applied sciences

MDPI

Article

Timed Colored Petri Net-Based Event Generators for Web Systems Simulation

Andrzej Bożek, Tomasz Rak * and Dariusz Rzonca

Department of Computer and Control Engineering, Rzeszow University of Technology, Powstancow Warszawy 12, 35-959 Rzeszow, Poland
* Correspondence: trak@kia.prz.edu.pl

Abstract: Simulation is a powerful process for perfectly planning and dimensioning web systems. However, a successful analysis using a simulation model usually requires variable load intensities. Furthermore, as the client's behavior is subject to frequent changes in modern web systems, such models need to be adapted as well. Based on web systems observation, we come across the need for tools that allow flexible definitions of web systems load profiles. We propose Timed Colored Petri Nets (TCPN) event generators for web environments that could be used to drive simulations for performance evaluation. The article proposes the systematization of the generators for future development. The theoretical part focuses on a classification with a division into deterministic/stochastic and untimed/timed generators. Next, in this study, we investigate representative models of generators from different classes based on the formalism of TCPN. We perform model design and simulation processes using CPN Tools software. Finally, we present a case study involving workload analysis of a web system that processes requests from the designed generator.

Keywords: simulation analysis; performance analysis; workload characterization; Timed Colored Petri Nets; event generator

Citation: Bożek, A.; Rak, T.; Rzonca, D. Timed Colored Petri Net-Based Event Generators for Web Systems Simulation. *Appl. Sci.* **2022**, *12*, 12385. https://doi.org/10.3390/app122312385

Academic Editors: Luis Gomes, João Paulo Barros and Habib Hamam

Received: 6 October 2022
Accepted: 30 November 2022
Published: 3 December 2022

1. Introduction

Modeling next-generation networks requires designers to prepare a traffic generator tailored to the needs of the generated request parameters. To deliver the required load it is necessary to prepare the load generation system. Our approach to implementing such a system is based on the Timed Colored Petri Nets (TCPN) formalism. We propose a configurable TCPN-based generator that could be used for various real system models. Using simulation, we show the proposed request generator models have expected functionality. It is an adaptation to alternative design demands achievable by minimal configuration changes.

Workload modeling problems have been addressed over the past years, resulting in models for generation workloads similar to those observed in the real world. Workload characterization plays a key role in many performance engineering studies. The produced workloads could also be used to drive simulations for system prototyping, testing, and benchmarking of computer systems and networks. Today there are many different areas of workload generator application, such as: cloud computing infrastructures, web systems, networking structures, and video services. In the remaining work we will use a term event generator as a generalization of the mentioned concepts of the request generator and workload generator. The event generator will be considered to be a configurable source of a stream of discrete events, similarly as in [1–4]. Approaches presented in [4–6] parameterize the system model using measurements gathered from real networks. They use workload models that allow the simulation of the network traffic created by real services in various possible scenarios. On the other hand, we can also find works based on an analytical approach [2,3,7,8].

Input stream models generated from navigational patterns cause many problems in software engineering. Workload modeling is also challenging when applied in a highly dynamic environment. The derivation of such a model is non-trivial, which is confirmed by [1–3]. This paper seeks to improve those previous attempts, which are outlined below in Sections 2.1 and 2.2.

Many works are focused on simulating models of different systems using the generated load. We need an input stream, but it could be obtained only from an actual system [5] or produced artificially [9,10] also based on the system log [11,12].

Workload scenarios may have three basic sources: benchmarks, real traces, and workload models. The objective is to get a workload model as close to the real workload as possible. The best option is to start with measured workload data. Downloading such a stream is not an easy task. Therefore, in most cases, an analytical model is used. When preparing the system model, designers do not study the nature of the input stream. It is mostly general and it is not the representation of reality that we expected. Some existing works have proposed workload prediction [13,14] or prediction of user behavior [15], but without adapting it to real needs.

In this context, the main contribution of this paper is a design of a collection of generator models capable of representing the resource demand of the application supported by different user profiles. Different workload classes are known for stressing each resource differently [16]. In addition, the proposed workload generators enable performance analysis and simulation. In the models, we refrain from modeling the physical network and servers. We assume that data transmission delays are negligible and the server model is related with the modeled system, not the requests stream generator.

The remainder of this article is organized as follows. We discuss related work and introduce our previous models of workload generators in Section 2. Section 3 presents our solution based on Petri nets and shows the numerical simulations. In Section 4, we evaluate the usefulness of our event generator model for performance analysis in the web system domain. Finally, Section 5 presents the conclusions and future work.

2. Related Work

Engineers need to have varied realistic workloads for studying the system environments. A workload model is an important way to specify and produce workloads for: cloud systems [7], web systems [11], Big Data systems [17], network systems [18], streaming systems [19], etc. Therefore, a deeper understanding of user behavior, workload properties, and patterns is required. Cloud computing providers expect an understanding of the typical workload patterns of their services. We can find a large number of successful commercial streaming services, such as Netflix, Amazon Prime, or Youtube. Providers stream live videos to a highly variable audience. Network traffic generators play an important role in the design and development of networks and in the security field. Some authors prepared scalable workload generators for testing and benchmarking of high-volume data processing systems [5].

Every month, billions of users access web systems. A large number of users and the huge amount of data processed by these applications make modeling web systems a challenging task. Various models are proposed to capture the behavioral patterns of different user profiles.

Workload generators become crucial tools, as they help system constructors to plan the system design. A survey on workload generators for web systems [20] presents reviewed work on this domain for different types of applications.

For instance, the authors of [21] presented a behavior of a Google Maps client. Based on this characterization, they propose a model of client actions as a simple workload generator. In this context, the authors of [11] proposed reconstructing users' web behaviors from web server logs. It is also possible that the use of simulation models enables the production of system logs based on realistic scenarios [5].

The trend of the computer world is no longer envisaging the operation of one single computer without interacting or cooperating with other computers. These distributed systems have to be designed to meet the new requirements. Creation of them may be facilitated by modeling. Some of the models use variants of Petri nets, and they are applied in a generic context for stream processing of distributed web systems. There are several generators in Petri networks of different classes. One can find publications on web system models in which the query generator [22,23] is described in detail but one can also find publications in which there is a generator in general form, which has not been described in detail due to its simplicity [24,25] (TCPN), [26] (QPN).

CPN (Colored Petri Nets) is a graphical language for modeling, simulating and validating concurrent and distributed systems [27]. It combines the strength of Petri nets (to model synchronization of concurrent processes) with programming languages (to define various data types and manipulate values). Usually, further extensions like TCPN (Timed CPN) or HTCPN (Hierarchical TCPN) are considered. Models developed in CPN (also TCPN or HTCPN) formalism are often implemented in CPN Tools [27]. It is a software tool that facilitates modeling, formal analysis and simulation of the CPN/TCPN/HTCPN models. Furthermore, it can generate a reachability graph, thus some CPN behavioral properties may be verified, for example in [28,29]. The TCPN models presented in this paper have also been developed using CPN Tools.

QPN (Queueing Petri Net) is another formalism based on Queueing Nets and Petri Nets. Queueing Nets are suitable for modeling competition of equipment. To analyze any queue system it is necessary to determine: the arrival process, service distribution, service discipline, and waiting room (scheduling strategies). QPN is a tuple [30]: CPN (a finite and non-empty set of places, a finite and non-empty set of transitions, a color function, the backward and forward incidence functions, an initial marking) and a set of timed and immediate queueing places and a set of timed and immediate transitions. The primary type of QPN place is a queueing place composed of a queue and a depository for tokens that completed their service at a queue. QPN models are typically implemented in QPME (Queueing Petri net Modeling Environment) [31]. It is an open-source tool for stochastic modeling and analysis, consisting of two components: QPE (QPN Editor) and SimQPN (Simulator for QPNs). Furthermore, the QPN models presented in this paper have been developed using QPME.

