We *Publish*, You *Subscribe* — *Hubbub* as a Natural Habitat for Students and Academic Teachers

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

Keywords
Publish-subscribe, PubSubHubbub protocol, PULL vs PUSH, notifications, OAuth, USOS API

1. ABSTRACT

Mobile applications are becoming a popular tool providing access to information stored in student management information systems (SMIS). There is no question of whether to allow such access, the question is how to deliver information in real time (instantly), in a user friendly manner, without exposing university servers to crashes in peak hours. The solution is **publish-subscribe** protocol, where information is not **pulled** by a subscriber (information consumer), but is **pushed** by a publisher (information provider) to all subscribers (with the help of the **hub**). Data confidentiality is ensured by the **OAuth** protocol [4]. The subject of this paper are new methods of public **API** for **USOS** [2], which implement publish-subscribe paradigm, and a notification daemon which plays the role of the hub. *USOS* comes from *University Study-Oriented System* [7], product of MUCI consortium [3], which is deployed in 40 HEIs in Poland.

2. INTRODUCTION

Members of academic community, i.e. students and academic teachers, want to use smart phones and mobile Internet to access data stored in academic databases. Providers of student management information systems (SMIS) for higher education offer mobile applications to fulfill these needs. There is no question of whether to allow such access, the question and challenge is how to do it fast, securely and in a user friendly manner.

University Study-Oriented System (in short: USOS [7]) is an integrated suite of applications, running on top of a central Oracle database, installed in 40 higher education institutions in Poland, altogether offering educational services to almost ¼ of the population of students from public sector HEIs. USOS API [2] is a public API to the system. Mobile applications may use API to get access to information handled by USOS. However the standard PULL method does not scale well. Systems like Facebook, Twitter, Flickr, Foursquare base their notification services on a PUSH paradigm, where it is a server, not a client which starts communication. Clients may subscribe to be notified about the events of interest. Information providers send notifications in real time to a designated hub, from which they are further distributed to all subscribers. The similar architecture was built as part of USOS API. Clients may invoke methods of USOS API to subscribe for events which happen in USOS. Notifications are delivered by a special daemon.

The academic community of the biggest Polish HEI, University of Warsaw, consists of more than 50 thousand students and more than 3.5 thousand academic teachers. What might be the number of events users might want to observe? Grades are definitely among the most interesting objects handled by any SMIS. We counted the number of grade changes per month. In the most busy month, June 2013, there were 247,685 changes concerning 40,248 users. That means that for this one particular event type the system should be able to send to each client (mobile application) about 250 thousand notifications concerning about 40 thousand users in time shorter than one month. This is the efficiency we want to deliver. The notification system should be thoroughly tested against this non-functional requirement.

There is also another aspect of PUSH vs PULL method of information transfer. PUSH is not only more efficient, but also more natural for phone owners who are used to obtain SMSs without any activity on their side. This is the responsibility of the telecom service provider to locate the phone with the appropriate number and deliver SMS straight to it. System of notifications follows the same idea.

Last but not least there is a problem of data confidentiality which should also be taken care of. Nobody wants to jeopardize personal data, photos, grades, registrations, diplomas etc. stored in an academic information system.

The paper describes a system of notifications being part of USOS and — in particular — USOS API. It is based on the **PubSubHubbub** protocol, which allows to notify subscribers in real time about events originated from the student management information system. It uses OAuth protocol [4] to ensure confidentiality of data access. Solution is not only secure and user friendly, but also scales well. PubSubHubbub protocol, USOS API and OAuth are shortly described in Chapter 3. The subject of Chapter 4 are new methods of USOS API which implement the notification system, some technical details concerning security and scalability, system architecture and flow of information. Conclusions are drawn in Chapter 5.

3. PUBSUBHUBBUB, USOS API AND OAUTH PROTOCOL

3.1. PubSubHubbub protocol

According to Wikipedia [5], PubSubHubbub is an open protocol used by parties which want to communicate in a publish-subscribe manner. Its purpose is to provide real-time notifications of changes made to some resource, without requiring clients to spend time on polling for changes. A subscriber initially polls an HTTP resource in the conventional way, i.e. by requesting it from the web server. The subscriber then inspects the HTTP header, and if it references a hub, the subscriber can subscribe to that resource's topic on that hub. The subscriber needs to run a web accessible server so that hubs can directly notify it when any of its subscribed topics have been updated.

