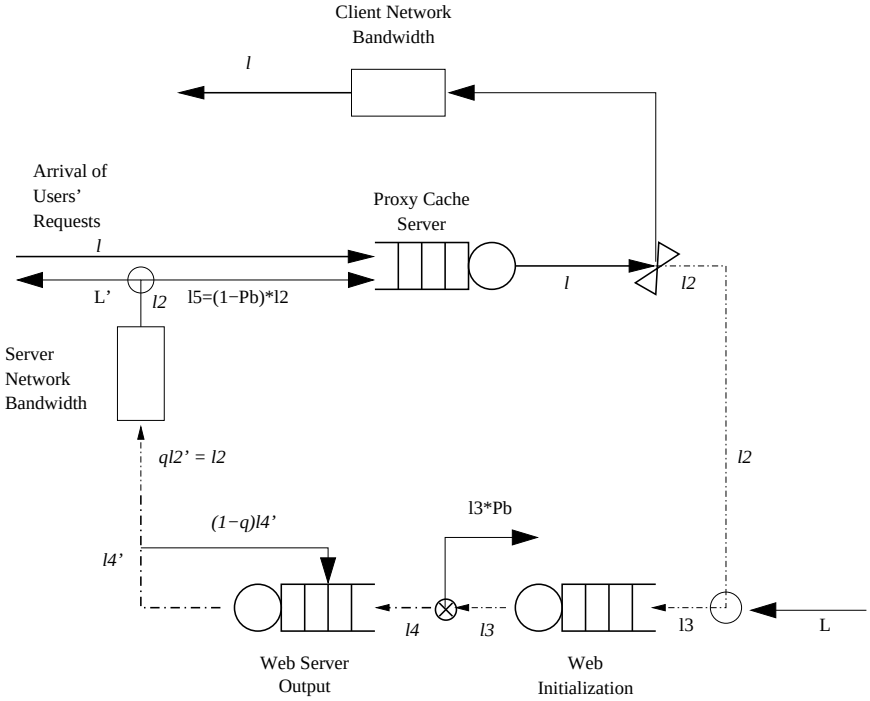

Figure 26: Queueing Network Model of a Proxy Cache Server (redrawn from [2])

> delivered at the first attempt. Let $\lambda_4^{'}$ be the rate of the requests arriving at the Web service considering the looping process. According to the conditions of equilibrium and the flow balance theory of queueing networks $\lambda_4 = q * \lambda_4^{'}$
> ...

The performance of the model is characterized by the parameters (in addition to those already listed in Section 2 and in Section 3).

- Visit rates for external users (Λ)
- Cache hit rate probability (p)
- Buffer size of the Web server given in requests ($K = 100$)

with default values $\alpha = \beta = \gamma = 1$ (and, different from the values listed in Section 2, $F = 5000$ and $N_c = 16000$).

The overall response time in the presence of the *PCS* is given as

$$
\begin{aligned}
T_{xc} \;=\; & \frac{1}{\frac{1}{I_{xc}}-\lambda} + p\left\{ \frac{1}{\frac{B_{xc}}{F*\left(Y_{xc}+\frac{B_{xc}}{R_{xc}}\right)}-\lambda_1} + \frac{F}{N_c} \right\} \\
& + (1-p)\left\{ \frac{1}{\frac{1}{I_s}-\lambda_3} + \frac{1}{\frac{B_s}{F*\left(Y_s+\frac{B_s}{R_s}\right)}-\frac{\lambda_4}{q}} + \frac{F}{N_s} + \frac{1}{\frac{B_{xc}}{F*\left(Y_{xc}+\frac{B_{xc}}{R_{xc}}\right)}-\lambda_5} + \frac{F}{N_c} \right\}
\end{aligned}
$$

In this formula, the first term denotes the lookup time to see if the desired files are available from the *PCS*, the second term (with factor p) describes the time for the content to be delivered to the requesting user, and the third term (with factor $1-p$) indicates the time required from the time the *PCS* initiates the fetching of the desired files to the time the *PCS* delivers a copy to the requesting user.

Furthermore, it is stated that without a *PCS* the model reduces to the special case

$$
T = \frac{1}{\frac{1}{I_s}-\lambda} + \frac{1}{\frac{F}{B_s}-\lambda/q} + \frac{F}{N_s} + \frac{F}{N_c}
$$

4.1 The Analytical Model Corrected

We have to notice that neither the client network nor the server network are treated as queues; thus the queues labelled as “server network bandwidth” and “client network bandwidth” should be removed and replaced by other visual elements indicating simple delays, that are just used to give additional time constants for file transfers as it is described in Section 3).

The error of the “repetition loop” that is described in Section 3 appears in the overall response time in the article [2] too.

So, actually a queue is missing; this is the one that models the repeated requests for blocks of size B_{xc} which are sent by the clients to the *PCS* (analogous to the repeated requests for blocks of size B_s sent by the client to the web server in the basic web server model); therefore the client indeed needs to be modeled by a queue (whose output is redirected with probability $1-p_{xc}$ is redirected to its input), but because of the looping process, not because of the client bandwidth.

Furthermore, the dotted arrow pointing to the input of the *PCS* queue is actually wrong; the corresponding requests do not flow to the *PCS* queue (where, since the queue cannot distinguish its inputs, they might generate new requests for the web server) but directly to the client queue.

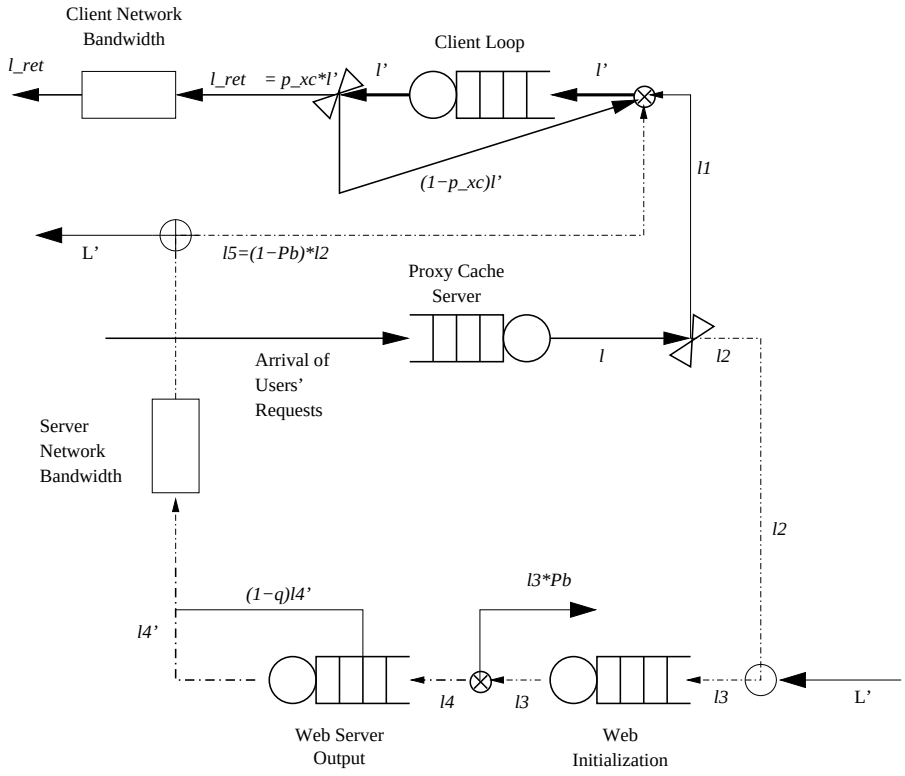

Figure 27: Corrected Model of Web Server with External Users

Summarizing, the actual, corrected queueing network modeled in [2] contains four nodes as shown in Figure 27: there are no queues for modeling the server and client network bandwidth but one for the repetition of block requests (it could be labelled in the figure as "client output" because it plays for the repetition the same role as the queue labeled "web server output").