TCPN formalism is often used for modeling a web system or its part. In [22], several web services interaction models have been proposed. The TCPN models have been prepared, evaluated and simulated in CPN Tools – it is the same formalism and software tool as used in this paper. One of the models presented in [22] is the model of the generator for web service applications. In fact, it generates a stream of discrete events, similar to the generators considered here. The model proposed in [22] uses the formula `()@+expTime(100)` to determine the distribution of application receipts under the exponential law with an intensity of 100 applications per unit of time. Thus, such a generator is an example of a timed stochastic generator, according to the classification introduced in Section 3 of this paper. In [23], the server side of web applications is considered and some models of web and database servers are presented. A separate model of a generator has not been applied there, instead, the server models contain a very simple generator consisting of a single place–transition pair. Such a trivial generator could be considered as an untimed deterministic one, according to Section 3. Petri nets with different extensions have been successfully used to analyze different types of systems, with web applications being one of them [24]. Rak et al. [25] presented a programming tool based on CPN that supports modeling and performance evaluation of system architectures for distributed web system environments. The studies [24,25] have generally used CPN Tools as a Petri net modeling tool with good results. On the superior level of timed stochastic model description [24] they defined the arrival process of the queueing network (two places). The timer place and transition constitute a clock-like structure that produces requests according to random, exponentially distributed frequency. These tokens are accumulated in a form of a timed

multiset in a place and then forwarded into the queueing-based model of the web system. Tokens generated by the arrival process are transferred in sequence by models of web system layers. Each token is equipped with an attached data value, the token color. A similar simple model was prepared using QPN. Client think time is modeled in [26] by the Infinite Server scheduling strategy (queueing place). The number of clients is configured by initial marking in this place. The token represents a client's requests in queueing place and generates a certain number of requests per second.

While there are a lot of articles regarding workload [18,32], little is known about generators with the expected by designers' workload. We address this issue by designing a new requests generator that identifies resources and users so it satisfies most of the system constructors. There are benefits of building specialized web system models with particular groups of users with similar behavior patterns as a workload, as opposed to using a single class for all users. The proposed workload models can be used in the construction of performance models used by many research domains.

2.1. Early CPN Models of Generators

Different formalisms may be used to model a generator of discrete events. Initially, Coloured Petri Nets (CPN) [27] were our first choice, to facilitate integration with existing CPN models of the web systems. Such early models of CPN request generators have been thoroughly analyzed in [3]. The previous results are briefly summarized here to make this paper self-contained.

The basic CPN model of the generator is shown in Figure 1. The places `Res` and `Users` model available resources and users accordingly. When the transition `Generator` fires, one of the users and one of the resources are randomly chosen and the resulting token with appropriate marking is created in the place `Stream`.

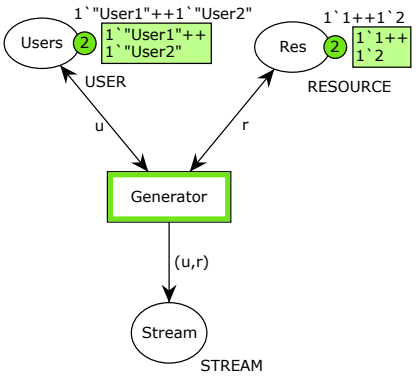
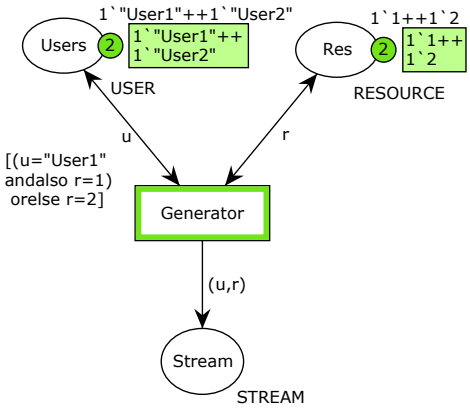


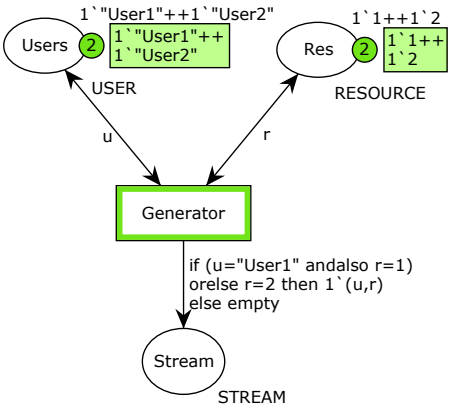
Figure 1. Simple CPN model of the workload generator.

Unfortunately, such a model is so simple that it cannot be realistic. It has been assumed that each user can freely choose each resource, which is a rare case. Let us consider the simple restriction—`User1` may use both resources, whereas `User2` is allowed to only use resource 2. Such a limitation may be imposed in different ways in the model, e.g., in the transition `Generator` guard or in the inscription of the arc connecting `Generator` transition with `Stream` place. The first case is shown in Figure 2a whereas the second one is given in Figure 2b.

Both models restrain `User2` from using resource `1`. However, simulation results for them are completely different and may appear to be a bit counter-intuitive at a first glance. Let us assume that the transition `Generator` has been fired 100,000 times in each model. The exemplary results are shown in Figure 3.

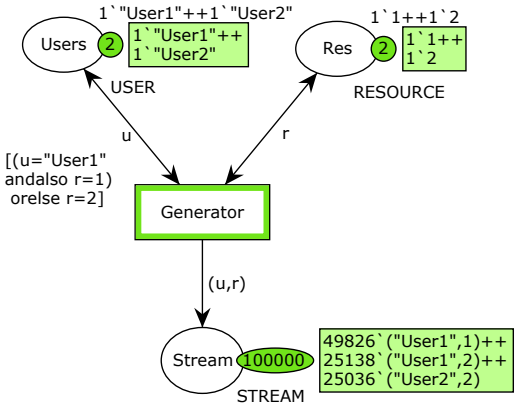


(**a**) Restriction in the transition guard.

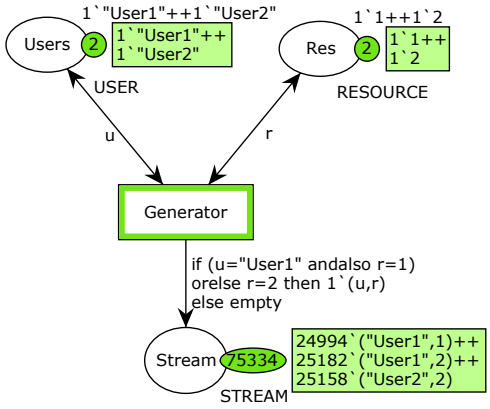


(**b**) Restriction in the inscription of the arc.

Figure 2. Restrictions in CPN models.



(**a**) Restriction in the transition guard.



(**b**) Restriction in the inscription of the arc.

Figure 3. Simulation result for both models.

It could be seen that the distribution of the tokens in the `Stream` place, representing the mapping between users and resources in the generated stream, is different. In the first model, approximately half of the resulting tokens are related to the `User1` and resource `1` case, twenty-five percent of the tokens represent `User1` and resource 2 mapping, and finally, twenty-five percent connect `User2` with resource 2. The total number of tokens in the `Stream` is 100,000 because each firing of the `Generator` transition produces one token.