Publishers expose their content with the inclusion of hub references in the HTTP headers. They post notifications to these hubs whenever they publish something. Thus, when a publication event occurs, the publisher calls its hubs and the hubs call their subscribers. In order to provide a secure path, subscribers should share a secret with the hub, to be used by the hub to compute an HMAC signature that will be sent to the subscriber. The latter can then easily verify the origin by comparing the supplied signature with a similarly computed signature on its end.

PubSubHubbub is implemented by many content providers. Facebook developed *Realtime Updates* which is part of *Graph API*. A client chooses an object he wants to observe, indicates object attributes of interest, URL and a verification token. The system first verifies the subscription by making request to the given URL, with the verification token attached. Subscription is confirmed when the server answers properly to this request. The purpose of this verification is to prevent DDoS attacks by making sure that the client controls the server available under the given URL.

Then, whenever one of the indicated attributes changes the value, the notification system makes HTTP POST to the given URL. The request (in JSON format) does not contain new values of the attributes. It has an extra HTTP X-Hub-Signature header with the value sha1 equal to HMAC checksum of the request body calculated using SHA1 algorithm with the key being the shared secret known to API and the client (private key from the OAuth protocol). One request may contain many notifications.

It is of great importance that no sensitive data is sent in notifications. If the client wants to obtain the new values, a new request should be made. The full description of *Realtime Updates* may be found in [1].

Flickr, Foursquare, Instagram implement real time notifications in a similar way. Twitter delivers Streaming API — users connect to endpoints from which they read potentially infinite stream of data.

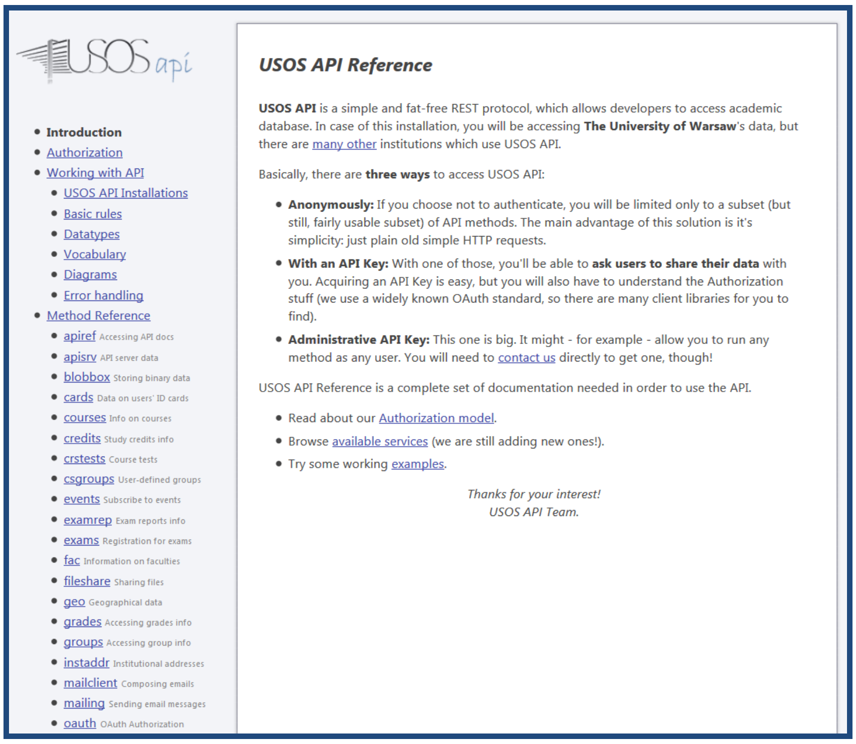
3.2. USOS API

USOS API is a standard REST-like interface to data gathered in USOS, publicly available, well documented, with guaranteed backward compatibility. Single USOS API installation consists of three functional parts:

1. Web services (API methods) available for applications.
2. Mini-portal for programmers, with public documentation (in English) of all methods. This portal is also used for generating keys used to identify applications with the server.

3. Administration panel for students and academic teachers. Users have access here to the list of applications, to which they have granted access to their USOS data.