So, the corrected overall response time in the presence of the *PCS* is given as

$$
\begin{aligned}
T_{xc} \;=\; & \frac{1}{\frac{1}{l_{xc}}-\lambda} + p\left\{ \frac{\frac{F}{B_{xc}}}{\frac{1}{\left(Y_{xc}+\frac{B_{xc}}{R_{xc}}\right)}-\frac{\lambda}{q_{xc}}} + \frac{F}{N_c} \right\} \\
& + (1-p)\left\{ \frac{1}{\frac{1}{l_s}-\lambda_3} + \frac{\frac{F}{B_s}}{\frac{1}{\left(Y_s+\frac{B_s}{R_s}\right)}-\frac{\lambda_4}{q}} + \frac{F}{N_s} + \frac{\frac{F}{B_{xc}}}{\frac{1}{\left(Y_{xc}+\frac{B_{xc}}{R_{xc}}\right)}-\frac{\lambda_1+\lambda_5}{q_{xc}}} + \frac{F}{N_c} \right\}
\end{aligned}
$$

where

$$\lambda_1 = p * \lambda, \qquad \lambda_2 = (1-p) * \lambda, \qquad \lambda_3 = \lambda_2 + \Lambda,$$
$$\rho = \frac{\lambda_3 R_S}{Y_S R_S + B_S}, \qquad P_b = P(N = K) = \frac{(1-\rho)*\rho^K}{1-\rho^{K+1}}, \quad \lambda_4 = (1-P_b) * \lambda_3,$$
$$\lambda_5 = (1-P_b) * \lambda_2.$$

The corrected overall response time without a *PCS* is given as

$$T = \frac{1}{\frac{1}{I_s} - \lambda} + \frac{\frac{F}{B_s}}{\frac{1}{\left(Y_s + \frac{B_s}{R_s}\right)} - \lambda/q} + \frac{F}{N_s} + \frac{F}{N_c}$$

4.2 PRISM Implementation

The PRISM implementation of this model can be found in Appendix C.1. It is based on the one given in Appendix B.2, which is referred as the "original" or "base" implementation in this subsection. We are now going to explain the differences of the two implementations.

We have seen that the new model has two new issues: (1) external visits (from the rest of the Internet) are also allowed to the remote Web servers, (2) the Web servers have limited buffer. In PRISM any queue must have a buffer limit. This means that we have to deal only with the first issue, i.e., we have to simulate external users. We simulate external users by the following new module:

```
module external
  [extaccept] true -> capitallambda : true ;
endmodule
```

This module generates external requests with rate Λ, which is called in the PRISM code *capitallambda*. The requests are sent to the Web servers input queue, i.e., we have to synchronize with the S_I module, therefore, the corresponding transactions have the same label, which is "extaccept". We show only those lines from the S_I module, which are not included in the base implementation:

```
module S_I
  ...
  [extaccept] waiting < IA -> 1 :
    (waiting' = waiting+1) ;
  ...
endmodule
```

After an external request has arrived, it is processed and the answer is placed in the output queue, which is simulated by the S_R module. Here one cannot distinguish between answers to external requests and answers to *PCS* requests (requests from the proxy cache server), but the two kind of answers have different impact on the system (answers to external requests have to send to the rest of the Internet, answers to *PCS* requests have to send to the proxy cache server).

So we do not know which answer belongs to which kind of answers, but we know the incoming rate of the two kind of requests: We know that external requests arrive with rate Λ, *PCS* requests arrive with rate $(1-p)*\lambda$. Let $\lambda_2 = (1-p)*\lambda$, and let $\lambda_3 = \lambda_2 + \Lambda$. So we know that λ_3 is the number of all the incoming requests per time unit, and λ_2 is the number of the incoming PCS requests per time unit, therefore, λ_2/λ_3 is the probability that a request is a *PCS* one. Since for each request we have an answer, we obtain that λ_2/λ_3 is the probability that an answer belongs to a *PCS* request.

We used this observation in the S_R module. We show only those lines from the S_R module, which are altered or not included in the base implementation:

```
module S_R
  ...
  [sanswer] (irwaiting > 0) &
            (q > 0) -> 1/(Ys+Bs/Rs)*q *(lambda2/lambda3):
    (irwaiting' = irwaiting-1) ;

  [extanswer] (irwaiting > 0) &
              (q > 0) -> 1/(Ys+Bs/Rs)*q *(1-(lambda2/lambda3)):
    (irwaiting' = irwaiting-1) ;
endmodule
```

Finally we had to rewrite the timing rewards. The original time reward was

```
rewards "time"
  true : (waiting + irwaiting + pxwaiting + icwaiting)/lambda;
endrewards
```

This is a very nice and concise form, but this hides the inner structure. A more verbose form of the original time reward could be this one:

```
rewards "time"
  true :  1 * pxwaiting / lambda +
          p * icwaiting / lambda1 +
         (1-p) * waiting /lambda2 +
         (1-p) * irwaiting / lambda2;
endrewards
```

Here $lambda1 = p * lambda$ and $lambda2 = (1-p) * lambda$. Each part of this reward has the form "probability of this branch times the actual size of the queue divided by the incoming rate". One can see that the two rewards are the same, but the second one helps us to write the new "time" and "time0" rewards:

```
rewards "time"
  true : (pxwaiting + icwaiting)/lambda +
         (1-p) * (waiting + irwaiting)/lambda3;
endrewards

rewards "time0"
  true : (pxwaiting + icwaiting)/lambda +
         (1-p) * (waiting + irwaiting)/lambda3 +
         (FS/Nc) + (1-p)*(FS/Ns);
endrewards
```

Here we have $lambda3 = lambda2 + capitallambda$. The new time reward is based on the fact that with probability $(1-p)$ the proxy cache server forwards the requests to the Web server's input queue. Since each request is served, the probability that the answer is placed in the output queue of the Web server is the same: $(1-p)$. For both queues the incoming rate is $lambda3$, because the Web server gets requests from the proxy cache server (with rate $lambda2$) and from external users (with rate $capitallambda$).

4.3 Test Results

Of course one has to counter-check the implementation against numerical results. Unfortunately we cannot use the diagrams in [2], because the article contains errors as we shown above. But we can use the results of the previous chapter. If we choose Λ to be a very small number then our implementation has to give virtually the same results as the base implementation. To check this we recall the Estimated Response Time of the PRISM code of Appendix B.2 in Figure 28 as the first diagram. The second one is the Estimated Response Time of the new implementation in case $\Lambda = 1$. We can see that the diagrams are virtually the same, except that in the new implementation we cannot simulate the case $p = 1$.