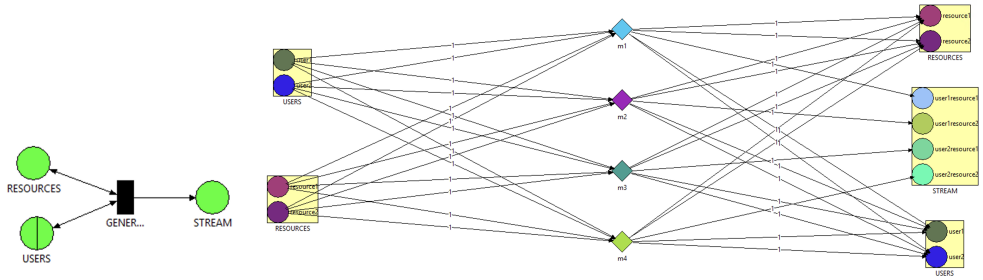
In the second model each mapping (`User1` and resource `1`, `User1`– resource 2, `User2`–resource 2) have approximately equal representation in the resulting stream. However the total number of tokens in the `Stream` place does not sum to 100,000. Transition `Generator` has been fired 100,000 times, but if the restricted binding (`User2` and resource 1) is randomly chosen, the inscription on the arc does not produce any token.

Detailed analysis of the causes of this phenomenon could be found in [3], so only brief conclusions are presented here. Such behavior is related to the exact execution semantics defined by CPN. In the first model, the restriction in the transition guard is applied during the binding of the tokens to the variables, before the transition fires. Thus, in approximately half of the cases, `User1` and resource `1` are chosen, whereas resource 2 (for any user) is chosen in the other half. In the second model, the binding for firing the transition is randomly chosen without any restriction and after the transition fires, the resulting token with restricted results (`User2`, resource `1`) is dropped if necessary. Unfortunately, it means that the modeling of the generators in the CPN formalism is not so trivial as it may seem, even for the basic case with two users and two resources, if precise control on the

distribution of the mappings between users and resources is expected. It is necessary to ensure that the generated stream meets designer requirements. It would be troublesome for a more sophisticated case with numerous users, resources, and complex restrictions. Therefore, we feel that more flexible models should be created to simplify future works. Considering the possible ways of applying the restrictions to CPN models as presented in Figure 2 and their different behavior (Figure 3), which may be counter-intuitive for an inexperienced engineer, one of the main reasons for the method proposed in the following sections is to facilitate the creation of complex models.

2.2. Early QPN Models of Generators

In the next step, based on Queueing Petri Nets (QPN) [33], we modeled a generator of requests. The study results show that appropriately adjusting queueing Petri net models could help produce expected streams of tokens. The general mathematical generator model of QPN was defined in [2]. The second model (Figure 4a) allows us to freely join the user tokens with the resource tokens to obtain any distribution (populations with the expected distribution of the output stream). Transition modes and firing weights (Figure 4b) could be used to model the probability of choosing the appropriate binding of the transition. The main logic lies in the firing of transitions with weights and handling of the token colors.



(**a**) Main model. (**b**) Incidence function.

Figure 4. QPN model.

The generator model is represented by a set of places (`RESOURCES` and `STREAM`), queueing place (`USERS`), and the immediate transition `GENERATOR`. Requests and resources are modeled by tokens of different colors. Place `USERS` generates tokens of requests as `user1` and `user2`. `RESOURCES` generates tokens of resources such as `resource1` and `resource2`. After acquiring a resource from the `Res` place, the requests are placed into the `STREAM` place. The `STREAM` place consists of `user1resource1`, `user1resource2`, `user2resource1` and `user2resource2` tokens. Place `USERS` has an infinite server queue with an initial population of tokens. Firing the `GENERATOR` transition creates a token in the `STREAM` place and assigns one of the resources to one of the users. `STREAM` is an ordinary place that contains the resulting stream of the mean token population. The restriction is modeled by setting the appropriate firing weight.

Detailed analysis of this approach could be found in [2], so only brief conclusions are presented here. The QPME tool [31] generates a report showing the predicted population for the individual model configuration. We can find that the mean token population is appropriately distributed for tokens (`user1resource1`, `user1resource2`, `user2resource1`, `user2resource2`). Some examples can be found in the article [2]. The quantities of the generated tokens in the `STREAM` place for each mode are consistent with the appropriate firing weights. Therefore, various distributions of the tokens in the stream can easily be modeled by appropriate changes in the firing weights.

3. Systematization of the Generator Models

As shown in Section 2, the models of web systems described in the literature usually implement some kind of generator. Sometimes such a generator is separated from the main

model (e.g., in [22]), and sometimes is an integral part of it (e.g., [23–25]). Various generators are considered, from trivial [23] to quite sophisticated [22]. Their time and stochastic parameters differ, therefore, we feel that the further development of the discrete event generators requires systematization. We decided to introduce four classes of generators, namely:

- untimed deterministic,
- untimed stochastic,
- timed deterministic,
- timed stochastic.

A generator is referred to as deterministic if its output is identical in every execution. In particular, the numbers of any type of generated events remain unchanged, as well as the times of event emission (in a timed model) are fixed. The deterministic model can be useful, e.g., to adjust the parameters of some system, provided that it is repeatedly excited by the same input. On the other hand, a stochastic generator produces events that are randomly disturbed in their number or/and emission time. The disturbance is, however, well-defined in the sense of probability distribution. A typical use case of this generator type involves performance analysis of a system in which parameters remain fixed but the input varies within some range. Untimed generators produce a complete set of events at once and the events are not ordered using the notion of time. It does not mean that models with such generators are trivial, as they may exhibit complex behavior dependent on the number of input events and proportions between events of different types. In the case of timed generators, an additional ordering parameter is taken into account that imitates the real-time flow. Note that the deterministic and stochastic variants of the event generators are opposite and mutually exclusive. Similarly, a generator has to be untimed or timed. It implies four combinations of generator sub-types presented in the following sections.

Although the introduced classification of event generators is general and can be applied in various formalisms, we decided to focus on the TCPN formalism to overcome the problems mentioned in Section 2.1. In this section, some examples of TCPN models of generators of each class are presented. For consistency, all models meet the following assumptions:

1. There is a place named `Config`. The initial marking of this place is a list and each element of this list describes one type of event generated by the model.
2. The element describing an event has the form of n-tuple in which the first element is a color representing the event type and the remaining $n-1$ elements specify in detail how this event is to be generated. This specification depends on the generator version.
3. Each generated event is represented by one token of type `EVT` (untimed models) or `EVT_T` (timed models). These tokens are collected in the place named `Events`.

In the examples presented in this section, to keep things clear and simple, event types are represented by an enumeration color set

```
colset EVT = with Evt1 | Evt2 | Evt3;
```

that distinguishes three types and assigns them simple labels.

3.1. Untimed Models

The untimed deterministic variant assumes that given numbers of events of given types are immediately present. In a CPN model, it is represented by a multiset of related event colors in the destination place `Events`. There is a need, therefore, to transform a configuration list to the marking of `Events`. The list has to specify the number of events of any type, so its color set `CFG` is defined as follows

```
colset EVTN = product EVT * INT;
colset CFG = list EVTN;  (*1*)
```

and i-th list element is a pair (e_i, n_i), where e_i is the element color and n_i denotes its multiplicity.

The untimed deterministic model is presented in Figure 5. It is a simple design with only one transition `generate`. An exemplary initial marking (Figure 5a) declares the generation of 30 events of type `Evt1` and `Evt2`, as well as 40 events of type `Evt3`. The transition `generate` fires only once that leads to the expected final marking shown in Figure 5b. To transform the configuration initial marking to the target marking of `Events`, the following SML function is used

```
fun generateEvents([]) = empty
| generateEvents((e,n)::r) = (n‘e) ++ generateEvents(r);
```

which recursively maps the head list element `(e,n)` to the multiset `n‘e`.

(**a**) Initial marking. (**b**) Final marking.

Figure 5. Untimed deterministic CPN-based event generator.