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 (see section 3.3) 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). The main page of USOS API installation of the University of Warsaw is shown in Figure 1. It is available at http://usosapps.uw.edu.pl/developers/api/.



- **Introduction**
- Authorization
- Working with API
  - USOS API Installations
  - Basic rules
  - Datatypes
  - Vocabulary
  - Diagrams
  - Error handling
- Method Reference
  - apiref Accessing API docs
  - apisrv API server data
  - blobbox Storing binary data
  - cards Data on users' ID cards
  - courses Info on courses
  - credits Study credits info
  - crstests Course tests
  - csgroups User-defined groups
  - events Subscribe to events
  - examrep Exam reports info
  - exams Registration for exams
  - fac Information on faculties
  - fileshare Sharing files
  - geo Geographical data
  - grades Accessing grades info
  - groups Accessing group info
  - instaddr Institutional addresses
  - mailclient Composing emails
  - mailing Sending email messages
  - oauth OAuth Authorization

USOS API Reference

USOS API is a simple and fat-free REST protocol, which allows developers to access academic database. In case of this installation, you will be accessing **The University of Warsaw**'s data, but there are many other institutions which use USOS API.

Basically, there are **three ways** to access USOS API:

- **Anonymously:** If you choose not to authenticate, you will be limited only to a subset (but still, fairly usable subset) of API methods. The main advantage of this solution is it's simplicity: just plain old simple HTTP requests.
- **With an API Key:** With one of those, you'll be able to **ask users to share their data** with you. Acquiring an API Key is easy, but you will also have to understand the Authorization stuff (we use a widely known OAuth standard, so there are many client libraries for you to find).
- **Administrative API Key:** This one is big. It might - for example - allow you to run any method as any user. You will need to contact us directly to get one, though!

USOS API Reference is a complete set of documentation needed in order to use the API.

- Read about our Authorization model.
- Browse available services (we are still adding new ones!).
- Try some working examples.

Thanks for your interest!
USOS API Team.

Figure 1 Home Page of USOS API installation

3.3. OAuth protocol in USOS API

OAuth (*Open* standard for *Authorization*) is an open protocol for clients (usually applications) to access resources (such as confidential data) on behalf of a resource owner (usually end user). In a classical model of client-server authentication, the client uses its credentials (username and password) to gain access to its resources located on the server. In OAuth model, the client (which is not the owner of the resource, but only acts on his behalf) requests access to the resources which are controlled by their owner, but are located on the server. The client has to get permission from the owner first. It is expressed in the form of an access token. The purpose of the token is to avoid the situation when the owner has to share his password with the client. Unlike passwords, tokens may be issued with scope and time constraints (and cancelled at any time).

There is an excellent analogy given by Eran Hammer-Lahav (http://hueniverse.com/oauth/guide/):

*Many luxury cars come with a **valet key**. It is a special key you give the parking attendant and unlike your regular key, will only allow the car to be driven a short distance while blocking access to the trunk and the onboard cell phone. Regardless of the restrictions the valet key imposes, the idea is very clever. You give someone **limited access** to your car with a **special key**, while using **another key** to unlock everything else. [..] The decoupling of the resource owner's username and password from the access token is one of the most fundamental aspects of the OAuth architecture.*