We have also numerical results based on the corrected equation in the subsection 4.1. The numerical results are as follows:

```
Parameters:
 p = 0.25
 capitallamba = 100
```

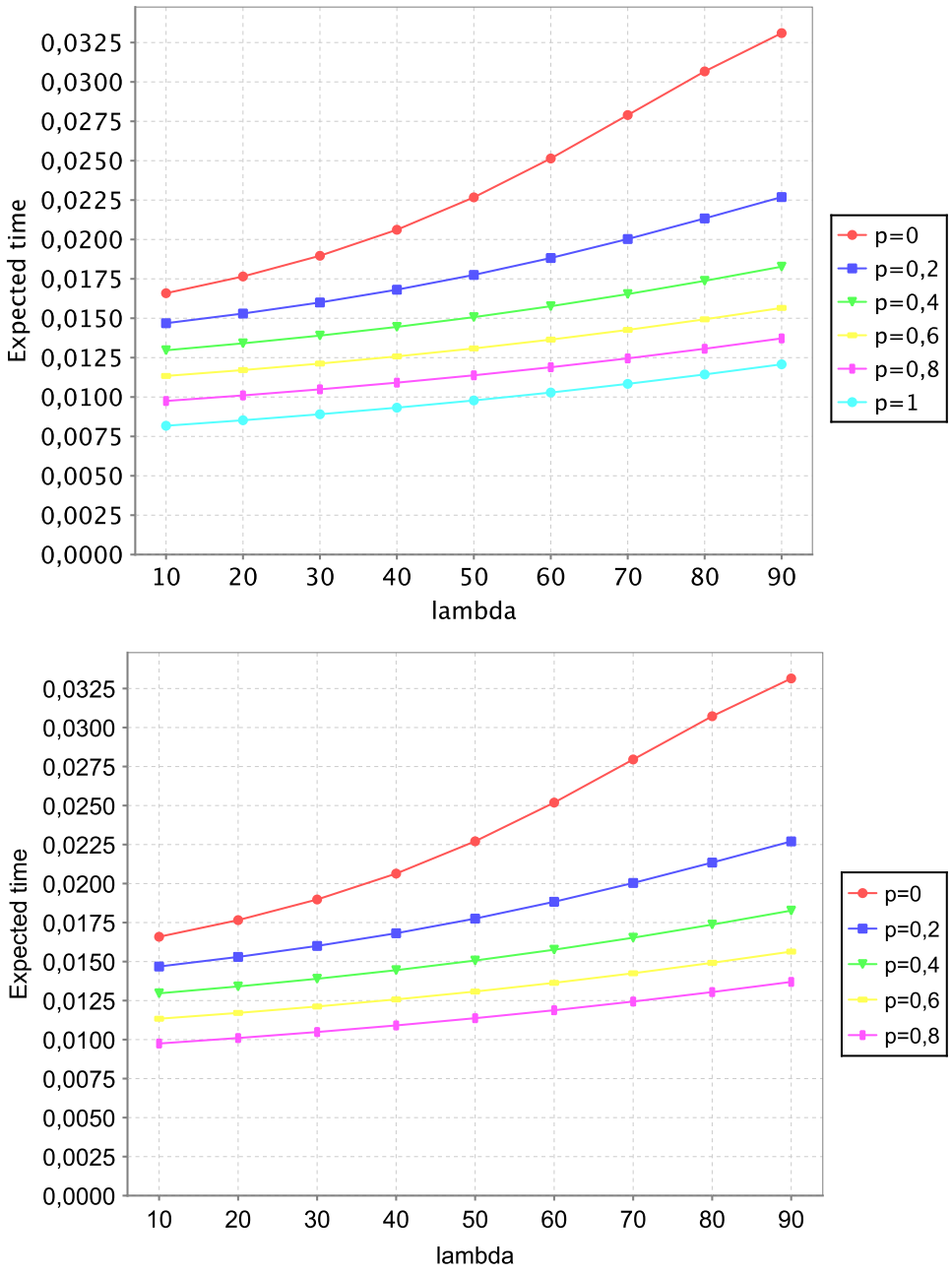

Figure 28: Estimated Response Time

```
 F = 5000

 Y_s = 0.000016,         Y_xc = 0.000016
 B_s = 2000,             B_xc = 2000
 R_s = 1250 * 1000 * 8, R_xc = 1250 * 1000 * 8
 I_s = 0.004,            I_xc = 0.004
 Ns = 1544 * 100,        Nc = 128 * 100

 mu_pcs = 1 / (Y_xc + (B_xc / R_xc))
 mu_web = 1 / (Y_s + (B_s / R_s))

Results:
 lambda: 10  0,425314728333584
 lambda: 20  0,425792543375320
 lambda: 30  0,426321265032608
 lambda: 40  0,426909696486849
 lambda: 50  0,427568831459086
 lambda: 60  0,428312592318381
 lambda: 70  0,429158892980198
 lambda: 80  0,430131208973998
 lambda: 90  0,431260965835961
```

In Figure 29 we can see the Estimated Response Time of the new implementation using the "time0" reward and the parameters shown above. The cache sizes are set as follows: $IP = 7, IC = 3, IA = 19, IR = 8$. With these, the test results are the same up to the 4-5th digit as the numerical results.

The next question was, how big should the queues be? If the queues are small then lot of requests are refused. If the queues are big then the running time of a PRISM experiment is to long. We did lot of experiments with cash sizes. The goal was to find the minimal cache size for which the acceptance ratio for all queue is at least 0.99 with incoming rates $lambda = 70$ and $capitallambda = 100$. We have found that the following cache sizes are the best choice: $IP = 7, IC = 3, IA = 19, IR = 8$ in case of $p = 0.2$. For this test we used the PRISM implementation which can be found in in Appendix C.2. We show only the acceptance ratio for the queues *IC*, *IA*, and *IR* in Figure 30, because the acceptance ratio for the queue *IP* is always more than 0.999.

We can see an interesting phenomenon that the acceptance ratio grows for *IC* as $capitallambda$ grows, but this is the opposite one would expect. The explanation of this phenomenon is that the acceptance ratio for queues *IA* and *IR* gets lower and lower as $capitallambda$ grows, and hence, more and more *PCS* request are lost. In case $lambda = 90$ and $capitallambda = 150$ around $90 * 0.99 * 0.95 = 84,645$ *PCS* requests are served, but in case $lambda = 90$ and $capitallambda = 200$ only around $90 * 0.9 * 0.91 = 73,71$ *PCS* requests are served and, of course, lower number of requests means bigger acceptance ratio.

