Responsive, resilient, elastic and message driven system solving scalability problems of course registrations

Janina Mincer-Daszkiewicz, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, Banacha 2, 02-097 Warszawa, jmd@mimuw.edu.pl

Keywords
Course registration, reactive system, Scala, model of cooperating actors, Single Page Application, NoSQL database, USOS, USOS API

1. ABSTRACT

Course registration is one of the most demanding functionalities of student management information systems. In the *University Study-Oriented System* (in short: USOS [7]) which is used in more than 40 Higher Education Institutions (in short: HEIs) in Poland (gathered in MUCI consortium [4]) we cope with it from the very beginning which dates back to 2000. There are various registration scenarios, some of them are more appealing to students and administration giving immediate feedback and being fully automatic, others are more practical and less demanding with respect to computing power. The optimal solution should stay user friendly but also get burden off the university administration, meaning both student's offices and IT departments.

We attacked the problem by a new approach inspired by the *Reactive Manifesto* ([5]) and built a *reactive system*, responsive, resilient, elastic and message driven. The registration is run in micro rounds until all the interested students register to courses which are offered by the university. Micro rounds last app. 5 minutes, so first come first served approach is avoided but anyway feedback is almost immediate. To achieve the respective responsiveness, scalability and resilience, involved technologies were chosen carefully. Backend server runs in asynchronous and distributed computation model of cooperating actors which exchange messages. Data is stored in NoSQL database kept in main memory for most of the time and the frontend is designed as a dynamic single page web application. The scalability of the new solution was tested using infrastructure of the University of Warsaw, the biggest HEI in Poland with more than 50 thousand students.

2. INTRODUCTION

Course registration is one of the most demanding functionalities of student management information systems. In USOS we cope with it from the very beginning (which means 15 years). Registrations based on the first come first served approach are more appealing to students who get immediate response to their registration requests and administration who need not be involved in any hand made decisions, but on the other hand are the most demanding with respect to computing power. If not supported by highly scalable infrastructure and carefully tuned software, may lead to disaster. There is a way of delivering more computing power during rush hours (e.g. by buying it from a cloud provider), but it involves additional cost and extra organizational effort without guarantee that the problem will be completely solved.

We attacked the problem in a different way. Inspired by the *Reactive Manifesto*, we decided to build a reactive system, responsive, resilient, elastic and message driven. First come first served approach is replaced by micro rounds (short time periods lasting app. 5 minutes). The registration starts with the first active micro round. Students deliver their preferences using a dynamic single page web application. During a subsequent short time interval backend server makes registration decisions which are immediately displayed to students. For some users the registration ends, those less lucky continue the game delivering new sets of preferences in the next micro round.

The new approach compliant with the Reactive Manifesto demands new architecture and technologies. The system consists of three layers, which are event-driven. The user interface is

implemented using AngularJS as a dynamic single page web application. The application server runs in asynchronous and distributed computation model that allows to scale both vertically and horizontally. The technology used is Scala and Akka. Backend functionalities are delivered to other layers by USOS API ([3]), which is public API to the system and USOS application server. Temporary and permanent data storage layer is focused on the performance of operations while ensuring data safety. This is supported by NoSQL database MongoDB.

University Study-Oriented System is an integrated suite of applications, running on top of a central Oracle database, installed in more than 40 higher education institutions in Poland, altogether offering educational services to almost ¼ of the population of students from public sector HEIs.

In this paper we describe the whole project from design through implementation to deployment. In Chapter 3 various approaches to registration implemented in USOS are described. Section 3.4 is devoted to the new registration approach. In Chapter 4 we explain the technological aspects of the solution and in chapter 5 show test results. Conclusions are drawn in the last chapter.

3. REGISTRATION FOR COURSES AND CLASSES IN USOS

3.1. Introduction

In modern higher education institutions flexibility of study programs means in particular that students may freely choose a substantial part of their curricula. The software support is needed to help to make good choices and enable to pass information on what has been selected. This information is later used for student assessment and checking whether a student met requirements of a particular study program.

The important aspect of this process is the huge amount of data which needs to be gathered in the system. Doing by hand all the necessary registration activities would be a substantial burden for every student administration office. The only economically acceptable solution is to let the students do it by themselves.

Course registration is just the first step in the registration procedure. At the next stage students choose particular student groups (classes, labs, lectoria, project teams etc.) which gather at various times and days of the week. This step has a significant impact on a student's weekly schedule and has to be carried out taking his/her preferences into account.