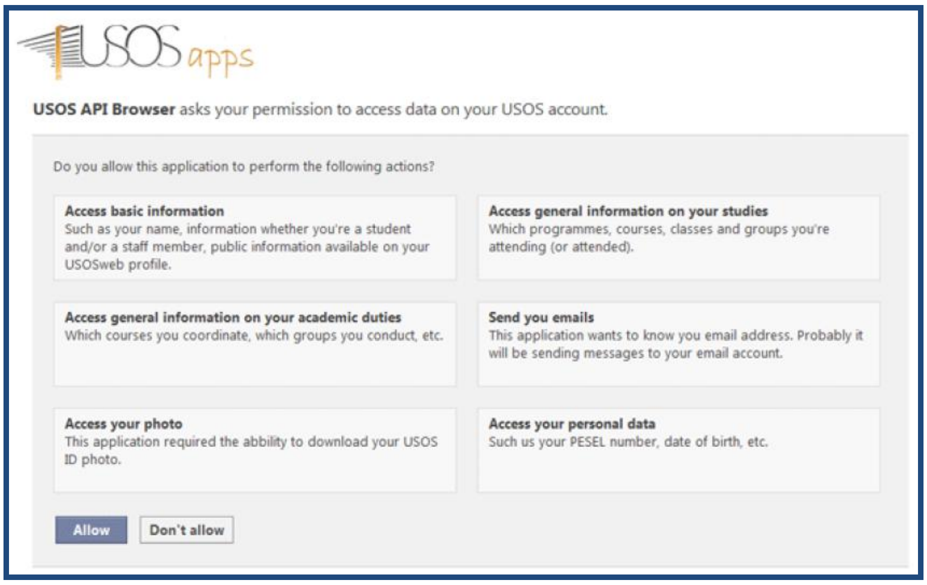
OAuth is used by Facebook, GitHub, Google, Microsoft, PayPal, Twitter, Yahoo!, Flickr, Foursquare, Instagram, LinkedIn, and many others.

Academic databases store private data (like citizen identity numbers, names, birth dates etc.), confidential data (like information about handicapped students or social aid), data of business value (like mailing lists of students and staff members), data which is vulnerable for theft or destruction (like grades or thesis reviews). USOS API opens the data repositories for public access while still holding them under control, protecting against misuse, violation of privacy, or destruction.

There are three ways to access data with USOS API methods:

- **Anonymously**: no authentication is required, gives access only to publicly available information, like courses or study programs.
- **With an API Key**: the permission of the user (data owner) is required to access data on his behalf (see Figure 2). This is implemented by the OAuth protocol.
- **With an administrative API Key**: gives full access to data of all users. It is not granted to external applications.

API keys are obtained/revoked from the USOSapps Administration Panel (the one for the University of Warsaw is available at https://usosapps.uw.edu.pl/developers/).


USOS apps

USOS API Browser asks your permission to access data on your USOS account.

Do you allow this application to perform the following actions?

Access basic information
Such as your name, information whether you're a student and/or a staff member, public information available on your USOSweb profile.

Access general information on your studies
Which programmes, courses, classes and groups you're attending (or attended).

Access general information on your academic duties
Which courses you coordinate, which groups you conduct, etc.

Send you emails
This application wants to know you email address. Probably it will be sending messages to your email account.

Access your photo
This application required the abbility to download your USOS ID photo.

Access your personal data
Such us your PESEL number, date of birth, etc.

Figure 2 Application asks resource owner (end user) for the permission to access data

4. IMPLEMENTING NOTIFICATIONS IN USOS API

There are two main problems to be solved when implementing the system of notifications as described in section 3.1: how to manage subscriptions and how to distribute notifications. In our implementation subscriptions are managed by USOS API, what means that the new API module had to be designed and implemented, with methods to subscribe/unsubscribe to particular events. Notifications may be distributed in two possible ways. First, they might be sent when the event occurs. This solution is simple but potentially absorbs many resources. The second possibility is to queue information about events and sent notifications later, grouping them by event type. The second solution was chosen. In standard PubSubHubbub protocol notifications are handled by the hub. In our case the hub is a special daemon which has access to USOS API tables where subscriptions and events are stored. The daemon periodically browses the tables for new records and distributes notifications to the clients. Subscriptions and notifications are described in section 4.1. Confidentiality and efficiency of the solution, which are of critical importance, are the subject of sections 4.2 and 4.3. Summary of the system architecture and implementation is given in section 4.4.

General overview of the system architecture is shown in Figure **3**.