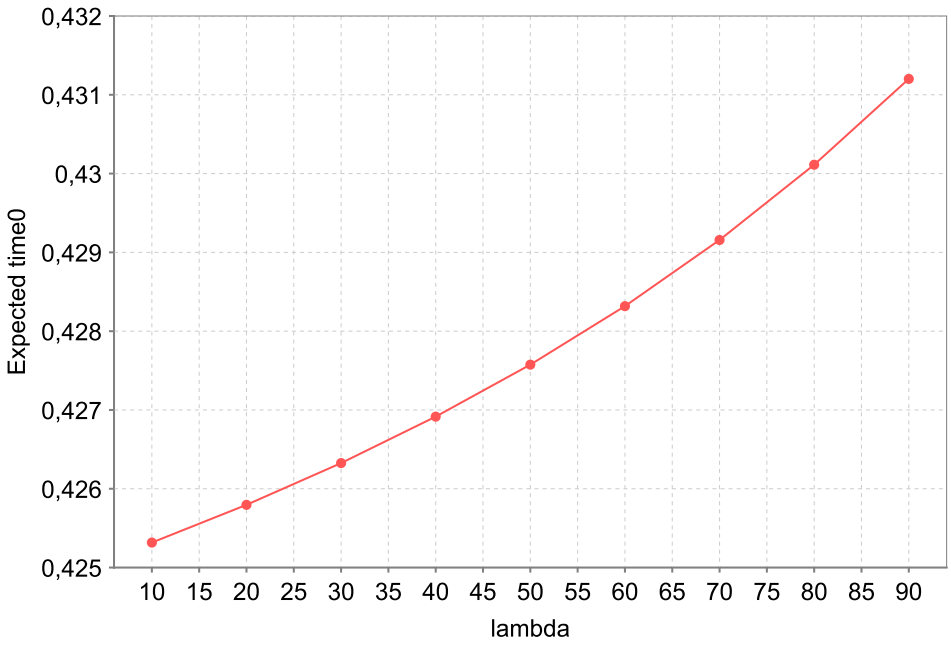

Figure 29: Estimated Response Time

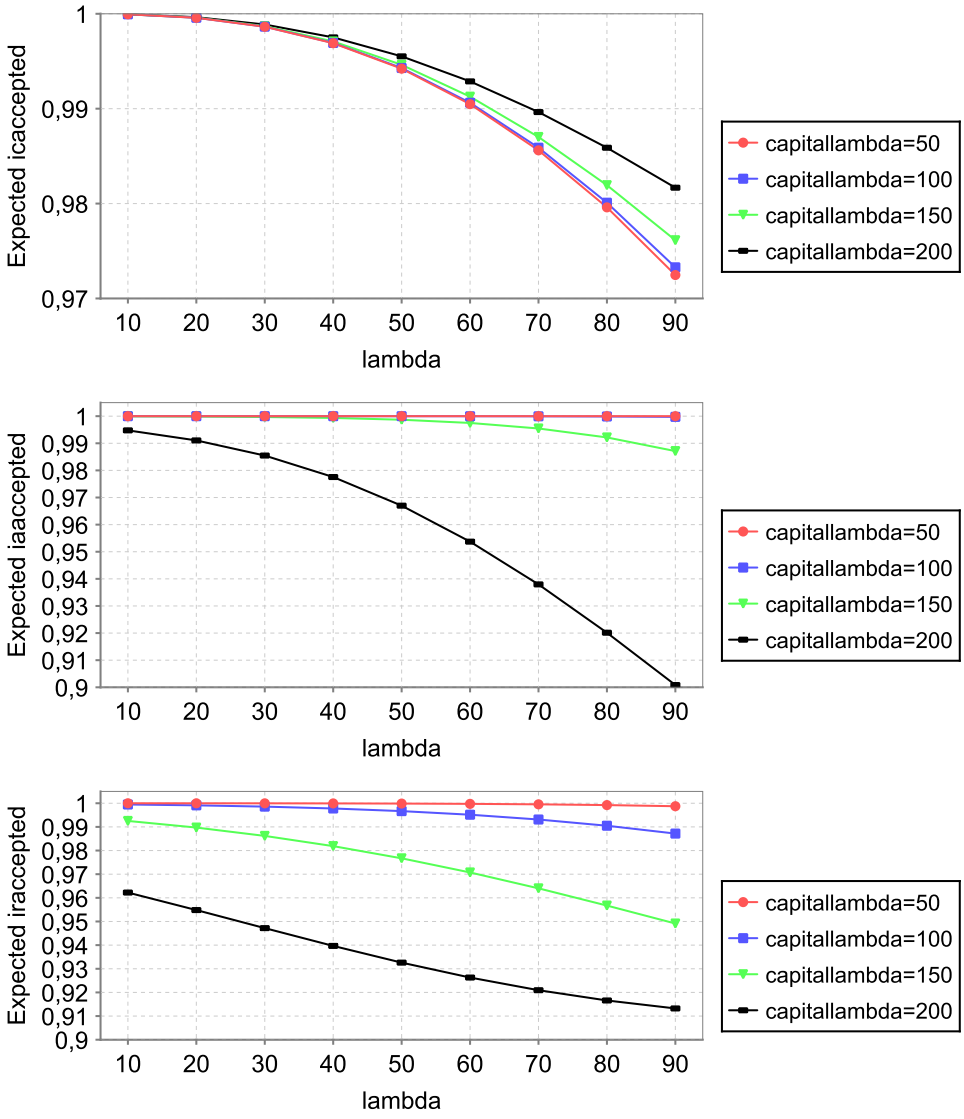

Figure 30: Estimated Response Time

5 Conclusions

The work described in this paper seems to justify the following conclusions:

- The informal models used in literature for the performance analysis of computing systems are often ambiguous. This may lead to misunderstandings of other researchers that build on top of prior work; e.g. [3] and [2] describe their results as to be based on the model presented in [8], but actually [8] models the server network by a delay element rather than by a queue which gives different results in the performance evaluation.

- The use of diagrams of queue networks is an insufficient substitute for a formal specification of a system model and a constant source of pitfalls. In [8], the diagram depicts a queue where the actual performance model uses a constant delay; likewise [3] and [2] depict queues for the server network but also use delays in their analysis. Furthermore, in all three papers there is an apparent confusion of the roles of the "loop-back" arrows which are shown in the diagrams in places that are misleading with respect to the role that they actually play in the analyzed models.

- Two of the papers [3, 2] have errors in the analytical models; these errors were only detected after trying to reproduce the results with the PRISM models. This demonstrates that performance evaluation results published in literature cannot be blindly trusted without further validation.

- Most important, after correcting the diagrams to match the actually analyzed models, a question mark has to be put on the adequacy of the models with respect to real implementations. All three [8, 3, 2] model the client network bandwidth outside the "loop" for the repeated transfer of blocks from the web (respectively proxy cache) server to the client. While the informal descriptions seem to suggest that this is intended to model the underlying network protocol, i.e. presumedly TCP, the "sliding windows" implementation of TCP lets the client interact with the server to control the flow of packets; this interaction is not handled in the presented performance models (because then the network delay must be an element of the interaction loop).

- The PRISM modeling language can be quite conveniently used to describe queueing networks by representing every network node as an automaton ("module") with explicit (qualitative and quantitative) descriptions of the interactions between automata. This forces us to be much more precise about the system model, which may first look like a nuisance, but shows its advantage when we want to argue about the adequacy of the model.

- The major limitation of a PRISM model is that it can be only used to model finitely bounded queues, while typical performance models use infinite queues. However, by careful experiments with increasing queue sizes one may determine appropriate bounds where the finite models do not significantly differ from the infinite models any more. Furthermore, since actual implementations typically use (for performance reasons) finite buffers anyway, such models more adequately describe the real-world situation; the work performed for the analysis may be therefore used to determine appropriate bounds for the implementations and reason about the expected losses of requests for these bounds.

In the future, we intend to continue this line of work by progressing towards the modeling and analysis of more complex systems that are derived from real implementations rather than from models published in literature. By this work, we hope to gain further insight into the real-world applicability of probabilistic model checking to the performance analysis of computing systems.

References

[1] Tamas Berczes, Gabor Guta, Gabor Kusper, Wolfgang Schreiner, and Janos Sztrik. Comparing the Performance Modeling Environment MOSEL and the Probabilistic Model Checker PRISM for Modeling and Analyzing Retrial Queueing Systems. Technical Report 07-17, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, 2007.

[2] Tamas Berczes and Janos Sztrik. Performance Modeling of Proxy Cache Servers. *Journal of Universal Computer Science*, 12(9):1139–1153, 2006.