Web based registration is vulnerable to scalability problems. In a medium-size faculty, like Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw, every year about 1500 students register for about 300 courses. In the University of Warsaw in 26 didactic organizational units study more than 50 thousand students, the largest faculties have more than 6000 students. There are approximately 20 thousand courses offered every academic year. These numbers give some impression about the scale of the process.

For the 15 years of the USOS development we designed and implemented a couple of registration procedures. The main two are described in the following sections. In the end the newly designed registration method is described.

3.2. Two-phase registration

In this model students register for courses during the first phase and for classes during the second phase. The procedure does not require students to be available during registration at the same time or at the same place.

During the first phase students choose courses they would like to attend. The decision whether a particular student is accepted for a particular course is made off-line. When the registration module is switched off, the authorities get the lists of students asking for acceptance for particular courses and make decisions. If the lower limit on the number of course participants is not reached, the course is cancelled. If there is no limit on the number of students who can be accepted or the limit is not exceeded, all the students are allowed to take the course. Otherwise students are accepted according to some predefined criteria. Usually these criteria are difficult to implement so it is more reasonable to make the decisions by hand. This may seem cumbersome but in the past all attempts

to implement this process failed due to many exceptions to general rules. It means that students have to wait until the end of the course registration to get the results.

Then the group registration module is switched on. These students who were accepted for the course can now state their preferences concerning various study groups of that course. These preferences can be expressed in a couple of ways, e.g. a student may create some variants of his/her weekly schedule and/or list classes which should be chosen with high/low priority and/or list days and time periods which should be chosen with high/low priority (e.g. low priority given to Friday evenings, high priority given to classes starting after 11 am). After a couple of days the module is switched off and the group registration engine assigns students to classes taking into account their preferences, trying to minimize the number of time conflicts and preserving other requirements. Sophisticated algorithms were implemented for this engine (see e.g. [2]).

During the second stage students are guaranteed that they will be accepted to one of the groups, but not necessarily to the favored one. It may even happen that a student has to drop the course because the group schedule is unacceptable.

There is still another module which may be (optionally) used. This is called stock-exchange, although what is really being exchanged are places in particular classes. A student can submit a place in a particular class for sale and ask for a place in a different class. After a couple of days the stock-exchange engine is switched on and it automatically matches buyers with sellers (even in cycles longer than 2) exchanging students among groups.

Such registration procedure works reasonably well at the faculty level, where students take courses which are in line with their specialization. It is relatively easy for the faculty authorities to make decisions whom to accept and whom to reject. Students know where to complain in case they feel they are not treated fair.

3.3. Direct (combined) registration for courses and classes

Curriculum of a typical study program involves also some courses which are not offered locally by the faculty to its students, but are delivered by some designated faculties to all students of the university. These are for example foreign language courses which are offered by the Center for Foreign Language Teaching, physical education classes which are offered by the Physical Education and Sports Center or general introductory lectures on philosophy which are provided by the Philosophy Faculty.

There are also faculties which prefer fully automatic registrations not involving human decisions.

The model of registration for such courses and such faculties is different and needs separate software support. Students register directly for course groups on a first come first served basis. This means that if only the group limit is not yet reached and a student is entitled for registration, he/she gets registered and this registration gets approved straight away. Students get immediate feedback, but they have to be on-line on the moment registration starts and compete with thousands of other students.

There is a variant of this procedure called token based registration [1]. Every student receives a number of tokens of different kinds and registers to classes paying with these tokens. Each course has its price given in tokens, usually this price corresponds to the number of hours. After reaching class limits registration to this class is suspended. When a student registers, the system charges him/her the appropriate amount of tokens of the proper kind. If more tokens are needed than the student possesses, he/she should be given the possibility to cancel or to approve the operation before the due payment is charged. The student should transfer money using his/her unique account number.

Tokens serve one more purpose. They set a limit for the maximum number of groups a student can choose in one registration round. This limit is used to prevent over-registration. If there was no limit, some students might register to many groups, just to postpone the final decision to the very last moment and unregister right before the registration is switched-off.

3.4. Micro rounds

Direct registrations are more appealing to students and administration giving immediate feedback and being fully automatic, two phase registrations based on preferences are more user-oriented by giving the possibility to match supply and demand and also by being less demanding with respect to computing power.

