Scientific Journal of Riga Technical University
Computer Science. Information Technology and Management Science
2011
Volume 49

Use of the Deferred Approach in Scientific Applications

Pavel Osipov[1], Arkady Borisov[2], [1-2]*Riga Technical University*

***Abstract* – In this paper, the implementation of security system that has strict requirements on the time of evaluation of each transaction made by the user is examined on the example of building a system for user behaviour modelling using Markov models. Evaluation of the effectiveness of both the classical approach to the implementation of a server that calculates metric of the user model and with the use of lightweight threads, as well as of a new ideology - Deferred-based event model is performed.**

A number of tests of various configurations are conducted, showing the preferred server for the Deferred-based type of system as well as an approach to implementing this type of request service.

***Keywords* – Deferred, Python, server, highload, response time**

I. Introduction

Nowadays, there are many different algorithms that require large hardware resources for their operations. This is due both to an increase in complexity of the algorithms and the volume and structure of the data being processed. The usual practice in such cases is to increase the number and capacity used for data processing machines, in particular, the transition to a cluster or cloud. But not always this kind of approach is economically justified and there is a question of performing such tasks on the available hardware, but using different optimization techniques.

In [1] the implementation and evaluation of the algorithm design and analysis of user behaviour model of the electronic system to detect its abnormal activity is described. It is shown that the proposed algorithm can be used in systems operating with sensitive data. However, the established test system implemented in the programming language PHP showed a very low rate of model treatment. This is due to poor optimization of the PHP language with regard to performing complex mathematical calculations.

To implement the final system, the Python programming language was selected as a well-established and highly-loaded language in the creation of systems and implementation of a variety of scientific problems [2].

When using the Python language in this area, there is a requirement for the effective organization of the server processing the request to review and update models as the server itself is also created in Python. In the second part of this paper a technique to select the best server and the results of its application are presented.

There are very strict requirements for speed of processing each transaction: the upper threshold of processing time for 100 models is selected to be 500 milliseconds. There is also a requirement to use typical server hardware, without having to purchase expensive hardware. Depending on the processing power of the equipment rate of shortchanging the same model can vary. At present, an algorithm for computing the metrics and model updates is implemented in Python, it uses about 50 milliseconds of CPU time on a computer Intel Core i3 560 + 4Gb RAM DDR3 running on Ubuntu operating system. However, in addition it is necessary to consider the time spent on initializing Python interpreter, as well as temporary costs required to load the model from the database, to receive parameters and issue the result. Finally, the time may reach more than 200 milliseconds on a script that cannot consistently handle 90 models for 500 ms.

Therefore, the need for optimization of the treatment process arises, through the use of parallel computing, the use of all possible cores, pooling database connections, using effective models of long-term storage, as well as efficient allocation of resources to handle a large number of simultaneous requests.

Due to the above-mentioned limitations, it was decided to use the experience and some software tools used to create electronic systems aimed at heavy loads.

II. Target System Structure

The developed module is part of a big information system security module, which imposes certain restrictions on the possible architectural solutions for its implementation.

The target system is a set of distributed services, and security is one of its modules. In turn, the system modelling user behaviour and the evaluation of each action on the basis of this model is a module of a common security system.

The general logic of the workflow lies in the fact that at any given time many users perform a variety of activities - transactions, each of which, before being processed by the system must be approved by the security module. During processing, each transaction security module in particular examines how different a query is from the typical behaviour for this user. For this purpose, transaction metric is calculated [1]. To calculate it, the user must have a personal behaviour model. When entering the system for the first time, a user is assigned a typical model of the corresponding type of user class. In the course of work, personal model is varied according to some rules, thus adjusting to the changing behaviour of the individual.

A more detailed block-schema of this process is shown in Fig**. 1**.

A. Python Servers Implementation Approaches

Almost all of the servers implemented in Python, use one of two classical approaches to the implementation logic that are described below.

Creating a system process to handle each request

The so-called "Heavy servers", when requested, create a system process that handles the request independently of the server itself. This allows the server to effectively handle many simultaneous requests. However, the establishment of a systemic process is fairly resource-expensive operation and often the system has a limit on the number of processes simultaneously available for the application. Their number is seldom greater than 100, even on powerful modern servers.