[3] Indranil Bose and Hsing Kenneth Cheng. Performance Models of a Firm's Proxy Cache Server. *Decision Support Systems*, 29:47–57, 2000.

[4] Robert B. Cooper. *Introduction to Queueing Theory*. North Holland, 2nd edition, 1981.

[5] Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006, Vienna, Austria, March 27–30*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[6] G. Norman M. Kwiatkowska and D. Parker. Stochastic Model Checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270, Bertinoro, Italy, May 28 – June 2, 2007. Springer.

[7] PRISM — Probabilistic Symbolic Model Checker, November 2008. http://www.prismmodelchecker.org.

[8] Louis P. Slothouber. A Model of Web Server Performance. In *5th International World Wide Web Conference*, Paris, France, 1996.

A PRISM Model of a Web Server

A.1 A First PRISM Model

```
// ------------------------------------------------------------------------
// webServer0.sm
//
// a bounded queue approximation of the web server model presented in
//
//   Louis P. Slothouber "A Model of Web Server Performance"
//
// with parameters partially taken from
//
//   Bose and Cheng "Performance models of a firm's proxy cache server"
//
// (c) 2008 Wolfgang Schreiner
// Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria
// http://www.risc.uni-linz.ac.at
// ------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// ------------------------------------------------------------------------
// system parameters
// ------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate

// parameters from paper (units are bytes and seconds)
const double FS = 5275;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time (Bose/Cheng)
const double Ys = 0.000016; // static server time (Bose/Cheng)
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s, Bose/Cheng)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 88375;    // client network bandwidth (707 kbps)

// dervied values
const double q = func(min, 1, Bs/FS);     // probability of last server block

// queue capacities
const int IA = 3;   // capacity of server arrival queue
const int IR = 3;   // capacity of server output queue
const int IS = 3;   // capacity of internet queue of server
const int IC;       // capacity of internet queue of client

// ------------------------------------------------------------------------
// system model
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// server arrival queue
module S_I
```

```
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;
  iraccepted: bool init false;

  // request from arrival queue
  [forward] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [forward] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  // repetition request from client
  [repeat] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [repeat] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  // forwarding of full block
  [isforward] irwaiting > 0 ->  1/(Ys+Bs/Rs) :
    (irwaiting' = irwaiting-1) ;
endmodule

// internet queue of server
module S_S
  iswaiting: [0..IS] init 0;
  isaccepted: bool init false;

  [isforward] iswaiting = IS -> 1 :
    (isaccepted' = false) ;
  [isforward] iswaiting < IS -> 1 :
    (isaccepted' = true) &
    (iswaiting' = iswaiting+1) ;
  [icforward] iswaiting > 0 -> (Ns/Bs) :
    (iswaiting' = iswaiting-1) ;
endmodule

// internet queue of client
module S_C
  icwaiting: [0..IC] init 0;
  icaccepted: bool init false;

  // accept answer
  [icforward] icwaiting = IC -> 1 :
    (icaccepted' = false) ;
  [icforward] icwaiting < IC -> 1 :
    (icaccepted' = true) &
    (icwaiting' = icwaiting+1) ;
```

```
  // request is repeated with probability 1-p
  [repeat] (icwaiting > 0) & (1-q > 0) -> (Nc/Bs)*(1-q) :
    (icwaiting' = icwaiting-1) ;

  // request is completed with probability p
  [done] (icwaiting > 0) & (q > 0) -> (Nc/Bs)*q :
    (icwaiting' = icwaiting-1) ;
endmodule

// ------------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------------

rewards "allaccepted"
  accepted   : 1;
  iraccepted : 1;
  isaccepted : 1;
  icaccepted : 1;
endrewards

rewards "accepted"
  accepted : 1;
endrewards

rewards "iraccepted"
  iraccepted : 1;
endrewards

rewards "isaccepted"
  isaccepted : 1;
endrewards

rewards "icaccepted"
  icaccepted : 1;
endrewards

rewards "pending"
  true : waiting + irwaiting + iswaiting + icwaiting;
endrewards

rewards "time"
  true : (waiting + irwaiting + iswaiting + icwaiting)/lambda;
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-3, and model constants
//
//   lambda = 5..40, IA = 3, IR = 3, IS = 3, IC = 5..35
//
// gives reliable results for lambda <= 20

// estimated percentage of requests accepted by queues
R{"allaccepted"}=? [ S ]
R{"accepted"}=? [ S ]
R{"iraccepted"}=? [ S ]
R{"isaccepted"}=? [ S ]
R{"icaccepted"}=? [ S ]
```

```
// number of requests pending in system
R{"pending"}=? [ S ]

// average request response time
R{"time"}=? [ S ]
```

A.2 The PRISM Model Corrected

```
// -------------------------------------------------------------------------
// webServer1.sm
//
// a bounded queue approximation of the web server model presented in
//
//   Louis P. Slothouber "A Model of Web Server Performance"
//
// with parameters partially taken from
//
//   Bose and Cheng "Performance models of a firm's proxy cache server"
//
// (c) 2008 Wolfgang Schreiner
// Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria
// http://www.risc.uni-linz.ac.at
// -------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// -------------------------------------------------------------------------
// system parameters
// -------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate

// parameters from paper (units are bytes and seconds)
const double FS = 5275;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time (Bose/Cheng)
const double Ys = 0.000016; // static server time (Bose/Cheng)
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s, Bose/Cheng)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 88375;    // client network bandwidth (707 kbps)

// dervied values
const double q = func(min, 1, Bs/FS);     // probability of last server block

// queue capacities
const int IA = 3;   // capacity of server arrival queue
const int IR = 3;   // capacity of server output queue
const int IS;       // capacity of internet queue of server

// -------------------------------------------------------------------------
// system model: client bandwidth is ignored
// (file transfer time to be added to total request time)
// -------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
```

```
  [accept] true -> lambda : true ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;
  iraccepted: bool init false;

  // request from arrival queue
  [forward] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [forward] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  // repetition request from client
  [repeat] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [repeat] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  // forwarding of block
  [isforward] (irwaiting > 0) -> 1/(Ys+Bs/Rs) :
    (irwaiting' = irwaiting-1) ;
endmodule

// internet queue of server
module S_S
  iswaiting: [0..IS] init 0;
  isaccepted: bool init false;

  [isforward] iswaiting = IS -> 1 :
    (isaccepted' = false) ;
  [isforward] iswaiting < IS -> 1 :
    (isaccepted' = true) &
    (iswaiting' = iswaiting+1) ;

  // request is repeated with probability 1-q
  [repeat] (iswaiting > 0) & (1-q > 0) -> (Ns/Bs)*(1-q) :
    (iswaiting' = iswaiting-1) ;

  // request is completed with probability q
  [done] (iswaiting > 0) & (q > 0) -> (Ns/Bs)*q :
    (iswaiting' = iswaiting-1) ;
endmodule

// ------------------------------------------------------------------------
```

```
// system rewards
// ------------------------------------------------------------------------

rewards "allaccepted"
  accepted   : 1;
  iraccepted : 1;
  isaccepted : 1;
endrewards

rewards "accepted"
  accepted : 1;
endrewards

rewards "iraccepted"
  iraccepted : 1;
endrewards

rewards "isaccepted"
  isaccepted : 1;
endrewards

rewards "pending"
  true : waiting + irwaiting + iswaiting;
endrewards

rewards "time"
  true : (waiting + irwaiting + iswaiting)/lambda;
endrewards

rewards "time0"
  true : (waiting + irwaiting + iswaiting)/lambda+(FS/Nc);
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-3, and model constants
//
//   lambda = 5..40, IA = 3, IR = 3, IS = 3..33
//
// gives reliable results for lambda <= 35

// estimated percentage of requests accepted by queues
R{"allaccepted"}=? [ S ]
R{"accepted"}=? [ S ]
R{"iraccepted"}=? [ S ]
R{"isaccepted"}=? [ S ]

// number of requests pending in system
R{"pending"}=? [ S ]

// time request spends in system
R{"time"}=? [ S ]

// time request spends in system plus file transfer time
R{"time0"}=? [ S ]
```