The optimal solution should stay user friendly but also get burden off the university administration, meaning both student's offices and IT departments.

The new type of registration is run in micro rounds (short time periods lasting app. 5 minutes) until all the interested students register. The registration starts with the first active micro round. Students enter their preferences using a dynamic single page web application. During a subsequent time interval backend server makes registration decisions which are immediately displayed to students. Having access to all requests, the server may optimize distribution of places between students. For some users the registration ends, those less lucky continue the game delivering new preferences in the next active micro round.

Registration may be in one of the following stages (see Figure 1):

1. Waiting — registration has not started yet but details of courses and classes are available to students who can plan in advance their preferable schedules.
2. Active — micro round is active, students may define preferences. The micro round is short but long enough to diminish the pick loads of first come first served approach.
3. Break — interval between two active micro rounds, students' requests are processed, results of registrations are displayed to students and stored in USOS database.
4. Finished — registration is finished

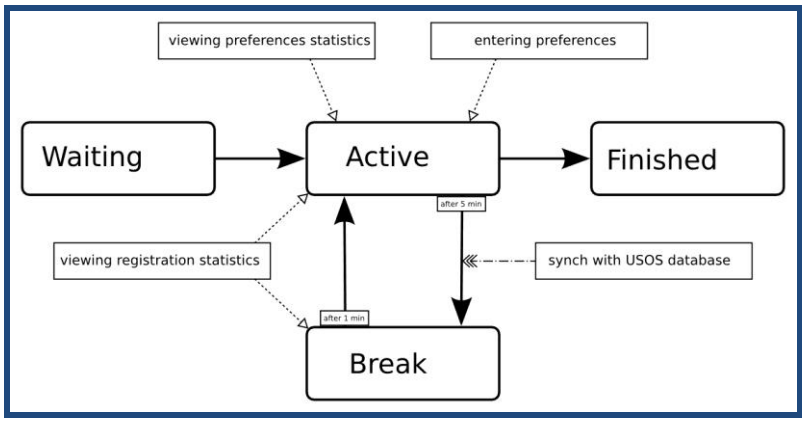

Figure 1 Stages of the registration with micro rounds

This model of registrations not only seems to be an acceptable tradeoff between various requirements but also enables effective implementation. Students' requests can be processed in parallel — preferences from various students are independent and need not be stored immediately in the database. They are kept in memory and synchronized with the central databases during intervals between active micro rounds. During synchronization all the records are transferred to the database in an efficient manner, which helps to increase throughput and to reduce the load of the database. The implementation details are described in Chapter 4.

4. DESIGN AND IMPLEMENTATION

According to the *Reactive Manifesto* "Systems built as *Reactive Systems* are more **flexible**, **loosely-coupled** and **scalable**. This makes them easier to **develop** and amenable to **change**. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback."

Requirements against the new model of registration comply with the principles of reactive applications — applications which, generally speaking, are event driven. By placing the new registration system in this context we can take advantage of the universal knowledge and experience, as well as patterns and ready solutions developed by the creation of similar systems.

4.1. System architecture

General overview of the system architecture is shown in Figure 2. It comprises the following layers:

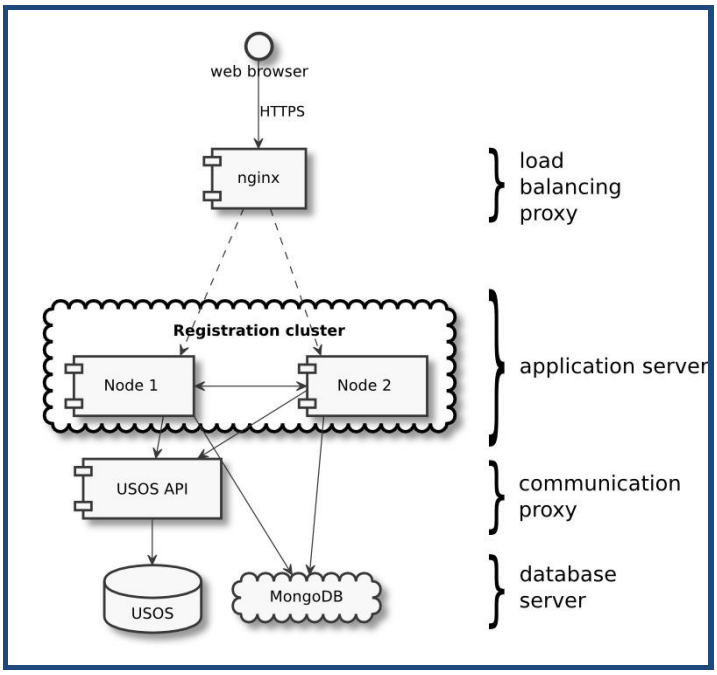

Figure 2 The overview of system architecture

1. **User interface**

   User interface is a separate module of USOSweb, web based application for students and staff. High performance should be guaranteed in the frontend, as well as in the backend. We decided to implement user interface as a single page application. Such applications can reduce the amount of data sent over the network and lower the load on the HTTP server. In this approach, all HTML, JavaScript and CSS is taken during single page load or downloaded dynamically when necessary, usually in response to user actions. Downloaded resources are placed in the browser cache. More details on a single page approach is given in section 4.3.

2. **Load balancing proxy**

   In order to ensure proper performance and scalability we decided that the application server will be distributed. That means that we need a proxy responsible for load balancing. Mechanism used must comply with the following assumptions:

   a. support for Server-Sent Events,
   b. routing of calls from the same client to the same node — so called sticky sessions,
   c. support for secure SSL connection,
   d. support for proxy cache.

   From various possible candidates **nginx** has been chosen.

3. **Application server**

   The application server is a distributed **cluster of nodes**, from which everyone can accept any connections from customers. These can be both regular HTTP connections and asynchronous connections to send notifications. The nodes communicate between each other to support

load balancing and to provide resistance to failure. Application server is described in detail in section 4.4.

4. **Communication proxy**

   USOS is a suite of software applications built in a distributed architecture. There is a central Oracle database with many packages, functions, triggers, jobs, etc. USOS API is a standard REST-like interface to data gathered in USOS, publicly available, well documented, with guaranteed backward compatibility. API methods are gathered in modules, e.g. *users*, *courses*, *terms*, *geo*, *mailing*. Each API method is well-documented in XML, so methods and their parameters may be automatically validated. Results of methods are delivered in XML or JSON format. Methods with the administrative key may fetch data from the central Oracle database, the others get access only to a local MySQL database (which is periodically synchronized with the central one). USOS API implements business logic and makes it available to other modules of the software system.

   One HTTP request stores all served requests made for one course conducted at the end of each micro round. Thanks to this database is not unduly burden even when many students try to register in a short time.

5. **Database server**

   The application server uses USOS API to retrieve and store data from/to Oracle database (such as the fact that someone is registered to a course). Data associated with the application itself, such as sessions, registration requests, and also provisionally collected data on courses and classes, are stored in a separate database. Because this database will have to serve a large number of read and write requests, which can be processed in parallel (requests of different customers do not require any synchronization), the data store does not have to comply with all the requirements posed by relational databases. NoSQL databases give the opportunity to achieve higher performance at the expense of, among others, deterioration of insulation of transactions, and are more cost efficient than commonly used relational databases.

   We considered a couple of NoSQL solutions but finally have chosen MongoDB. Mongo nodes can be reproduced increasing the degree of data replication, and also their safety. Data can also be distracted between nodes, enabling greater degree of parallelization of read and write operations.

   The chosen solution allows to achieve two important goals. The first is to take off the load from Oracle with minimal interference with the existing structure present in the database. Data from Oracle database are made available by USOS API in a convenient form that allows to group read and write requests. The second objective is to use the local database MongoDB, so that several common operations performed by the students — registration requests — are processed locally, in an efficient manner. We have achieved this by reducing the required number of queries to the database (especially writes) and assuring high independence between data structures for different students.

This architecture has a decisive impact on the performance and scalability of the system. Chosen components and communication between them ensure high effectiveness and resistance to failures. What's more, some components of the system, e.g. database and load balancing proxy, are interchangeable. If in the future better solutions appear, it should not be difficult to use them instead of the chosen ones.

4.2. Communication between system components

Communication between system components is also thoroughly designed. USOS API connects directly with USOS Oracle database. For this purpose it uses a pool of TCP connections. Thanks to this no time is lost for establishing a new connection when a new request appears for processing. Together with the reduction of superfluous operations, it minimizes database load during registration.

Application server communicates with USOS API via HTTPS. Data are transmitted in JSON format.