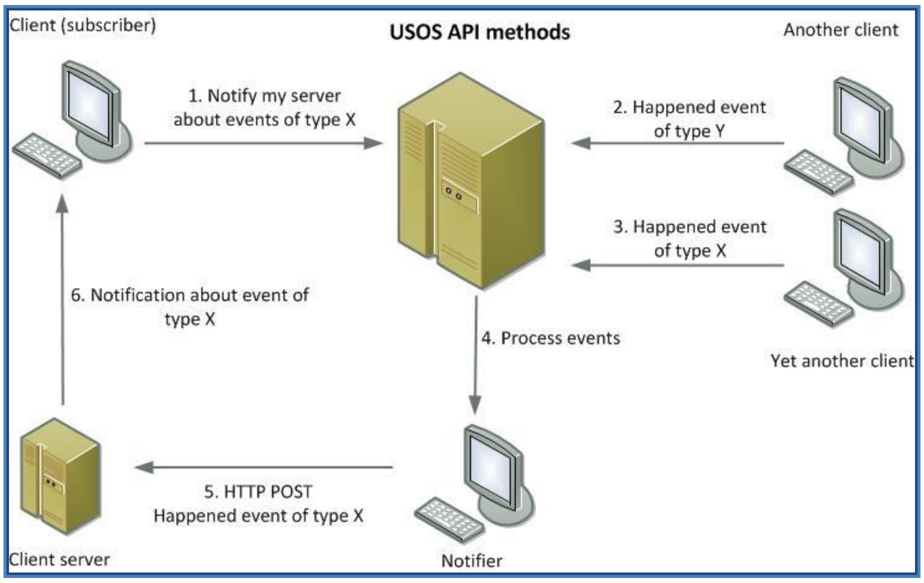

Client (subscriber)
USOS API methods
Another client
1. Notify my server about events of type X
2. Happened event of type Y
3. Happened event of type X
6. Notification about event of type X
4. Process events
Yet another client
5. HTTP POST Happened event of type X
Client server
Notifier

Figure 3 General overview of the system architecture

4.1. Events, event types, *services/events* module of USOS API

Here is an overview of the entire subscription and notification process.

1. Client (mobile application) chooses the type of events it wants to subscribe to.

   What are these abstract 'events'? Changes to data stored in USOS are made by invocations of API methods. Event 'occurs' when the method is invoked and changes data in the database. In USOS API, some of the methods allow to retrieve single entities — for instance, the *services/grades/grade* method returns a single grade. Such methods are named after entities they return. Event types are paths to such methods. A path to a method consists of module name and method name, thus *grades/grade* event type can be used to subscribe to changes related to grade entities.

2. Client subscribes to a given type of events through *subscribe_event* method.

   One client serves many end users and subscribes to events in their behalf. Aside from the event type, a client needs to provide a URL to the publicly available server — the endpoint that will receive notifications. Requests to this method have to be signed with the OAuth consumer key of the client. This is because subscriptions are created in context of the consumer (application). Typically, a subscription will be created by the application developer. It suffices to make one subscription for one event type per application — not one per application instance (user).

3. Client's request is validated to make sure that it is in fact under control of the server described by the URL delivered in the previous step. Providing the validation was successful, a new subscription is created.

4. Assume that an event of the type chosen in step 1 occurred.

   USOS API receives information about events from two kinds of sources: internal systems and event triggering methods. There is one such method for every event type. The naming convention for these methods is *{entity}_modified*. Example: s*ervices/grades/grade_ modified* method from *grades* module.

5. USOS API checks it the client is authorized to receive the notification (if needed).

6. The client's server receives an HTTP request with one or more notifications.

   The notifications are delivered to the server in the body of HTTP request. The request method is POST. The request body is in JSON format and contains a dictionary with two fields:

   a. *event_type*: a string being the event type of events that caused the notifications from this request,

   b. *entry*: a list of notifications in JSON format; one request can contain up to a thousand of them. As one can tell from the *event_type* description above, all notifications that come in a single request were caused by events of the same type, thus they have the same structure.

   The server should be able to handle requests from USOS API in a timely manner. Redirects are not allowed. There are no retries if something goes wrong. If the server frequently fails to respond to requests, the subscriptions may be removed.

The newly developed module *services/events* contains the following methods:

- *subscribe_event* — subscribe to events of a given type. Parameters are: *event_type*, *callback_url*, (optional) *verify_token*.
- *unsubscribe* — unsubscribe from events. Parameters (optional) are: *id*, *event_type*, *callback_url*.
- *subscriptions* — list the consumer subscriptions. Parameters are: *id*, *event_type*, *callback_url*.
- *notifier_status* — get information on the status of the notification daemon. Parameters are: *daemon_running*, *total_pending_events_count*.