A.3 The PRISM Model Simplified

```
// ------------------------------------------------------------------------
// webServer2.sm
//
// a bounded queue approximation of the web server model presented in
//
//   Louis P. Slothouber "A Model of Web Server Performance"
//
// with parameters partially taken from
//
//   Bose and Cheng "Performance models of a firm's proxy cache server"
//
// (c) 2008 Wolfgang Schreiner
// Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria
// http://www.risc.uni-linz.ac.at
// ------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// ------------------------------------------------------------------------
// system parameters
// ------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate

// parameters from paper (units are bytes and seconds)
const double FS = 5275;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time (Bose/Cheng)
const double Ys = 0.000016; // static server time (Bose/Cheng)
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s, Bose/Cheng)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 88375;    // client network bandwidth (707 kbps)

// dervied values
const double q = func(min, 1, Bs/FS);     // probability of last server block

// queue capacities
const int IA;       // capacity of server arrival queue
const int IR = 2;   // capacity of server output queue
const int IS = 2;   // capacity of internet queue of server

// ------------------------------------------------------------------------
// system model: client bandwidth is ignored
// (file transfer time to be added to total request time)
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;
```

```
  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;

  // request from arrival queue
  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;

  // repetition request from client
  [repeat] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;

  // forwarding of block
  [isforward] (irwaiting > 0) -> 1/(Ys+Bs/Rs) :
    (irwaiting' = irwaiting-1) ;
endmodule

// internet queue of server
module S_S
  iswaiting: [0..IS] init 0;

  [isforward] iswaiting < IS -> 1 :
    (iswaiting' = iswaiting+1) ;

  // request is repeated with probability 1-q
  [repeat] (iswaiting > 0) & (1-q > 0) -> (Ns/Bs)*(1-q) :
    (iswaiting' = iswaiting-1) ;

  // request is completed with probability q
  [done] (iswaiting > 0) & (q > 0) -> (Ns/Bs)*q :
    (iswaiting' = iswaiting-1) ;
endmodule

// ------------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------------

rewards "accepted"
  accepted   : 1;
endrewards

rewards "pending"
  true : waiting + irwaiting + iswaiting;
endrewards

rewards "time"
  true : (waiting + irwaiting + iswaiting)/lambda;
endrewards

rewards "time0"
  true : (waiting + irwaiting + iswaiting)/lambda+(FS/Nc);
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-3, and model constants
//
//   lambda = 5..40, IA = 10..40, IR = 2, IS = 2
//
// gives reliable results for lambda <= 30

// estimated percentage of requests accepted by queues
R{"accepted"}=? [ S ]

// number of requests pending in system
R{"pending"}=? [ S ]

// time request spends in system
R{"time"}=? [ S ]

// time request spends in system plus file transfer time
R{"time0"}=? [ S ]
```

B PRISM Model of a Proxy Cache Server

B.1 The Model without PCS

```
// ------------------------------------------------------------------------
// webServer3.sm
//
// a bounded queue approximation of the web server model without proxy used in
//
//   Bose and Cheng "Performance models of a firm's proxy cache server"
//
// (c) 2008 Wolfgang Schreiner
// Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria
// http://www.risc.uni-linz.ac.at
// ------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// ------------------------------------------------------------------------
// system parameters
// ------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate

// parameters from paper (units are bytes and seconds)
const double FS = 5000;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time
const double Ys = 0.000016; // static server time
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 16000;    // client network bandwidth (128 kbps)
```

```
// web server and proxa cache server blocks
const double q = func(min, 1, Bs/FS);     // probability of last server block

// queue capacities
const int IA;   // capacity of server arrival queue
const int IR;   // capacity of server output queue

// ------------------------------------------------------------------------
// system model: server and client bandwidth is ignored
// (file transfer time has to be added to total request time)
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;
  accepted: bool init false;

  [accept] waiting = IA -> 1 :
    (accepted' = false) ;
  [accept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server processing queue
module S_R
  irwaiting: [0..IR] init 0;

  // request from arrival queue
  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;

  // request is completed with probability p
  [done] (irwaiting > 0) & (q > 0) -> 1/(Ys+Bs/Rs)*q :
    (irwaiting' = irwaiting-1) ;
endmodule

// ------------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------------

rewards "accepted"
  accepted : 1;
endrewards

rewards "pending"
  true : waiting + irwaiting;
endrewards

rewards "time"
  true : (waiting + irwaiting)/lambda;
endrewards

rewards "time0"
  true : (waiting + irwaiting)/lambda + (FS/Nc)+(FS/Ns);
```

```
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-4, and model constants
//
//   lambda = 10..90, IA = 10, IR = 3
//
// gives reliable results for all lambda

// estimation of acceptance ratio
R{"accepted"}=? [ S ]

// number of requests pending in system
R{"pending"}=? [ S ]

// time request spends in queue
R{"time"}=? [ S ]

// total time including network transfer
R{"time0"}=? [ S ]
```