Communication between user interface and the application server takes place in two ways:

a. synchronous — HTTP calls of server methods,
b. asynchronous — implemented by the SockJS protocol. In case of new web browsers it is implemented via Server-Sent Events which provide a standardized way for the server to send content to the browser without being solicited by the client, and allowing for messages to be passed back and forth while keeping the connection open. In this way a two-way (bi-directional) ongoing conversation can take place between a browser and the server.

4.3. User interface (frontend)

User interface is not just a collection of static pages, but a separate web application. It has been written — according to the latest design pattern in creating interfaces — using SPA (Single Page Application) architecture.

The latest development in technology, e.g. HTML5 and fast runtime environments for JavaScript code, has allowed developers to create large and functional applications running on a client side, not — as before — on a server platform. The concept of one-page application refers to web application that runs in a browser, can immediately respond to a user action without sending to the server a request to render a new page, using the capabilities of current technologies and browsers. A classic example of such application is Gmail. SPA approach has many advantages:

- **Improved productivity** —most of the interaction is handled without the server since a large portion of logic is implemented on the client side. When the page is loaded the client retrieves the data from the server in JSON format which limits the flow of data over the network comparing with the traditional approach. The page consists mainly of static files, which makes caching easier and gives a better effect.
- **Separation of** the client side of an application from the server side — in SPA approach client side is separated from the server side and the two parts can be developed by independent teams of programmers. Their collaboration can be limited to determining API and method of communication. Both parts can be distributed separately provided that they are compatible with each other.
- **Easier testing** — easier testing is a consequence of the previous advantages. Greater modularity of the system promotes testing.
- **Ready API for other clients** — one page applications draw data from the server using API. This API provides access to the data also to other customers, e.g. mobile applications.
- **More fluid user experience** — the user running SPA has the impression as if it was a desktop application. The application immediately responds to user actions. Thanks to this user in almost every minute knows what's going on.

There are also some drawbacks of SPA which can be overcome by a careful design.

We use client-side library AngularJS which is web browser JavaScript framework adopting SPA principles. AngularJS's templating is based on bidirectional data binding. The HTML template is compiled in the browser. It implements the MVVM pattern to separate presentation, data, and logic components. Angular brings traditionally server-side services, such as view-dependent controllers, to client-side web applications. Consequently, much of the burden on the server can be reduced.

Students register in USOSweb on a web page showing available classes on a timetable, as demonstrated in Figure 3.

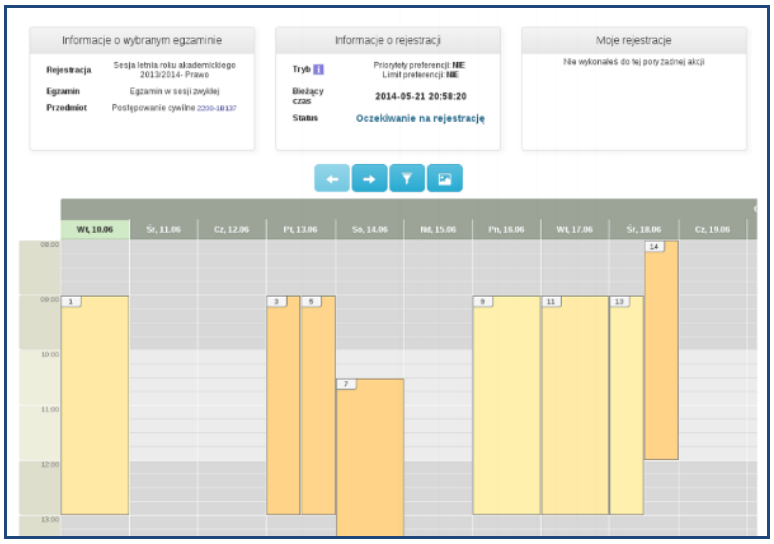

Figure 3 User interface

4.4. Application server (backend)

Internal architecture of backend

Application server is responsible for handling the registration process, in particular:

- providing information on available registrations, courses, and classes,
- collecting registration requests,
- aggregating and delivering to end users statistics on registrations and registration requests,
- processing of requests, calculating the results and storing them in the database.

It is implemented as a stand-alone application serving as a proxy between user interface and USOS API. It is run in Java Virtual Machine. Data between user interface and the application servers and between application server and USOS API is send by HTTP protocol in JSON format.

Internal architecture of the server is based on the **model of cooperating actors.** The central concept is an **actor** — from an abstract point of view it is a single thread, executing its actions sequentially. Programming with the use of actors is an effective way to implement concurrent and distributed systems. By hiding issues such as the physical location of the components of an application (represented by actors), it allows for easy dispersion of these components on multiple machines.