USOS API stores subscriptions in the local table *api_events_subscriptions* (columns: *id*, *consumer_key*, *event_type*, *callback_url*) — one subscription in one record. Events which have occurred and have not yet been processed are stored in the table *api_events* (columns: *id*, *type*, *scopes*, *os_ids*, *timestamp*, *operation*, *event_json*). They are selected and deleted from the table by the notification daemon. The third important table is *api_oauth_tokens* for access tokens from the OAuth protocol (columns: *key*, *secret*, *token_type*, *longlived*, *timestamp*, *is_approved*, *os_id*, *consumer_key*, *scopes*, *verifier*, *callback*).

4.2. Security

There are two security requirements concerning the system of notifications:

1. protecting sensitive data,
2. ensuring authenticity of notifications received by subscribers.

To fulfill the first requirement we should address the following issues:

1. How to be sure that the subscriber has permission to obtain sensitive data?
2. How to avoid possible leaks when sending sensitive data?

The first issue is solved by the OAuth protocol. When the client asks the end user for the access token it declares the needed scope of privilege (e.g. the permission to read or update photos, grades, schedules, or other data). This information is recorded in a database along with the token (if the permission is granted). The client has to attach this token to the request to get the sensitive data. The access token is removed from the database when the end user logs out from CAS or after some predefined time — unless the client submits the long-lived token, which remains valid as long as the user does not revoke the permission. There is also a special type of permission — the client may possess an administrative key to a certain method of USOS API. In such case it does not have to possess any access token to invoke this method, what corresponds to a situation of having all tokens.

It should also be noted that client's privileges should be checked at the time of sending the notice, not at the moment of making a subscription, since the client may lose these privileges at any time (end user may revoke permission).

The problem of possible leaks of sensitive data is solved in a somewhat tricky way. Instead of sending sensitive data the notification daemon sends only an information about the event. This is kind of invitation for the client to get the updated, actual data by calling the method derived from the event type (with appropriate access token if needed), and this is already as safe as any other request to USOS API. Note that such solution has also been adopted by most of the systems mentioned in section 3.1.

The second security requirement is the ability to verify authenticity of notifications received by the subscriber. This problem is solved by the PubSubHubbub protocol. The HTTP request with the notification in its body contains an extra X-Hub-Signature header with a value of sha1 = {checksum}, where the checksum is an HMAC SHA1 signature of the request body. SHA1 key is shared by USOS API and the client, it is the consumer key from the OAuth protocol. Because the consumer key is known exclusively to USOS API and the client (consumer), if the HMAC signature calculated in the same way on the client side will be consistent with the value from the X-Hub-Signature header, the client will have the certainty that the sender is in fact USOS API.

4.3. Performance

The second most important non-functional requirement is system ability to handle many events in a short time. To have some point of reference we selected one possible event type — *grades/grade* — and counted the number of grade changes per month. In the most busy period (June 2013) there were 247,685 changes concerning 40,248 students. That means that for this one particular event type the system should be able to send to each client (mobile application) about 250 thousand notifications concerning about 40 thousand students in time shorter than one month. This is the efficiency we want to deliver.

The performance of the system was tuned very carefully, in a sequence of steps. The *cProfile* Python profiler was used to identify the most time consuming parts of the code.

The first trivial implementation was much below expectations: 60 events concerning 10 users and 3 clients were handled in 94 seconds.

The most consuming part of the code execution were HTTP requests. To reduce the overhead associated with execution of a large number of requests, we decided to group notifications. The idea is based on a very simple observation that there are as many different addresses to which notifications should be sent as there are subscriptions. So instead of sending notifications immediately, they are grouped by URLs and notification requests are made only after processing all collected events. This change in implementation reduced the overhead ten times (to 3.3 seconds). We enlarged the number of users and events — the time needed to handle 3 clients, 1,000 end-users and 6,000 events was 189.7 seconds.

As the next improvement we applied data caching which reduced the time to about 30 seconds.

Then we replaced ORM (the system is implemented in Django) used in the original implementation with direct SQL statements — the overhead was further reduced to 18.8 seconds. We again enlarged the test data — the time needed to handle 3 clients, 10,000 users and 60,000 events was 180.8 seconds.