B.2 The Model with PCS

```
// -----------------------------------------------------------------------------
// webProxy.sm
//
// a bounded queue approximation of the web server proxy model presented in
//
//   Bose and Cheng "Performance models of a firm's proxy cache server"
//
// (c) 2008 Wolfgang Schreiner
// Research Institute for Symbolic Computation (RISC)
// Johannes Kepler University, Linz, Austria
// http://www.risc.uni-linz.ac.at
// -----------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// -----------------------------------------------------------------------------
// system parameters
// -----------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate
const double p;             // probability that file is on PCS

// proxy cache server performance
const double alpha = 1;     // alpha = Bxc/Bs
const double beta = 1;      // beta = Rxc/Rs = Yxc/Ys
const double gamma = 1 ;    // gamma = Ixc/Is;

// parameters from paper (units are bytes and seconds)
const double FS = 5000;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time
```

```
const double Ys = 0.000016; // static server time
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 16000;    // client network bandwidth (128 kbps)

// proxy characteristics
const double Bxc = alpha*Bs; // proxy buffer size
const double Yxc = beta*Ys;  // static proxy time
const double Rxc = beta*Rs;  // dynamic proxy rate
const double Ixc = gamma*Is; // proxy initialization time

// web server and proxa cache server blocks
const double q = func(min, 1, Bs/FS);     // probability of last server block
const double pxc  = func(min, 1, Bxc/FS); // probability of last proxy block

// queue capacities
const int IP;   // capacity of proxy queue
const int IC;   // capacity of client queue
const int IA;   // capacity of server arrival queue
const int IR;   // capacity of server output queue

// ------------------------------------------------------------------------
// system model: client/server network is not considered
// (hence network transfer time has to be added to average request time)
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// proxy cache server
module PCS
  pxwaiting: [0..IP] init 0;
  pxaccepted: bool init true;

  // request from arrival queue
  [accept] pxwaiting = IP -> 1 :
    (pxaccepted' = false) ;
  [accept] pxwaiting < IP -> 1 :
    (pxaccepted' = true) &
    (pxwaiting' = pxwaiting+1) ;

  // with probability (1-p), request is forwarded to server
  [sforward] (pxwaiting > 0) & (1-p > 0) -> (1/Ixc)*(1-p) :
    (pxwaiting' = pxwaiting-1) ;

  // with probability p, block is forwarded to client
  [panswer] (pxwaiting > 0) & (p > 0) -> (1/Ixc)*p :
   (pxwaiting' = pxwaiting-1) ;
endmodule

// client queue
module S_C
  icwaiting: [0..IC] init 0;

  // accept answer found on proxy cache server
  [panswer] icwaiting < IC -> 1 :
    (icwaiting' = icwaiting+1) ;

  // accept answer found on web server
  [sanswer] icwaiting < IC -> 1 :
```

```
    (icwaiting' = icwaiting+1) ;

  // request is completed with probability pxc by transfer of block
  [done] (icwaiting > 0) & (pxc > 0) ->  1/(Yxc+Bxc/Rxc)*pxc :
    (icwaiting' = icwaiting-1) ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;

  [sforward] waiting < IA -> 1 :
    (waiting' = waiting+1) ;
  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;

  // request from arrival queue
  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;

  // forwarding of block to internet queue
  [sanswer] (irwaiting > 0) & (q > 0) -> 1/(Ys+Bs/Rs)*q :
    (irwaiting' = irwaiting-1) ;
endmodule

// ------------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------------

rewards "accepted"
  pxaccepted: 1;
endrewards

rewards "pending"
  true : waiting + irwaiting + pxwaiting + icwaiting;
endrewards

rewards "time"
  true : (waiting + irwaiting + pxwaiting + icwaiting)/lambda;
endrewards

rewards "time0"
  true : (waiting + irwaiting + pxwaiting + icwaiting)/lambda
         + (FS/Nc) + (1-p)*(FS/Ns);
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-4, and model constants
//
//   lambda = 10..90, IP = 5, IC = 3, IA = 1, IR = 1
//
// gives reliable results up to lambda = 70
```

```
// estimation of acceptance ratio
R{"accepted"}=? [ S ]

// number of requests pending in system
R{"pending"}=? [ S ]

// time spent in system
R{"time"}=? [ S ]

// time spent in system including network transfer time
R{"time0"}=? [ S ]
```

C PRISM Model of a Proxy Cache Server with External Users

C.1 The Model with No Acceptance Reward

```
// ------------------------------------------------------------------------
// webProxyWithExternalUsers.sm
//
// a bounded queue approximation of the web server proxy model presented in
//
//   Bérczes and Sztrik "Performance Modeling of Proxy Cache Servers"
//
// (c) 2008 Gábor Kusper
// Mathematics and Informatics Institute
// Eszterházy Károly College
// http://www.ektf.hu/
// ------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// ------------------------------------------------------------------------
// system parameters
// ------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate
const int capitallambda;    // visit rate for external users
const double p;             // probability that file is on PCS

// helper constans
const double lambda1 = lambda * p;
const double lambda2 = lambda * (1-p);
const double lambda3 = lambda2 + capitallambda;

// proxy cache server performance
const double alpha = 1;     // alpha = Bxc/Bs
const double beta = 1;      // beta = Rxc/Rs = Yxc/Ys
const double gamma = 1 ;    // gamma = Ixc/Is;

// parameters from paper (units are bytes and seconds)
const double FS = 5000;     // average file size
const double Bs = 2000;     // buffer size
```

```
const double Is = 0.004;    // initialization time
const double Ys = 0.000016; // static server time
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 16000;    // client network bandwidth (128 kbps)

// proxy characteristics
const double Bxc = alpha*Bs; // proxy buffer size
const double Yxc = beta*Ys;  // static proxy time
const double Rxc = beta*Rs;  // dynamic proxy rate
const double Ixc = gamma*Is; // proxy initialization time

// web server and proxa cache server blocks
const double q = func(min, 1, Bs/FS);     // probability of last server block
const double pxc  = func(min, 1, Bxc/FS); // probability of last proxy block

// queue capacities
const int IP;   // capacity of proxy queue
const int IC;   // capacity of client queue
const int IA;   // capacity of server arrival queue
const int IR;   // capacity of server output queue

// ------------------------------------------------------------------------
// system model: client/server network is not considered
// (hence network transfer time has to be added to average request time)
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// proxy cache server
module PCS
  pxwaiting: [0..IP] init 0;

  // request from arrival queue
  [accept] pxwaiting < IP -> 1 :
    (pxwaiting' = pxwaiting+1) ;

  // with probability (1-p), request is forwarded to server
  [sforward] (pxwaiting > 0) & (1-p > 0) -> (1/Ixc)*(1-p) :
    (pxwaiting' = pxwaiting-1) ;

  // with probability p, block is forwarded to client
  [panswer] (pxwaiting > 0) & (p > 0) -> (1/Ixc)*p :
    (pxwaiting' = pxwaiting-1) ;
endmodule

// client queue
module S_C
  icwaiting: [0..IC] init 0;

  // accept answer found on proxy cache server
  [panswer] icwaiting < IC -> 1 :
    (icwaiting' = icwaiting+1) ;

  // accept answer found on web server
  [sanswer] icwaiting < IC -> 1 :
    (icwaiting' = icwaiting+1) ;

  // request is completed with probability pxc by transfer of block
```

```
  [done] (icwaiting > 0) & (pxc > 0) ->  1/(Yxc+Bxc/Rxc)*pxc :
    (icwaiting' = icwaiting-1) ;
endmodule

// generate external requests at rate capitallambda
module external
  [extaccept] true -> capitallambda : true ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;

  // requests from the PCS
  [sforward] waiting < IA -> 1 :
    (waiting' = waiting+1) ;

  // requests from external users
  [extaccept] waiting < IA -> 1 :
    (waiting' = waiting+1) ;

  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;

  // request from arrival queue
  [forward] irwaiting < IR -> 1 :
    (irwaiting' = irwaiting+1) ;

  // forwarding of block to internet queue
  [sanswer] (irwaiting > 0) &
            (q > 0) -> 1/(Ys+Bs/Rs)*q *(lambda2/lambda3):
    (irwaiting' = irwaiting-1) ;

  // forwarding of block to external users, it is not synchronized
  [extanswer] (irwaiting > 0) &
              (q > 0) -> 1/(Ys+Bs/Rs)*q *(1-(lambda2/lambda3)):
    (irwaiting' = irwaiting-1) ;

endmodule

// ------------------------------------------------------------------------
// system rewards
// ------------------------------------------------------------------------

rewards "pending"
  true : waiting + irwaiting + pxwaiting + icwaiting;
endrewards

rewards "time"
  true : (pxwaiting + icwaiting)/lambda +
         (1-p) * (waiting + irwaiting)/lambda3;
endrewards

rewards "time0"
  true : (pxwaiting + icwaiting)/lambda +
         (1-p) * (waiting + irwaiting)/lambda3 +
         (FS/Nc) + (1-p)*(FS/Ns);
```

```
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-4, and model constants
//
//   lambda = 10..90, capitallambda = 100, IP = 7, IC = 3, IA = 19, IR = 8
//
// gives reliable results up to lambda = 70 and capitallambda = 100

// number of requests pending in system
R{"pending"}=? [ S ]

// time spent in system
R{"time"}=? [ S ]

// time spent in system including network transfer time
R{"time0"}=? [ S ]
```