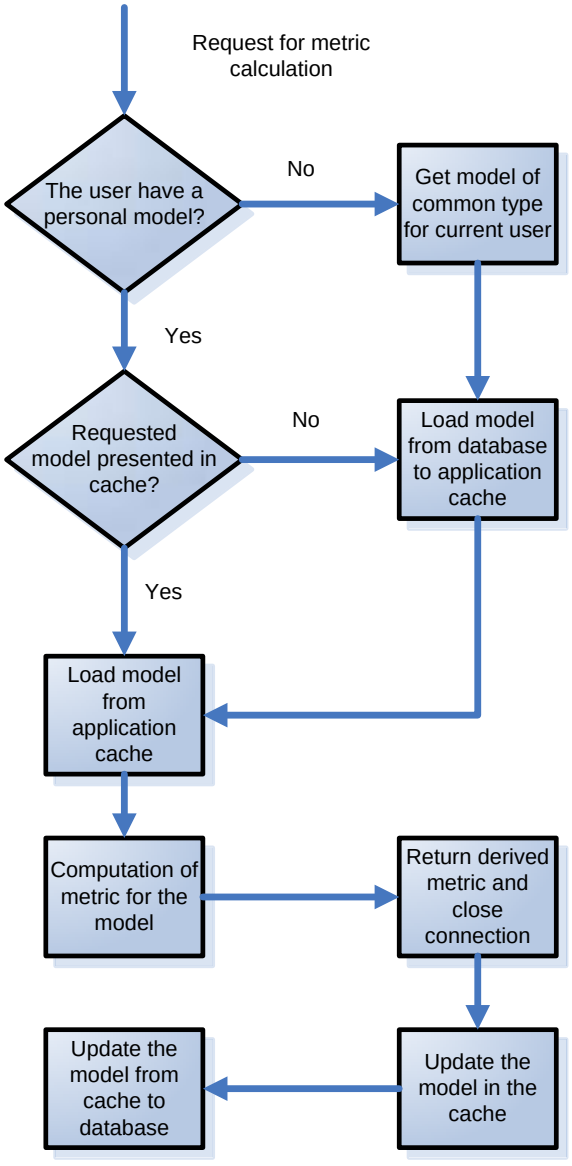

Fig. 1. Module functioning algorithm

Alternatively, to save time for the initialization process, the server can use a previously created pool; it increases efficiency, but does not eliminate the problem of limiting the number of available processes.

Using system threads instead of system processes to handle each request

On the so-called "Light-servers", depending on the implementation of threads in the operating system, the amount of resources needed to create a thread can be nearly the same as the necessary resources to create a system process or significantly less (depending on task specifics). But there the limit is higher and, at maintaining large numbers of simultaneous connections (thousands and above), this model may prove to be unworkable for the following reasons: consumption of the address space on the stack for each thread, a large load on the scheduler and the restriction on the number of threads in the system.

As can be seen, the second approach allows one to simultaneously handle the number of connections within the given requirements. However, there are some drawbacks:

- As the load increases, the requirements may change;
- Physically, the server needs a lot of resources to support threads, but the main load is on the modules working with models and the total load has to be taken into account.

In view of these shortcomings, there is a requirement for a more optimal way to use server resources to handle a large number of simultaneous requests.

B. Deferred Approach

In recent years, an approach called Deferred has become increasingly popular [3]. Its essence lies in the fact that when a request arrives, its processing module is called and it is assigned an event handler - "processing completed" and then the server forgets about the request received and does not spend resources on the treatment of this request. After some time, the request is processed, the server receives an event, "processing complete" and the result, which it sends as a response.

The Deferred concept differs markedly from the typical methods and also from the software implementation. In the server code, the function that uses data from remote services is called to handle the requested task. Since the Deferred ideology itself does not imply wasting time while waiting for an answer, this function immediately returns the result, despite the fact that it had not been received yet. This is achieved by returning a special object such as deferred, to which functions called handlers are added upon the return (usually, there are handlers of received results and handlers of errors) and then almost all the resources spent on the handling of the request are released. The next step comes only at the time of receiving the result of the requested operation.

Fig. 2 shows the basic organization of processing of the incoming requests, using the Deferred approach, in the case when the data received are processed by the server itself. It can be seen that the there are no advantages over a simple FIFO queue of requests. Additionally, costs are spent for resources to maintain the context of each request handler and more complex logic implementation.