The untimed stochastic model is presented in Figure 6. The configuration color set `CFG` (*1*) is reused again in this model, but its interpretation changes. Now, the i-th element (e_i, f_i) of the configuration list specifies the event color e_i followed by its relative frequency f_i. In the initial marking (Figure 6a) the transition `init` is ready. Its firing moves the configuration token from `Config` to `Cfg2` and additionally calls the function

```
fun fsum([]) = 0
| fsum((e,f)::r) = f+fsum(r)
```

calculating the sum $f_{\text{sum}} = \sum_{i=1}^{N_{\text{tp}}} f_i$, where N_{tp} denotes the number of event colors, i.e., the length of the configuration list. The result is inserted to the place `FSum`. After that, only the transition `generate` remains active, as the marking of `Config` is empty. Each firing of `generate` produces one new event by insertion the related token to the place `Events`. It is controlled by the function

```
fun generateEvent(c,s,[]) = empty
| generateEvent(c,s,(e,f)::r)
= if c<=s andalso c+f>s then 1‘e
else generateEvent(c+f,s,r)
```

In the function call expression (see the inscription of the arc from `generate` to `Events`), an integer from $\mathcal{U}_{\text{d}}(0, f_{\text{sum}}-1)$ is assigned to the variable s. The function recursively inspects intervals

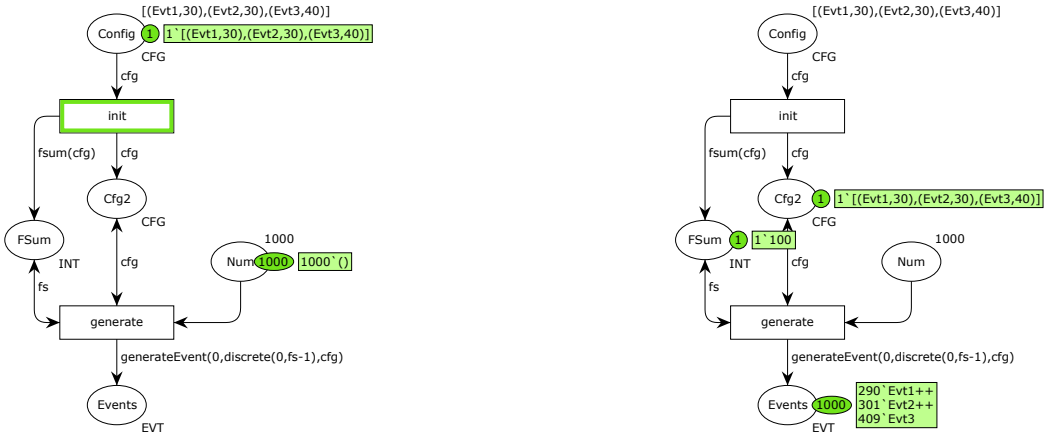
$$I_i = \left[\sum_{k=1}^{i-1} f_k, \sum_{k=1}^{i} f_k\right), \qquad i = 1, 2, \ldots, N_{\text{tp}},$$

and returns the event e_i if $s \in I_i$. Therefore, f_i has indeed the meaning of the relative frequency of e_i in the stochastic process. In other words, the probability of e_i has the value

$$\pi_i = \frac{f_i}{\sum_{i=1}^{N_{\text{tp}}} f_i}.$$

Note that if $\sum_{i=1}^{N_{\text{tp}}} f_i = 100$, then f_i directly expresses the probability in percents.

Let N_{evt} be the initial marking of the place `Num`. Each firing of `generate` consumes one token from `Num`, so the considered model generates N_{evt} events. In particular, it is 1000 events in the presented example (Figure 6b).



(**a**) Initial marking. (**b**) Final marking.

Figure 6. Untimed stochastic CPN-based event generator.

Assuming an ideal stochastic process, the expected number of events of type e_i equals $\pi_i N_{\text{evt}}$. The actual number of such events η_i is a random variable of binominal distribution $\mathcal{B}(N_{\text{evt}}, \pi_i)$. It is well known that if $\pi_i N_{\text{evt}}$ and $(1-\pi_i)N_{\text{evt}}$ are large enough, in practice greater than 5, this distribution is well approximated by the normal distribution [34]

$$\mathcal{N}_\eta\left(\mu_i = \pi_i N_{\text{evt}},\ \sigma_i = \sqrt{\pi_i N_{\text{evt}}(1-\pi_i)}\right). \tag{1}$$

Let us evaluate the model against the expected distribution. First, for convenience, we define the relative deviation of the i-th type events number from its expected value

$$\delta_i = \frac{\eta_i}{\pi_i N_{\text{evt}}} - 1. \tag{2}$$

The mapping from η_i to δ_i given by (2) transforms the distribution $\mathcal{N}_\eta$ (1) to

$$\mathcal{N}_\delta\left(\mu_i^* = 0,\ \sigma_i^* = \sqrt{\frac{1-\pi_i}{\pi_i N_{\text{evt}}}}\right). \tag{3}$$

Two series of 1000 simulation experiments have been performed for $N_{\text{evt}} = 1000$ and $N_{\text{evt}} = 10{,}000$. Their results in the form of δ_i distributions are presented in Figures 7 and 8, respectively. The red lines show the expected limit normal distribution $\mathcal{N}_\delta$ (3). One can observe good compatibility between the experimental and predicted results, in particular, the dispersion of δ_i decreases with the increase of N_{evt}.

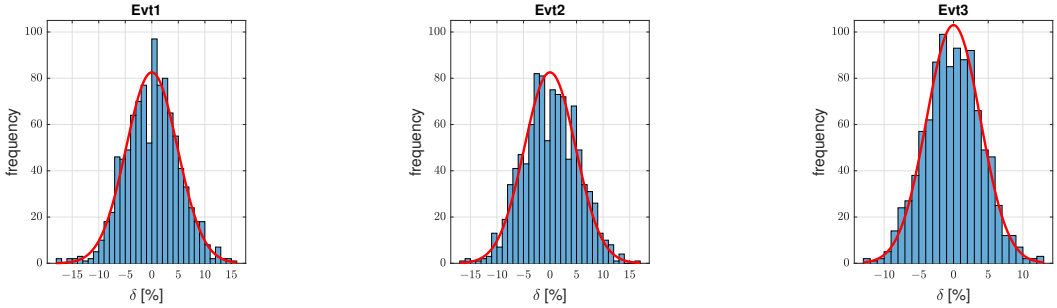


Figure 7. Distribution of δ_i for $N_{\text{evt}} = 1000$.

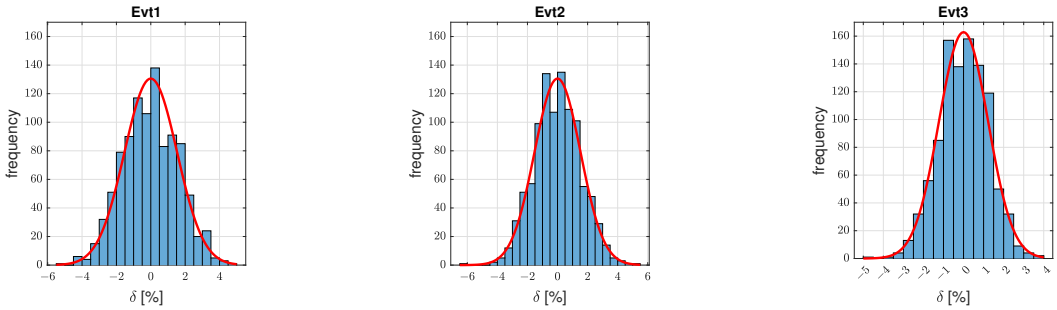


Figure 8. Distribution of δ_i for $N_{\text{evt}} = 10{,}000$.