Actors communicate by sending messages using a message queue. Actor can be chosen for execution if its message queue is not empty. In practice actors are implemented using a pool of threads, an actor is chosen for execution by runtime but should give back a thread to the pool before blocking.

Server is mostly written in Scala with some parts written in Java. Scala is a functional/object-oriented programming language with a very strong static type system. Programs written in Scala are very concise and smaller in size than programs written in other general purpose programming languages. We use Akka toolkit which supports the model of actor-based concurrency, in particular the libraries akka-actor and akka-cluster.

System scalability

Server scales according to the number of users — students participating in the registration. Model of actors and Akka make it possible to scale both vertically (for more processors) and horizontally (for more machines) — distribution of the server on multiple physical machines is almost imperceptible in the code of business logic (responsible for handling the registration).

Registration model is very simple. Preferences of a particular student are independent of preferences made by another student. All necessary validation can be made in parallel with validations done for other students.

By a computation node we understand single server process run as a separate instance of JVM. Each process is given its share of resources — number of processors (threads) and RAM. Vertical scaling means an increase of application performance with increasing computational power of a single node.

Requests coming from a particular student are handled by a single actor. User interface in the browser maintains a connection using SockJS protocol. The user can log on to the server from many different computers, browsers, their windows or tabs — each copy of an open interface corresponds to one SockJS connection. The server manages all connections associated with one account by a common instance of an actor. Thus, we consciously limit the impact of the simultaneous actions performed by many users logged into the account of the same student — the most important operations in the context of a single account are synchronized, which contributes to fair distribution of resources between actors, which translates to the performance felt by all users.

Horizontal scaling means an increase of application performance with increasing the number of computational nodes. Thanks to Akka, remote communication between actors is almost as simple as between local actors. Philosophy behind Akka suggests such programming that implies nothing about the time or delivery of messages. Written code should be both asynchronous (also taking into account potentially very long response time), as well as handle lost messages (or replies). With this way of writing applications, decision about distributing actors on many nodes can be taken at a later time, e.g. in a configuration file.

5. TESTS

The preliminary tests were run in a computer lab of the Faculty of Mathematics, Informatics, and Mechanics of the University of Warsaw, on desktop class computers. Each machine hosted one MongoDB and one application server. Experiments were conducted for the increasing number of machines and increasing number of requests per second. We looked for the moment the system is not able to handle the increasing workload. We show just a couple of the results. On all attached drawings axis X shows the time of the experiment in seconds, left (black) axis Y shows the overall number of requests initiated in that second, right (red) axis Y shows the average response time for the request started in that second (the smaller the better).

Figure 4 compares the results for 1500 requests per second for one machine (left diagram) and two machines (right diagram). It can be seen that by adding the second machine we reduce the average response time to a value imperceptible to the human.

Figure 5 compares the results for 2500 requests per second for two machines (left diagram) and three machines (right diagram). Again, it can be seen that the system scales well since one more machine allows to lower the average response time to an acceptable value of less than 0,2 seconds.

We plan to run more tests using the central server infrastructure of the University of Warsaw. With demonstrated scalability obtained on desktop computers, we may expect significantly higher performance on professional servers, as well as their clusters.

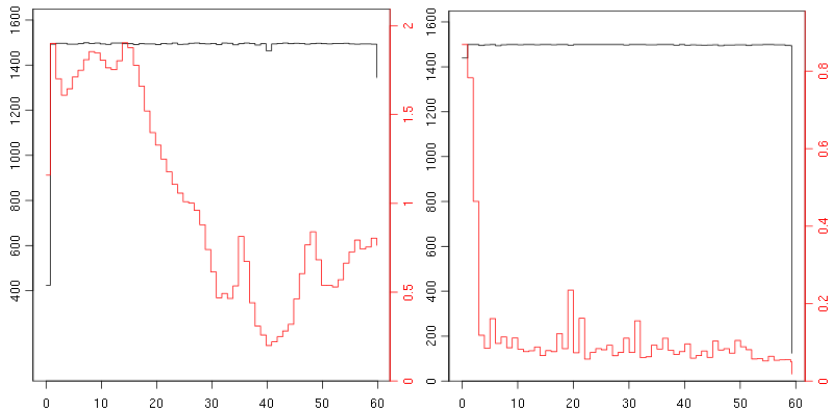

Figure 4 Test results for one machine (left) and two machines (right) for 1500 requests per second