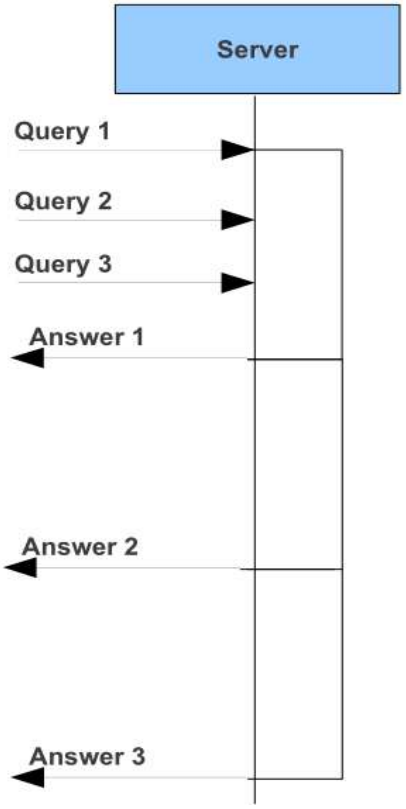

Fig. 2. Request processing by the server

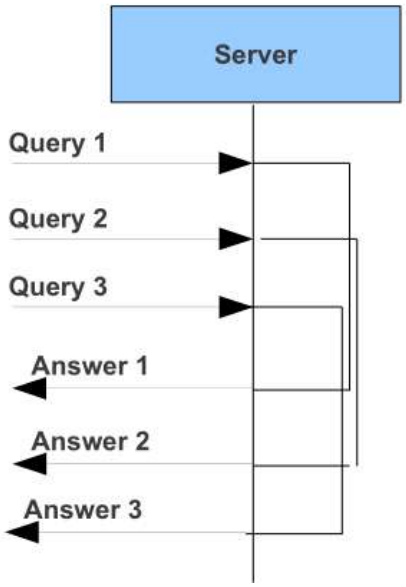

Fig. 3. Deferred when remote services are used

However, if the requests are handled by third-party services, the efficiency increases markedly.

As can be seen from Fig. 3, the overall processing time of three requests theoretically can be much smaller.

III. Reasoning for Server Type Selection

Based on the requirements available, the use of server based on the call of system processes is not possible, so the choice must be made between the use of lightweight threads or the Deferred approach.

Since there is currently no information that more than one physical server would be available, i.e., all tasks will be processed on the same computer, the main advantage of Deferred cannot be fully realized.

However, to justify the choice of the type of server, both possible configurations have to be tested.

IV. Testing

As candidates, there are selected three popular Python web servers:

- Twisted [4]
- Tornado [5]
- Cyclone

Of these, only Twisted directly supports the Deferred approach. All three use a default thread to process incoming data.

Testing methodology was as follows: the server run the modified code to which emulation of processing complex queries taking 0.005 seconds of CPU time, was added. A large number of concurrent requests arrived at the server and statistics of time taken to process them was collected.

The response delay is realized by means of Python:

```
import time
import random
rnd_delay = 0.005
time.sleep( rnd_delay )
```

When testing the Deferred approach, the delay in the code does not make sense, but an additional service is used - available on a different port local server Apache, which also returns the requested data in 0.005 seconds.

Self testing was performed using a console program ApacheBench [6]. We used the following command:

ab-n 800-c 100 http://localhost:8007/

here:

ab - the team causing the processor test;

-n - Total number of requests;

-c - Number of simultaneous requests;

http://localhost:8007/ - The host name and port for testing.

V. Experiments

A. Twisted Server + Processing Without Deferred

Concurrency Level	100	75	50	25
Time taken for tests	4.441	4.434	4.421	4.439
Requests per second	180.15	180.4	180.96	180.21
Time per request	555.102	415.732	276.309	138.729
Mean time per request	5.551	5.543	5.526	5.549