3.2. Timed Models

Events generated in timed simulation could be defined in many ways. A basic idea is to describe events of the i-th type by the data structure

$$\left(e_i, [t_{i,1}, t_{i,2}, \ldots, t_{i,n_i}]\right),$$

where e_i is the event color and the values $t_{i,j}$, $j = 1, 2, \ldots, n_i$ directly specify the times at which the event has to be generated. Such an approach could be based on a simple CPN model, similar to the one presented in Figure 5, so the solution is not going to be presented.

The scenario considered here shows that generator configuration determines the frequency of events on given time intervals, while the individual events are generated automatically. It is a more practical variant, useful in many simulations, e.g., for system performance analysis, but it is also more complex for implementation.

The i-th element of the configuration list has the form

$$\left(e_i, [(d_{i,1}, p_{i,1}), (d_{i,2}, p_{i,2}), \ldots, (d_{i,j}, p_{i,j}), \ldots, (d_{i,m_i}, p_{i,m_i})]\right), \tag{4}$$

where $d_{i,j}$ specifies the duration of the interval during which events are generated with the period $p_{i,j}$. When the time of the j-th interval expires, the generation switches to the $(j+1)$-th one, and so on, with the exception that $j = m_i$ is followed by $j = 1$. In this way, the generation process is a cycle. More precisely, the events of type e_i are issued at the times

$$t_i \in \mathbb{T}_i = \bigcup_{g=0}^{\infty} \bigcup_{h=0}^{\left\lfloor \frac{-1+d_{i,(g \bmod m_i)+1}}{p_{i,(g \bmod m_i)+1}} \right\rfloor} \left\{ hp_{i,(g \bmod m_i)+1} + \sum_{k=1}^{m_i} \left\lfloor \frac{m_i + g - k}{m_i} \right\rfloor d_{i,k} \right\}, \tag{5}$$

where $g+1$ indicates the generated interval number and $h+1$ is the event number in the interval.

To implement the generator in the form of a CPN model, the following additional color set definitions are needed

```
colset INT2 = product INT * INT;                    (*2*)
```

```
colset INT2_L = list INT2;                                  (*3*)
colset ECFG = product EVT * INT2_L;                         (*4*)
colset CFG = list ECFG;                                     (*5*)
colset EVT_T = EVT timed;                                   (*6*)
colset INTV = product EVT * INT * INT * INT * INT;          (*7*)
colset INTV_T = INTV timed;                                 (*8*)
```

The declarations `(*2*)`-`(*5*)` construct the color set compatible with (4). The color set `(*6*)` is a timed version of the event type. The declarations `(*7*)`-`(*8*)` introduce untimed and timed variants of auxiliary 5-tuple describing a single interval of generated events.

The TCPN model of the generator is presented in Figure 9. Most of the arc inscriptions are 5-tuple adjacent to the color set `INTV`. They are interpreted as

$$\left(e_i, j, n_{i,j}, d_{i,j}, p_{i,j}\right), \tag{6}$$

where the meaning of e_i, j, $d_{i,j}$, $p_{i,j}$ is consistent with (4), and $n_{i,j}$ indicates the number of the next interval in the repetitive cycle, namely

$$n_{i,j} = \begin{cases} j+1, & j < m_i \\ 1, & j = m_i \end{cases}. \tag{7}$$

Each firing of the transition `init1` consumes one configuration element from `Config` and splits it into separate interval representations adjacent to (6) using the function

```
fun splitCfg(j,(evt,[(dur,per)]))
= 1‘(evt,j,1,dur,per)
| splitCfg(j,(evt,(dur,per)::r))
= 1‘(evt,j,j+1,dur,per)++splitCfg(j+1,(evt,r))
```

Note that this function derives the values of $n_{i,j}$ (the third tuple element) according to (7). Firings of the transition `init2` move the tokens describing the intervals with $j = 1$ to `Intvs` and the remaining tokens from `StartIntvs` are moved to `ActiveIntv` as a result of the transition `init3` firings. All the transitions `init1/2/3` have all their input arcs from the places of untimed types, so they execute at the global time 0 and initialize the generation mechanism.

After the initialization, the generation of the events comes down to firing the transition `generate` over and over again. For example, consider the configuration imposed by the initial marking of `Config` in Figure 9, specifically, the part related to `Evt1`. As a result of the initialization, the token `(Evt1,1,2,3600,30)` is in `ActiveIntv` and the tokens `(Evt1,2,3,3600,60)`, `(Evt1,3,1,3600,10)` are in `Intvs` at time 0. At this simulation time, the transition `generate` is executed with `evt` bind to `Evt1` on all its input/output arcs. According to inscriptions of these arcs, the token `(Evt1,1,2,3600,30)` is effectively replaced by `(Evt1,2,3,3600,60)` in `ActiveIntv`. Simultaneously, the time stamp of the token inserted to `ActiveIntv` is delayed by the value `durA` of the removed token, i.e., $d_{1,1}$ from (4) and (6). Therefore, the next token exchange related to `Evt1` will happen at the simulated time $d_{1,1}$, and this delay represents the duration of the first interval. Then, it repeats for `(Evt1,2,3,3600,60)`, `(Evt1,3,1,3600,10)`, again for `(Evt1,1,2,3600,30)`, and so on. The right order is asserted by binding the colors representing j and $n_{i,j}$ to the arc expressions. Of course, each event type is subject to this procedure independently.

Note that the transition `generate` is fired once at the beginning of each event generation interval, but many events have to be issued during the interval. The individual events within intervals are prepared by the following function building the tokens inserted to the place `Events`

```
fun generateEvent(t,dur,per,evt) = if t<dur then (1‘evt@+t)
+++generateEvent(t+per,dur,per,evt) else empty
```

It recursively constructs a sequence of tokens representing the events delayed by $0, p_{i,j}, 2p_{i,j}, \ldots$, from starting time of the interval up to its completion. It is equivalent to iteration over h subject to fixed g in (5).

The set $\mathbb{T}_i$ in (5) is infinite, but an actual simulation generates a finite number of events. In particular, if the simulation finishes at the global clock value T, there is asserted that all the e_i events with time stamps $t_i \in \mathbb{T}_i$ such that $t_i < T$ have been generated.

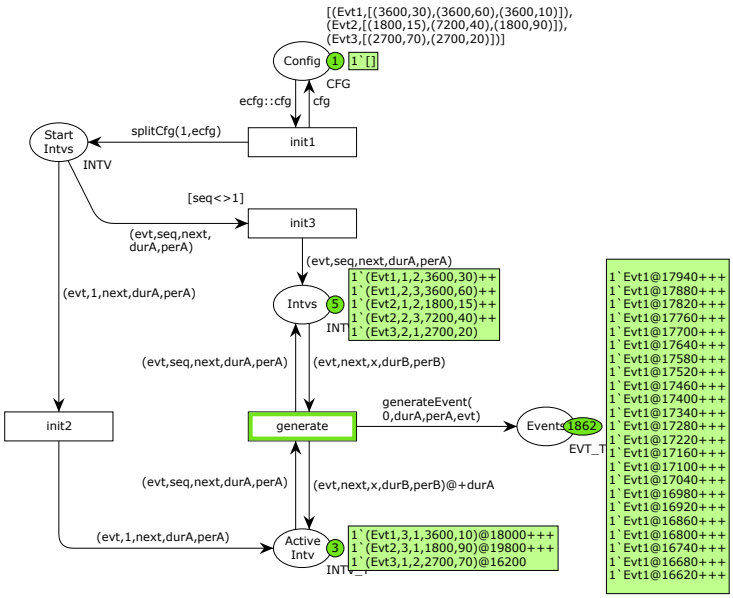
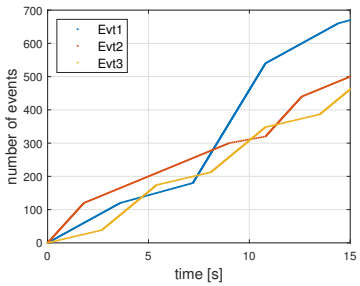
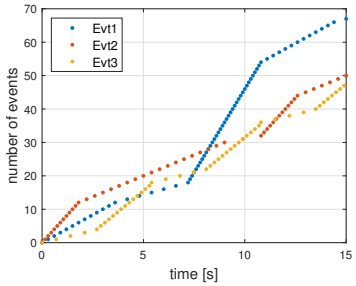


Figure 9. Timed deterministic CPN-based event generator.