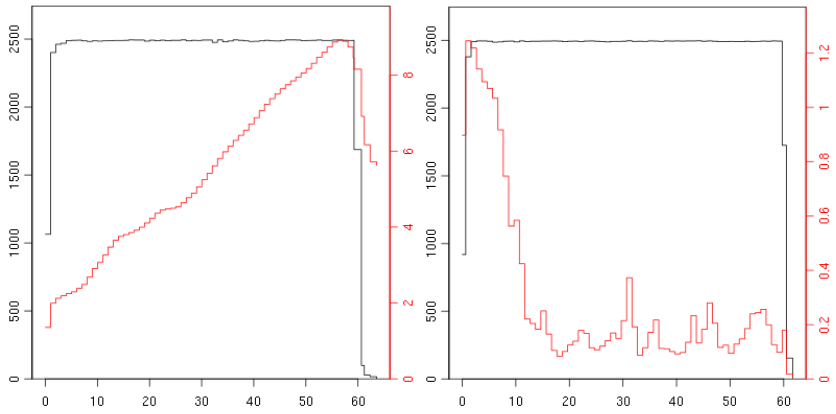

Figure 5 Test results for two machines (left) and three machines (right) for 2500 requests per second

6. CONCLUSIONS

Through the use of modern programming techniques the new registration system of USOS reaches high scalability which makes it possible to increase throughput at key moments of the academic year. It is possible to increase performance within the same machine by increasing amount of available computing resources. In case of special requirements, it is possible to run the application on multiple machines. Scaling the application vertically and horizontally is possible thanks to the use of the model of actors and non-blocking asynchronous programming. Both the operation of the server and the user interface is event-driven. The end result is a registration system capable of handling 1500 requests per second on a single desktop class computer, while preserving response time acceptable to the human. With demonstrated scalability, performance on professional servers, as well as their clusters, will be significantly better.

Behavior on multiple machines is essential to achieve reliability in case of failure of individual machines. We are ready for this thanks to the use of the programming platform that implements the model of actors along with support for distributed communication between components. In our opinion, the language Scala and Akka library were a good choice.

First real life usage of the new registration system is planned for spring 2015 registrations.

7. Acknowledgements

This paper is based on the Master thesis of Grzegorz Swatowski, Maxymilian Śmiech and Michał Żak [6], supervised by Janina Mincer-Daszkiewicz. All programming work was done by Grzegorz, Max and Michał, under the guidance of a senior programmer Michał Kurzydłowski.

8. REFERENCES

[1] Ciebiera K., Mincer-Daszkiewicz J., Waleń T. (2004). *New Course Registration Module for the University Study-Oriented System*. EUNIS 2004, Bled.

[2] Ciebiera K., Mucha M. (2014). *Student-class Assignment Optimization Using Simulated Annealing*. EUNIS 2014, Umeå.
http://www.eunis.org/download/2014/papers/eunis2014_submission_49.pdf.

[3] Mincer-Daszkiewicz J. (2012). *USOS API — how to open universities to Web 2.0 community by data sharing*. EUNIS 2012, Vila Real.
http://www.eunis.pt/index.php/programme/full-programme.

[4] MUCI consortium. Retrieved in January 2015 from: http://muci.edu.pl.

[5] The Reactive Manifesto. Retrieved in January 2015 from: http://www.reactivemanifesto.org/.

[6] Swatowski G., Śmiech M., Żak M. (2014). *USOSregistration — scalable registration system*. Master thesis, University of Warsaw (in Polish).

[7] USOS website. *University Study Oriented System*. Retrieved in January 2015 from:
http://usos.edu.pl.

9. AUTHORS' BIOGRAPHIES

Janina Mincer-Daszkiewicz graduated in computer science in the University of Warsaw, Poland, and obtained a Ph.D. degree in math from the same university. She is an associate professor in Computer Science at the Faculty of Mathematics, Informatics and Mechanics at the University of Warsaw. Her main fields of research include operating systems, distributed systems, performance evaluation and software engineering. Since 1999, she leads a project for the development of a student management information system USOS, which is used in about 40 Polish Higher Education Institutions, gathered in the MUCI consortium. In 2008, she started the Mobility Project with RS3G. Janina takes active part in many nation-wide projects in Poland.