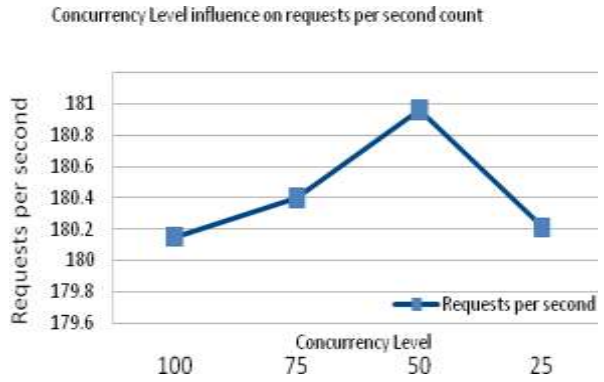

Fig. 4 Twisted, concurrency Level VS requests per second without Deferred

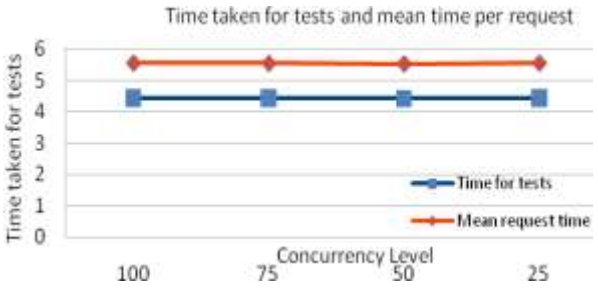

Fig. 5 Twisted, time taken for tests VS mean time per request without Deferred

This configuration processed any number (within the limits of experiment specifics) of simultaneous requests without changing the processing time per request (Fig. 4, Fig. 5). It shows that 100 concurrently created threads a modern operating system processes without significant delay.

B. Twisted Server + Deferred Processing

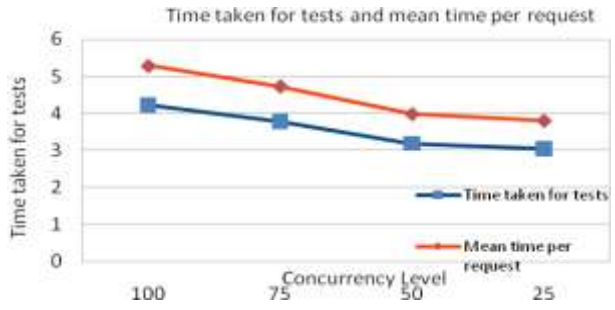

Fig. 6 Twisted, concurrency Level VS requests per second with Deferred

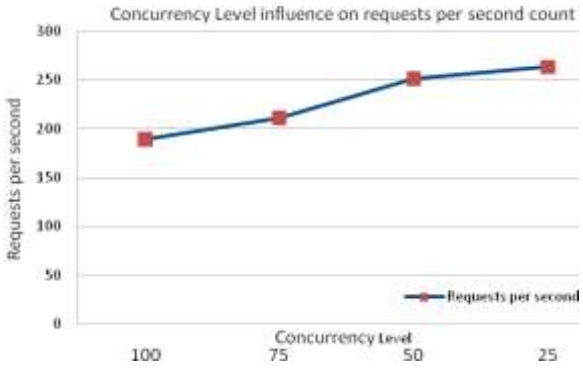

Fig. 7 Twisted, time taken for tests VS mean time per request with Deferred

Concurrency Level	100	75	50	25
Time taken for tests	4.231	3.786	3.186	3.041
Requests per second	189.09	211.28	251.07	263.1
Time per request	528.861	354.972	199.149	95.02
Mean time per request	5.289	4.733	3.983	3.801

When the number of simultaneous requests increases, the average processing time per request grows as well, but smaller number of simultaneous requests has shown better results than those obtained when using threads with corresponding amounts (Fig. 6, Fig. 7).

C. Tornado Server Without Deferred

Concurrency Level	100	75	50	25
Time taken for tests	4.475	4.498	4.496	4.5
Requests per second	178.77	177.85	177.93	177.76
Time per request	559.38	421.708	421.503	140.635
Mean time per request	5.594	5.623	5.62	5.625