C.2 The Model with Acceptance Rewards

```
// -------------------------------------------------------------------------
// webProxyWithExternalUsers2.sm
//
// a bounded queue approximation of the web server proxy model presented in
//
//   Bérczes and Sztrik "Performance Modeling of Proxy Cache Servers"
//
// Acceptance rewards are added.
//
// (c) 2008 Gábor Kusper
// Mathematics and Informatics Institute
// Eszterházy Károly College
// http://www.ektf.hu/
// -------------------------------------------------------------------------

// continuous time markov chain (ctmc) model
stochastic

// -------------------------------------------------------------------------
// system parameters
// -------------------------------------------------------------------------

// variable parameters
const int lambda;           // network arrival rate
const int capitallambda;    // visit rate for external users
const double p;             // probability that file is on PCS

// helper constans
const double lambda1 = lambda * p;
const double lambda2 = lambda * (1-p);
const double lambda3 = lambda2 + capitallambda;

// proxy cache server performance
const double alpha = 1;     // alpha = Bxc/Bs
const double beta = 1;      // beta = Rxc/Rs = Yxc/Ys
const double gamma = 1 ;    // gamma = Ixc/Is;
```

```
// parameters from paper (units are bytes and seconds)
const double FS = 5000;     // average file size
const double Bs = 2000;     // buffer size
const double Is = 0.004;    // initialization time
const double Ys = 0.000016; // static server time
const double Rs = 1310720;  // dynamic server rate (1.25 Mbytes/s)
const double Ns = 193000;   // server network bandwidth (1544 kbps)
const double Nc = 16000;    // client network bandwidth (128 kbps)

// proxy characteristics
const double Bxc = alpha*Bs; // proxy buffer size
const double Yxc = beta*Ys;  // static proxy time
const double Rxc = beta*Rs;  // dynamic proxy rate
const double Ixc = gamma*Is; // proxy initialization time

// web server and proxa cache server blocks
const double q = func(min, 1, Bs/FS);     // probability of last server block
const double pxc  = func(min, 1, Bxc/FS); // probability of last proxy block

// queue capacities
const int IP;   // capacity of proxy queue
const int IC;   // capacity of client queue
const int IA;   // capacity of server arrival queue
const int IR;   // capacity of server output queue

// ------------------------------------------------------------------------
// system model: client/server network is not considered
// (hence network transfer time has to be added to average request time)
// ------------------------------------------------------------------------

// generate requests at rate lambda
module jobs
  [accept] true -> lambda : true ;
endmodule

// proxy cache server
module PCS
  pxwaiting: [0..IP] init 0;
  pxaccepted: bool init true;

  // request from arrival queue
  [accept] pxwaiting = IP -> 1 :
    (pxaccepted' = false) ;
  [accept] pxwaiting < IP -> 1 :
    (pxaccepted' = true) &
    (pxwaiting' = pxwaiting+1) ;

  // with probability (1-p), request is forwarded to server
  [sforward] (pxwaiting > 0) & (1-p > 0) -> (1/Ixc)*(1-p) :
    (pxwaiting' = pxwaiting-1) ;

  // with probability p, block is forwarded to client
  [panswer] (pxwaiting > 0) & (p > 0) -> (1/Ixc)*p :
    (pxwaiting' = pxwaiting-1) ;
endmodule

// client queue
module S_C
  icwaiting: [0..IC] init 0;
  icaccepted: bool init true;
```

```
  // accept answer found on proxy cache server
  [panswer] icwaiting = IC -> 1 :
    (icaccepted' = false) ;
  [panswer] icwaiting < IC -> 1 :
    (icaccepted' = true) &
    (icwaiting' = icwaiting+1) ;

  // accept answer found on web server
  [sanswer] icwaiting = IC -> 1 :
    (icaccepted' = false) ;
  [sanswer] icwaiting < IC -> 1 :
    (icaccepted' = true) &
    (icwaiting' = icwaiting+1) ;

  // request is completed with probability pxc by transfer of block
  [done] (icwaiting > 0) & (pxc > 0) ->  1/(Yxc+Bxc/Rxc)*pxc :
    (icwaiting' = icwaiting-1) ;
endmodule

// generate external requests at rate capitallambda
module external
  [extaccept] true -> capitallambda : true ;
endmodule

// server arrival queue
module S_I
  waiting: [0..IA] init 0;
  accepted: bool init true;

  // requests from the PCS
  [sforward] waiting = IA -> 1 :
    (accepted' = false) ;
  [sforward] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;

  // requests from external users
  [extaccept] waiting = IA -> 1 :
    (accepted' = false) ;
  [extaccept] waiting < IA -> 1 :
    (accepted' = true) &
    (waiting' = waiting+1) ;

  [forward] waiting > 0 -> (1/Is) :
    (waiting' = waiting-1) ;
endmodule

// server output queue
module S_R
  irwaiting: [0..IR] init 0;
  iraccepted: bool init true;

  // request from arrival queue
  [forward] irwaiting = IR -> 1 :
    (iraccepted' = false) ;
  [forward] irwaiting < IR -> 1 :
    (iraccepted' = true) &
    (irwaiting' = irwaiting+1) ;

  // forwarding of block to internet queue
  [sanswer] (irwaiting > 0) &
            (q > 0) -> 1/(Ys+Bs/Rs)*q *(lambda2/lambda3):
```

```
    (irwaiting' = irwaiting-1) ;

  // forwarding of block to external users, it is not synchronized
  [extanswer] (irwaiting > 0) &
              (q > 0) -> 1/(Ys+Bs/Rs)*q *(1-(lambda2/lambda3)):
    (irwaiting' = irwaiting-1) ;

endmodule

// -------------------------------------------------------------------------
// system rewards
// -------------------------------------------------------------------------

rewards "ipaccepted"
  pxaccepted: 1;
endrewards

rewards "icaccepted"
  icaccepted: 1;
endrewards

rewards "iaaccepted"
  accepted: 1;
endrewards

rewards "iraccepted"
  iraccepted: 1;
endrewards

rewards "pending"
  true : waiting + irwaiting + pxwaiting + icwaiting;
endrewards

rewards "time"
  true : (pxwaiting + icwaiting)/lambda +
 (1-p) * (waiting + irwaiting)/lambda3;
endrewards

rewards "time0"
  true : (pxwaiting + icwaiting)/lambda +
 (1-p) * (waiting + irwaiting)/lambda3 +
         (FS/Nc) + (1-p)*(FS/Ns);
endrewards
```

CSL Queries

```
// run experiments with JOR, termination epsilon 10^-4, and model constants
//
//   lambda = 10..90, capitallambda = 100, IP = 7, IC = 3, IA = 19, IR = 8
//
// gives reliable results up to lambda = 70 and capitallambda = 100
// number of requests pending in system
R{"pending"}=? [ S ]

// time spent in system
R{"time"}=? [ S ]

// time spent in system including network transfer time
R{"time0"}=? [ S ]
```

```
R{"ipaccepted"}=? [ S ]

R{"icaccepted"}=? [ S ]

R{"iaaccepted"}=? [ S ]

R{"iraccepted"}=? [ S ]
```