For the generator configuration from Figure 9, the resulting cumulated number of generated events as a function of simulated time is shown in Figure 10a. It is assumed that the simulation time unit corresponds to one millisecond, so the plot covers the simulated time interval [0, 15,000]. As the periods $d_{i,j}$ are tiny compared to the simulation time-span, the events are dense so that the related points form a solid line in the plot. However, if one increases the periods 10 times, the events become distinguishable, as shown in Figure 10b.



(**a**) Original configuration.



(**b**) Increased periods.

Figure 10. Number of events as a function of time obtained from the timed deterministic model.

For uniformity, we propose the stochastic variant of the timed generator very similar to the deterministic one. The difference is that periods of event generations are no longer fixed, but are defined as random variables. Namely, the event type description (6) obtains the form

$$\left(e_i, j, n_{i,j}, d_{i,j}, p^{\mathrm{L}}_{i,j}, p^{\mathrm{U}}_{i,j}\right),$$

where $p_{i,j}$ is replaced by the pair $p^{\mathrm{L}}_{i,j}$, $p^{\mathrm{U}}_{i,j}$. The redefined period is now interpreted as the random variable

$$\mathcal{U}_{\mathrm{d}}\left(p^{\mathrm{L}}_{i,j}, p^{\mathrm{U}}_{i,j}\right). \tag{8}$$

The model of the timed stochastic generator is shown in Figure 11. Compared to the design from Figure 9, some color sets have been extended to include one new parameter and some functions have been slightly modified. There are two relevant changes.

First, the function `generateEvent` is redefined in the form

```
fun generateEvent(t,dur,perL,perU,evt) = if t<dur then (1‘evt@+t)
+++generateEvent(t+discrete(perL,perU),dur,perL,perU,evt) else empty
```

Therefore, the time differences between consecutive events of the same interval are randomly generated using the discrete uniform distribution function `discrete(perL,perU)`, being an implementation of (8).

Second, whole interval times are also randomly disturbed. The disturbing component is given in the third line of the inscription on the arc from `generate` to `ActiveIntv` (Figure 11). It is the random distribution

$$\mathcal{U}_{\mathrm{d}}\left(\left\lfloor\frac{p^{\mathrm{L}}_{i,j}-p^{\mathrm{U}}_{i,j}}{2}\right\rfloor, \left\lfloor\frac{p^{\mathrm{U}}_{i,j}-p^{\mathrm{L}}_{i,j}}{2}\right\rfloor\right),$$

equivalent to (8) with an accuracy of integer values, but with the expected value changed to 0, so that the expected value of the interval time remains $d_{i,j}$.

The stochastic character of the model asserts different outcomes when the simulation is repeated multiple times. Indeed, the effect of 100 repetitions is shown in Figure 12, and stochastic fluctuations result in a bunch of dispersed values (green). The outcome of one exemplary simulation is distinguished (red). The reference deterministic profile (blue) is also shown, having the ratios $d_{i,j}/p_{i,j}$ equal to mean values from the configuration given in Figure 12. As expected, the stochastic bunch disperses around this profile.

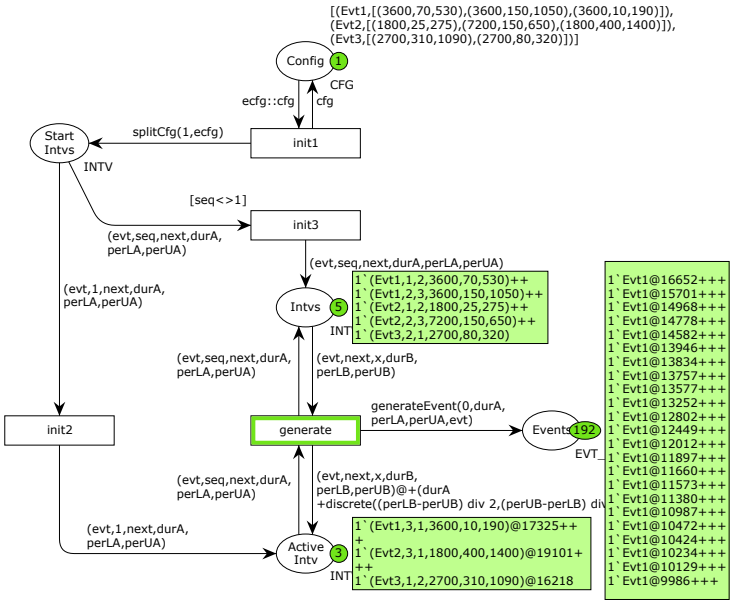


Figure 11. Timed stochastic CPN-based event generator.

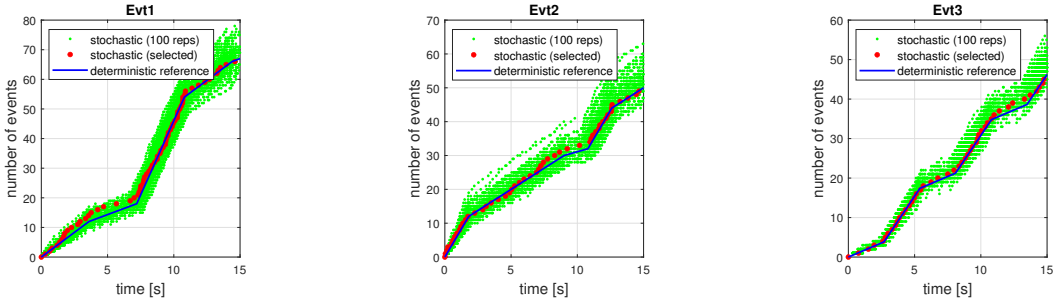


Figure 12. Number of events as a function of time obtained from the timed stochastic model.

4. Case Study

We provide a simple example to demonstrate the usage of the proposed solution. It combines the most advanced timed stochastic generator model and a simplified model of a web server. The HTCPN structure of the system is presented in Figure 13.

The model reuses the generator structure presented in Figure 11 along with all the initial markings, resulting in the sequence of generated events shown in Figure 12. The generator is now enclosed inside the substitution transition `EventGenerator` to better display the new web server structure.

In the part consisting of the transition `mapToRequest` and the place `Requests`, the generated events are first mapped from their generic representation `Evt1`, `Evt2`, `Evt3` to request form `(Usr1,Res1)`, `(Usr1,Res2)`, `(Usr2,Res2)`, respectively. Requests are pairs that indicate a user and a resource needed for it.