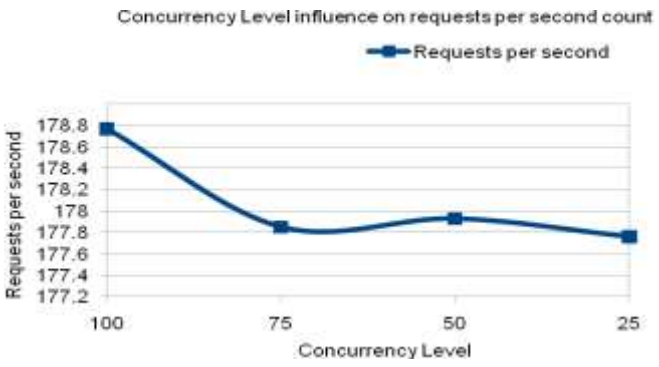

Fig. 8 Tornado, concurrency Level VS requests per second without Deferred

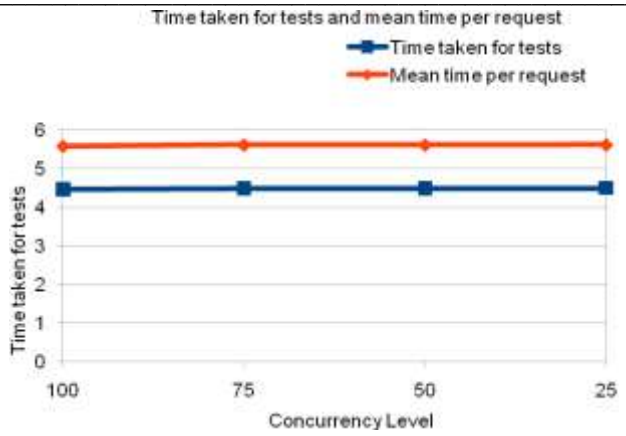

Fig. 9. Tornado, time taken for tests VS mean time per request without Deferred

The result is similar to Twisted without the use of Deferred, just a little more time on average is used to process each request (Fig. 8, Fig. 9). It also shows the efficient use of systems threads by Twisted server.

D. Cyclone Server Without Deferred

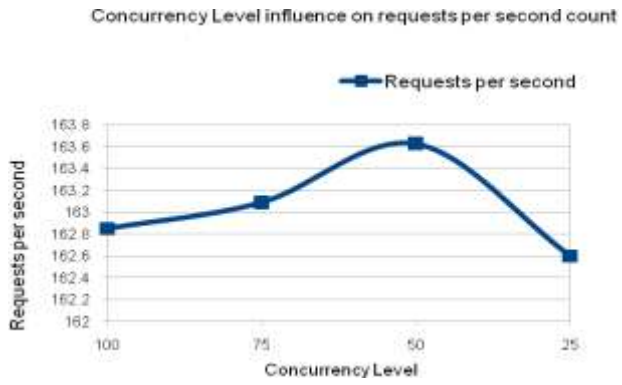

Fig. 10. Cyclone, concurrency Level VS requests per second without Deferred

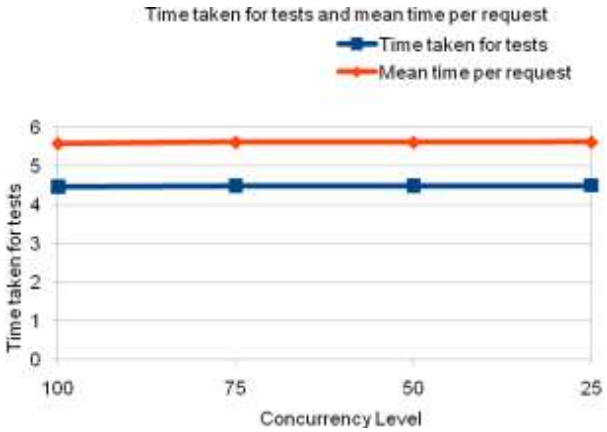

Fig. 11. Cyclone, time taken for tests VS mean time per request without Deferred

Concurrency Level	100	75	50	25
Time taken for tests	4.913	4.905	4.889	4.92
Requests per second	162.85	163.09	163.63	162.6
Time per request	614.071	459.876	305.567	153.75
Mean time per request	6.141	6.132	6.111	6.15

This server is based on the Twisted protocol, so similar behaviour is expected. The result, showing that processing each request was a little more time-consuming, as compared to Twisted server, was predictable (Fig. 10, Fig. 11).