The next improvement was more sophisticated. The most popular way to deal with a large number of concurrent tasks is currently to operate them in a **single thread**, but with the use of **asynchronous input-output** (this is also known as **non-blocking I/O**). From the possible frameworks which offer non-blocking I/O (we considered Twisted, Tornado, Node.js) we have chosen Tornado. The obtained time for this implementation was 92.5 seconds.

In the next step we gave up objects serialization and deserialization (which consumed too much time) and decided to replace in the database the serialized objects with the strings in JSON format — this reduced the overhead to 51 seconds.

Some extra tuning of database operations gave further improvements — 3 clients, 40,000 end-users and 250,000 events were handled in 77 seconds.

Table 1 contains the results obtained for all test data sets and implementations.

Subsequent amendments to the trivial implementation led to implementation in which the notification daemon is able to process 250,000 events within a little more than one minute. It is much better than the threshold of one month acceptable for the event *grades/grade*. It can therefore be safely assumed that the daemon may also cope with other types of events in a reasonable time.

Test data sets	*Trivial*	*Grouping*	*Caching*	*Direct SQL*	*Asynch. I/O*	*JSON*	*Extra tuning*
3 clients, 10 users, 60 events	93.2	3.3	1.8	1.6	0.64	0.59	0.56
3 clients, 100 users, 600 events		19.6	3.7	2.5	1.59	0.99	0.72
3 clients, 1 000 users, 6 000 events		189.7	30.1	18.2	9.85	5.53	2.46
3 clients, 10 000 users, 60 000 events				180.8	92.6	51.1	17.62
3 clients, 40 000 users, 250 000 events						209.7	77

Table 1 Test results for various data sets and implementation methods (times are given in seconds)

4.4. System architecture

Let us summarize the system architecture and elements of the final implementation:

1. A client makes subscriptions by invoking *services/events/subscribe_event* and provides a web server.
2. Notifications are HTTP requests to that server.
3. The system consists of new methods of USOS API for handling subscriptions and the notification daemon (which plays a role of the hub from the PubSubHubbub protocol).
4. Subscriptions are established in the context of the client, i.e. one subscription may cover notifications on all data to which the client has access.
5. Notification daemon wakes up periodically and processes events collected since last check.
6. Sensitive data are not transmitted; the notification only signals the changes and includes information necessary to fetch them.
7. Before sending notification about changes in a sensitive data of some user, permissions have to be checked; if the client has the OAuth access token for that user with sufficient privileges, the user can receive notification.
8. HTTP requests with notifications have an additional header — X-Hub-Signature, which allows to verify the sender.
9. Types of events are paths to methods returning a single record from the database; thanks to this, subscriber knows how to get the changed data. They are documented precisely in USOS API on-line documentation.

Final architecture of the system and information flow are demonstrated on Figure 4. It includes the flowchart for the notification daemon with implementation improvements described in section 4.3.

5. CONCLUSIONS

New methods of USOS API and the notification daemon comprise a basic infrastructure necessary to develop efficient, user friendly mobile applications for interacting with the student management information system. These applications have yet to be designed and implemented, what may be the task for a software company or individual developers who would like to take up the challenge. USOS API was designed with programmers in mind, yet this is their responsibility to best suit the needs of students and academic teachers.