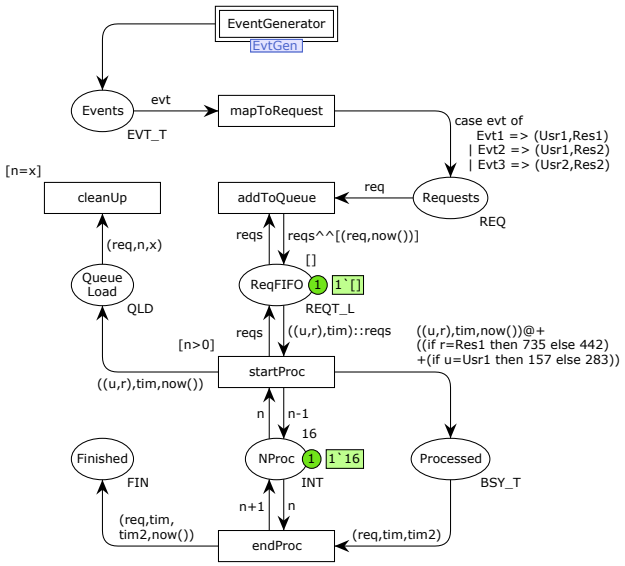


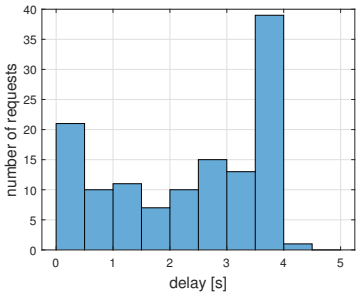
Figure 13. Simulation HTCPN-based model of web server system (initial marking).

Any execution of the transition `addToQueue` puts a triple including the user, the resource, and the entry time `now()` of a new request into the FIFO queue modeled by the place `ReqFIFO`. It is a model of the buffer collecting the requests waiting for processing by the server. The FIFO regime is asserted by the list in `ReqFIFO`, the input elements are added to its end, while the exiting ones are removed from its beginning by the transition `startProc`. The latter transition executes whenever the FIFO list is not empty and the guard expression `n > 0` is satisfied, and its execution models start request processing. In turn, execution of the transition `endProc` models the processing completion. Between the start and completion, the token representing the processed request is held in the place `Processed` with a time delay being the processing time depending on the related user and resource. The delay is determined by the expression of the arc from `startProc` to

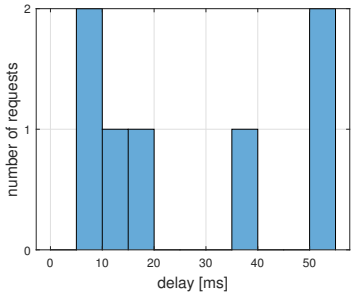
`Processed`. The value of the token in `NProc` represents the number of free processors and it is used to block the processing of new requests while all processors are busy.

Remaining elements: `QueueLoad`, `cleanUp`, `Finished` register data from the model for further analysis. In particular, the final marking of `QueueLoad` represents FIFO entry and exit times of the requests delayed by the queue, while the final marking of `Finished` contains the entry, starting, and completion times of all processed requests.

Simulation experiments have been performed for the configurations with 8 and 16 processors. One can observe on histograms in Figure 14 that there are many delayed requests while using 8 processors and the delays reach up to 4 s (Figure 14a). On the other hand, there are only a few insignificant delays in the case of 16 processors (Figure 14b). It immediately reveals that the requests cannot be effectively processed in the first configuration, while the second one is sufficient. This observation is consistent with other characteristics plotted below.



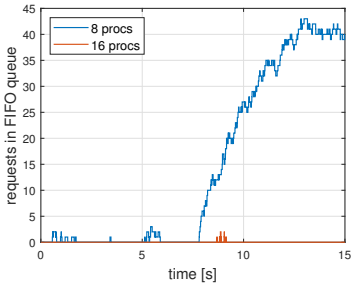
(**a**) 8 processors available.



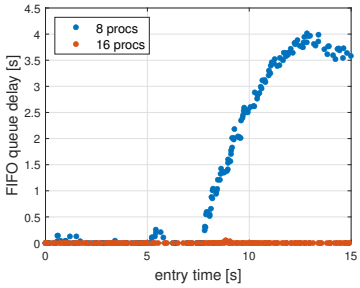
(**b**) 16 processors available.

Figure 14. Histograms of the input FIFO queue delays.

The number of requests waiting in the input FIFO queue as a function of time is shown in Figure 15a. There are almost no waiting requests for 16 processors, while their number starts to increase since about 8 s for the server with 8 processors, up to approximately 40 requests. Note that the FIFO queue delay time as a function of the request entry time (Figure 15b) is proportional to the number of waiting requests, and its maximum value is about 4 s, in accordance to the histogram in Figure 14a.



(**a**) Number of requests in the queue.



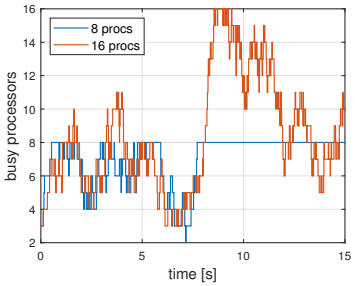
(**b**) Request delay as a function of entry time.

Figure 15. Input FIFO queue analysis.

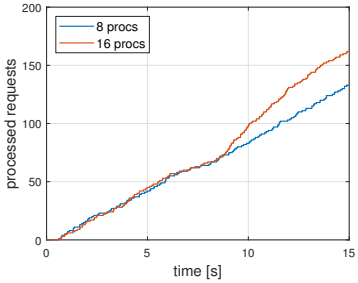
The data useful for processor load analysis is given in Figure 16. The 8-processor server is fully loaded since 8 s and it does not process requests on time. There is no such problem in the second case, where the number of processors turns out just right and all processors work together only at very short time intervals (Figure 16a). As long as the servers are not overloaded they process requests with similar efficiency, but then the 16-processor server works relatively faster (Figure 16b). It is easy to note that the 8-processor server overload observed in all Figures 15 and 16 is caused by the increased speed of request generation since about 8 s, visible in Figure 12 (`Evt1` and `Evt3`). Therefore, appropriate parametrization

of the applied generator, gradually increases the speed of generating requests, which may facilitate the detection of potential bottlenecks in the system model.

The described example indicates that event generators based on TCPNs are very useful components for web systems simulation and analysis. Here, connecting one of the proposed generators with a simple web server model, we have performed quite comprehensive load tests.



(**a**) Number of busy processors.



(**b**) Number of processed requests.

Figure 16. Processors load analysis.

5. Summary

The classification of discrete event generators has been introduced in the paper. Exemplary TCPN models of such generators have been described and analyzed. They can be used to facilitate the generation of input streams for TCPN models of distributed systems, e.g., web systems. The proposed approach is flexible and can be tailored to suit developer needs, to produce events according to requested constraints and predefined distribution. Thus, performance evaluation of system models under various workloads is possible even at the early stages of system development.

We developed an approach that helps us to prepare a stream of requests. The study demonstrates the modeling advantages and shows how the discussed generator models could be used in a simple web system. Next, we shall consider analyzing the web system response time characteristics for classes of clients to effectively model Internet requests. Future research will focus on the following: evaluation and optimization of the various model parameters; the impact of different requests to model results; and the quantitative study of the impact of the events generator model proposed in this paper on the performance of modeled web system.

Author Contributions: Conceptualization, T.R. and D.R.; formal analysis, software, A.B.; validation D.R.; writing—review and editing, A.B., T.R. and D.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The author declare no conflict of interest.