E. PHP — Performance of Server that Emulates a Remote Service

Concurrency Level	100	75	50	25
Time taken for tests	4.231	3.786	3.186	3.041
Requests per second	189.09	211.28	251.07	263.1
Time per request	528.861	354.972	199.149	95.02
Mean time per request	5.289	4.733	3.983	3.801

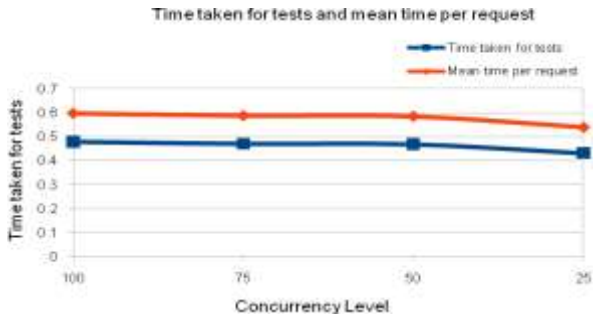

Fig. 12. Apache, concurrency Level VS requests per second

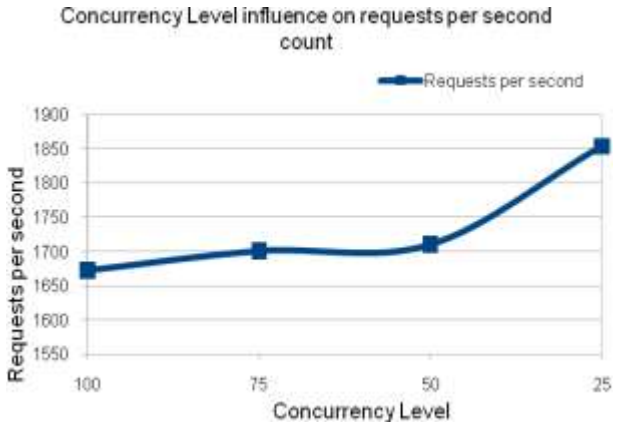

Fig. 13. Apache, time taken for tests VS mean time per request

To assess the impact of speed of processing each request, when emulating a remote server by means of PHP, a corresponding set of experiments was conducted.

However, there server coped with the requests very confidently and when the number of simultaneous requests

increased to 100, the average processing time of each request remained virtually unchanged (Fig. 12, Fig. 13).

VI. CONCLUSIONS

Experimental testing showed that using one physical server implementing the processing of incoming requests as well as direct operations on the model of user behaviour, the complexity of the Deferred approach may exceed the benefits derived from it. However, the result is still better when Deferred is used. Furthermore, with an increase in the number of simultaneously processed models in the future this approach will provide the best average time of processing each transaction.

REFERENCES

[1] P. A. Osipov and A. N. Borisov; Abnormal action detection based on Markov models; Automatic Control and Computer Sciences; Volume 41 / 2007 - Volume 45 / 2011; ISSN 0146-4116 (Print) 1558-108X (Online); May 05, 2011.
[2] Millman, K. Jarrod; Aivazis, Michael; Python for Scientists and Engineers. University of California, Berkeley. Computational Science & Engineering, IEEE. Volume 13 Issue 2. ISSN: 1521-9615.
[3] Event-Driven Programming for Embedded Systems by Miro Samek, Newnes 2008. ISBN-10: 0750687061; ISBN-13: 978-0750687065.
[4] Abe Fettig; Twisted Network Programming Essentials; 'Reilly Media; 2005; 238p; Print ISBN: 978-0-596-10032-2 | ISBN 10: 0-596-10032-9.
[5] Project web-page: www.tornadoweb.org
[6] ApacheBench; Copyright 1996 Adam Twiss, Zeus Technology Ltd.

Pavel Osipov is a Doctoral student at the Institute of Information Technology, Riga Technical University. He received his mg.sc.ing. degree from Transport and Telecommunications Institute, Riga. His research interests include web data mining, geo-location services application, machine learning and knowledge extraction. Additionally Python programming language usage for scientific tasks becomes more and more interesting theme to research, including servers, graphs, data proceeding and other tasks.
E-mail: pavels.osipovs@rtu.lv

Arkady Borisov has a Doctoral degree in Technical Sciences in the field of Control in Technical Systems and the Dr.habil.sc.comp. degree.
He is a Professor of Computer Science at the Faculty of Computer Science and Information Technology, Riga Technical University (Latvia). His research interests include fuzzy sets, fuzzy logic and computational intelligence. He has 210 publications in the field.
He has supervised a number of national research grants and participated in the European research project ECLIPS.
E-mail: arkadijs.borisovs@cs.rtu.lv