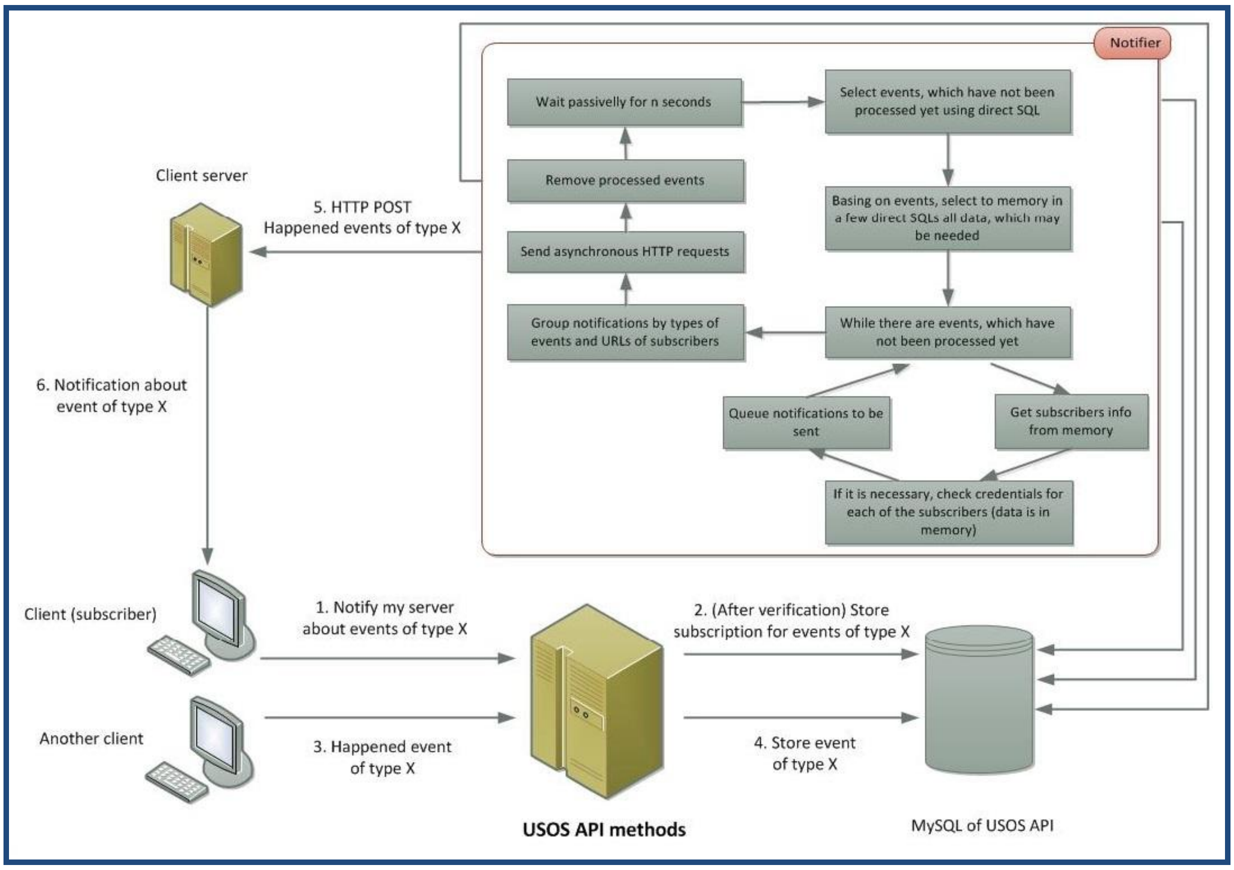

Notifier
Wait passivelly for n seconds
Select events, which have not been processed yet using direct SQL
Remove processed events
Basing on events, select to memory in a few direct SQLs all data, which may be needed
Send asynchronous HTTP requests
Group notifications by types of events and URLs of subscribers
While there are events, which have not been processed yet
Queue notifications to be sent
Get subscribers info from memory
If it is necessary, check credentials for each of the subscribers (data is in memory)
Client server
5. HTTP POST Happened events of type X
6. Notification about event of type X
Client (subscriber)
1. Notify my server about events of type X
2. (After verification) Store subscription for events of type X
Another client
3. Happened event of type X
4. Store event of type X
USOS API methods
MySQL of USOS API

Figure 4 System architecture with the algorithm performed by the notification daemon

6. Acknowledgements

This paper is based on the Master thesis of Kamil Szarek [6], supervised by Janina Mincer-Daszkiewicz. All programming work was done by Kamil.

7. REFERENCES

[1] Facebook *Realtime Updates*, documentation. Retrieved in January 2014 from: http://developers.facebook.com/docs/reference/api/realtime/ .

[2] 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.

[3] MUCI consortium. Retrieved in January 2014 from: http://muci.edu.pl.

[4] OAuth Community Site. Retrieved in January 2014 from: http://oauth.net/.

[5] PubSubHubbub protocol. Retrieved in January 2014 from: http://en.wikipedia.org/wiki/PubSubHubbub.

[6] Szarek, K. (2013). *Event notifications system for USOS API users*. Master thesis, University of Warsaw (in Polish).

[7] USOS website. *University Study Oriented System*. Retrieved in January 2014 from: http://usos.edu.pl.

8. 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.