References

1. Seshadri, K.; Pavana, C.; Sindhu, K.; Kollengode, C. *Unsupervised Modeling of Workloads as an Enabler for Supervised Ensemble-based Prediction of Resource Demands on a Cloud*; Verma, P., Charan, C., Fernando, X., Ganesan, S., Eds.; Advances in Data Computing, Communication and Security; Springer: Singapore, 2022; pp. 109–120.
2. Rak, T.; Rzonca, D. Recommendations for Using QPN Formalism for Preparation of Incoming Request Stream Generator in Modeled System. *Appl. Sci.* **2021**, *11*, 11532. [CrossRef]
3. Rzonca, D.; Rzasa, W.; Samolej, S. *Consequences of the Form of Restrictions in Coloured Petri Net Models for Behaviour of Arrival Stream Generator Used in Performance Evaluation*; Gaj, P., Sawicki, M., Suchacka, G., Kwiecień, A., Eds.; Computer Networks; Springer International Publishing: Cham, Switzerland, 2018; pp. 300–310. [CrossRef]
4. Abad, C.L.; Yuan, M.; Cai, C.X.; Lu, Y.; Roberts, N.; Campbell, R.H. Generating request streams on Big Data using clustered renewal processes. *Perform. Eval.* **2013**, *70*, 704–719. [CrossRef]

5. Rak, T.; Żyła, R. Using Data Mining Techniques for Detecting Dependencies in the Outcoming Data of a Web-Based System. *Appl. Sci.* **2022**, *12*, 6115. [CrossRef]
6. Gonçalves, G.D.; Drago, I.; Vieira, A.B.; Couto da Silva, A.P.; Almeida, J.M.; Mellia, M. Workload models and performance evaluation of cloud storage services. *Comput. Netw.* **2016**, *109*, 183–199. . [CrossRef]
7. St-Onge, C.; Benmakrelouf, S.; Kara, N.; Tout, H.; Edstrom, C.; Rabipour, R. Generic SDE and GA-Based Workload Modeling for Cloud Systems. *J. Cloud Comput.* **2021**, *10*, 6. [CrossRef] [PubMed]
8. Rak, T. Modeling Web Client and System Behavior. *Information* **2020**, *11*, 337. [CrossRef]
9. An, C.; Zhou, J.t.; Mou, Z. *A Generic Arrival Process Model for Generating Hybrid Cloud Workload*; Sun, Y., Lu, T., Xie, X., Gao, L., Fan, H., Eds.; Computer Supported Cooperative Work and Social Computing; Springer: Singapore, 2019; pp. 100–114.
10. Sun, J.; Zhao, H.; Mu, S.; Li, Z. Purchasing Behavior Analysis Based on Customer's Data Portrait Model. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; Volume 1, pp. 352–357. [CrossRef]
11. Liu, S.; Wang, J.; Wang, H.; Wang, H.; Liu, Y. WRT: Constructing Users' Web Request Trees from HTTP Header Logs. In Proceedings of the ICC 2019—2019 IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019; pp. 1–7. [CrossRef]
12. Magalhães, D.; Calheiros, R.N.; Buyya, R.; Gomes, D.G. Workload Modeling for Resource Usage Analysis and Simulation in Cloud Computing. *Comput. Electr. Eng.* **2015**, *47*, 69–81. [CrossRef]
13. Daradkeh, T.; Agarwal, A.; Zaman, M.; S, R.M. Analytical Modeling and Prediction of Cloud Workload. In Proceedings of the 2021 IEEE International Conference on Communications Workshops (ICC Workshops), Montreal, QC, Canada, 14–23 June 2021; pp. 1–6. [CrossRef]
14. An, C.; Zhou, J.t. Resource Demand Forecasting Approach Based on Generic Cloud Workload Model. In Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Guangzhou, China, 8–12 October 2018; pp. 554–563. [CrossRef]
15. Rizothanasis, G.; Carlsson, N.; Mahanti, A. Identifying User Actions from HTTP(S) Traffic. In Proceedings of the 2016 IEEE 41st Conference on Local Computer Networks (LCN), Dubai, United Arab Emirates, 7–10 November 2016; pp. 555–558. [CrossRef]
16. Grohmann, J.; Eismann, S.; Bauer, A.; Spinner, S.; Blum, J.; Herbst, N.; Kounev, S. SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation. *ACM Trans. Auton. Adapt. Syst.* **2021**, *15*, 1–31. [CrossRef]
17. Ajwani, D.; Ali, S.; Katrinis, K.; Li, C.H.; Park, A.J.; Morrison, J.P.; Schenfeld, E. A Flexible Workload Generator for Simulating Stream Computing Systems. In Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, Singapore, 25–27 July 2011; pp. 409–417. [CrossRef]
18. Bikmukhamedov, R.F.; Nadeev, A.F. Multi-Class Network Traffic Generators and Classifiers Based on Neural Networks. In Proceedings of the 2021 Systems of Signals Generating and Processing in the Field of on Board Communications, Moscow, Russia, 16–18 March 2021; pp. 1–7. [CrossRef]
19. Guarnieri, T.; Drago, I.; Cunha, Í.; Almeida, B.; Almeida, J.M.; Vieira, A.B. Modeling large-scale live video streaming client behavior. *Multimed. Syst.* **2021**, *27*, 1101–1124. [CrossRef]
20. Curiel, M.; Pont, A. Workload Generators for Web-Based Systems: Characteristics, Current Status, and Challenges. *IEEE Commun. Surv. Tutorials* **2018**, *20*, 1526–1546. [CrossRef]
21. Braga, V.G.; Correa, S.L.; Cardoso, K.V.; Viana, A.C. Data-Driven Characterization and Modeling of Web Map System Workload. *IEEE Access* **2021**, *9*, 26983–27002. [CrossRef]
22. Gozhyj, A.; Kalinina, I.; Gozhyj, V.; Vysotska, V. Web Service Interaction Modeling with Colored Petri Nets. In Proceedings of the 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Metz, France, 18–21 September 2019; Volume 1, pp. 319–323. [CrossRef]
23. Gaur, N.; Joshi, P.; Jain, V.; Srivastava, R. Coloured Petri Nets Model for Web Architectures of Web and Database Servers. *Int. J. Comput. Inf. Eng.* **2015**, *9*, 2066–2075.
24. Rak, T.; Samolej, S. Distributed Internet Systems Modeling Using TCPNs. In Proceedings of the International Multiconference on Computer Science and Information Technology, Wisla, Poland, 20–22 October 2008; pp. 515–522. [CrossRef]
25. Samolej, S.; Rak, T. Simulation and Performance Analysis of Distributed Internet Systems Using TCPNs. *Inform.-J. Comput. Inform.* **2009**, *33*, 405–415.
26. Rak, T. Response Time Analysis of Distributed Web Systems Using QPNs. *Math. Probl. Eng.* **2015**, *2015*, 490835. [CrossRef]
27. Jensen, K.; Kristensen, L.M. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*; Springer: Berlin/Heidelberg, Germany, 2009.
28. Rezig, S.; Achour, Z.; Rezg, N.; Kammoun, M.A. Supervisory control based on minimal cuts and Petri net sub-controllers coordination. *Int. J. Syst. Sci.* **2016**, *47*, 3425–3435. [CrossRef]
29. Rezig, S.; Rezg, N.; Hajej, Z. Online Activation and Deactivation of a Petri Net Supervisor. *Symmetry* **2021**, *13*, 2218. [CrossRef]
30. Bause, F. Queueing Petri Nets-A formalism for the combined qualitative and quantitative analysis of systems. In Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, 19–22 October 1993; pp. 14–23. [CrossRef]

31. Kounev, S.; Lange, K.D.; von Kistowski, J. *Systems Benchmarking: For Scientists and Engineers*; Springer: Berlin/Heidelberg, Germany, 2020. [CrossRef]
32. Patil, A.G.; Surve, A.R.; Gupta, A.K.; Sharma, A.; Anmulwar, S. Survey of synthetic traffic generators. In Proceedings of the 2016 International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 26–27 August 2016; Volume 1, pp. 1–3. [CrossRef]
33. Rak, T. Performance Analysis of Distributed Internet System Models using QPN Simulation. In Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS), Warsaw, Poland, 7–10 September 2014; Volume 2, pp. 769–774.
34. Neter, J.; Wasserman, W.; Whitmore, G. *Applied Statistics*; Allyn & Bacon: Boston, MA, USA, 1992.