Pāvels Osipovs, Arkādijs Borisovs. *Deferred* pieejas izmantošana zinātniskos lietojumos
Šajā rakstā ir apskatīti mūsdienīgi serveru realizācijas varianti, kuri apkalpo zinātniskos vai praktiskos uzdevumus ar stingriem ierobežojumiem, kuri attiecas uz katras transakcijas minimālo apkalpošanas laiku un lielām slodzēm. Rakstā ir izskatīts labākā servera izvēles jautājums, ar kura palīdzību varētu realizēt lietotāja uzvedības metrikas izskaitļošanas uzdevumu. Shematiski tika apskatīta mērķtiecīgas sistēmas kopējā struktūra no realizācijas viedokļa, izmantojot sistēmu ar klients–serviss pieeju. Pats uzdevums un tā realizācijai izmantotie algoritmi ir izskatīti iepriekšējā darbā, savukārt darbā izmantota realizācija, pielietojot programmēšanas valodu PHP, nespēja apmierināt visas stingrās prasības attiecībā uz katra pieprasījuma apstrādes laiku, līdz ar to radās nepieciešamība izmantot citas pieejas, kuru izpēte arī kļuva par pamatu šim rakstam.
Šajā darbā tika izmantota Python valoda, kura kalpo kā vispiemērotākais un visspēcīgākais līdzeklis gan praktisko, gan līdzīga tipa zinātnisko uzdevumu risināšanai. Tika izmantota arī klienta servera pieeja, kad serveris (realizēts ar Python valodas palīdzību) konfigurēts pašreizējo uzdevumu atrisināšanai. Papildus izmantotie instrumenti, kuri tika pielietoti modeļa apstrādei un glabāšanai, rakstā netiek apskatīti, jo dotajā etapā ir nepieciešams novērtēt vienas pieejas priekšrocības attiecībā pret citu. Ir apskatītas domēna īpašības un trīs iespējamās servera funkcionēšanas pieejas no katras transakcijas izmantošanas vidējā laika minimizēšanas viedokļa. Iespēja izsaukt sistēmas procesus, sistēmas pavedienus un jaunu pieeju. Notikumiem bagāta reaģēšana – *Deffered*. Tika novērtēta iespēja izmantot katru iespējamo pieeju piešķirtā uzdevuma robežās. Lai iegūtu servera efektivitātes skaitlisko novērtējumu, tika veikta to testēšana, izmantojot dažādus serveru realizācijas veidus. Tika iegūti serveru uzvedības rezultāti, izmantojot dažādas konfigurācijas pie lielām slodzēm. Tika atklāts realizācijas variants, kurš ir vislabāk piemērots domēna prasībām un īpatnībām. Kā jau tika prognozēts, *Deferred* pieeja parādīja sevi kā vispiemērotākā pieeja.

Павел Осипов, Аркадий Борисов. Использование *Deferred* подхода в научных приложениях
В статье рассмотрены современные варианты реализации серверов, обслуживающих научные либо практические задачи в рамках жёстких ограничений на минимальное время обработки каждой транзакции и при наличии высоких нагрузок. Рассмотрена задача выбора наилучшего сервера для реализации задачи вычисления метрики поведения пользователя. Схематически рассмотрена общая структура целевой системы с точки зрения её реализации с использованием клиент-серверного подхода. Сама задача и использованные для её реализации алгоритмы рассмотрены в предыдущей работе, однако использованная в ней реализация на языке программирования PHP не смогла удовлетворить жёстким требованиям на время обработки каждого запроса. Возникла потребность применения других подходов, исследование которых и послужило базой для данной статьи.
В текущей реализации использован язык Python, как наиболее подходящий инструмент для решения как практических, так и научных задач подобного типа. Также использован клиент-серверный подход, когда сервер (также реализованный на Python) сконфигурирован именно для решения текущей задачи. Дополнительные инструменты для хранения и обработки модели в статье не рассматриваются, так как на данном этапе требуется оценить преимущества одного из подходов перед другими.
Рассмотрены особенности предметной области и три возможных подхода к функционированию сервера с точки зрения минимизации среднего времени оценки каждой транзакции: возможность вызова системных процессов, системных нитей и новый подход: событийное реагирование — *Deferred*. Произведена оценка возможностей использования каждого подхода в рамках поставленной задачи. Для получения численной оценки эффективности серверов было проведено их тестирование с применением различных вариантов их реализации. Получены данные о поведении различных серверов при использовании различных конфигураций и под различной нагрузкой. Выявлен вариант реализации, наиболее соответствующий особенностям и требованиям текущей предметной области. Как и ожидалось, *Deferred* подход показал себя наиболее подходящим подходом.