Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 313–316

ISBN 978-83-60810-22-4

NotX
Service Oriented Multi-platform Notification System

Filip Nguyen, Jaroslav Škrabálek
Faculty of Informatics
Masaryk University
Brno, Czech Republic
Email: nguyen.filip@mail.muni.cz,skrabalek@fi.muni.cz

***Abstract*—This report describes NotX—service oriented system, that applies ideas of CEP and SOA to build highly reusable, flexible, both platform and protocol independent solution. Service oriented system NotX is capable of notifying users of external information system via various engines; currently: SMS engine, voice synthesizer (call engine) and mail engine. Adaptable design decision makes it possible to easily extend NotX with interesting capabilities. The engines are added as plug-ins written in Java. There are plans to further extend NotX with following engines: Facebook engine, Twitter engine, content management system engine. Also the design of NotX allows to notify users in theirs own language with full localization support which is necessary to bring value in today's market. Most importantly, the core design of NotX allows to run under heavy load comprising thousands of requests for notification per second via various protocols (currently Thrift, Web Services, Java Client). Thus NotX is designed to be used by state of the art Enterprise Applications that require by default certain properties of theirs external systems as scalability, reliability and fail-over.**

***Index Terms*—information system, soa, notx, cep, sms, voice, phone, mail, notification, enterprise, java, active mq, jms, jee, j2ee**

I. Introduction

NOTIFICATIONS have been studied as valuable tool in context of *ubiquitous computing* [3] and little more simplistic version of them (email notifications) are present in almost every *information system* as a standard approach to notify (and prompt) user in the case of password change, registration approval or account state change. But the real power of notifications come when there is more sophisticated business logic associated with generation of these events such as in [4]. Other useful applications of such a notification service are areas where traditional paper based communication/notification means are used [5]. Consider simple example—in information system dedicated to organize academic conferences important criterion for usability would be for user to receive notifications about paper submission and paper approval or rejection. They would also expect to be notified about other more real-time events like reschedule of certain presentation. This kind of business logic is usually system-specific but means of delivering these notifications are usually the same: email or SMS (short message service for cellular network). There is one additional channel that we find very useful (also indicated in [1]) and that is voice channel, namely text to speech synthesis delivered into cellular network. Because of repetitive use of this notification infrastructure (e.g. [6]) it would be beneficial to create service that would provide all these notification means.

In this report we describe service that complies to above description—NotX. In the first part of the paper we describe business requirements that are relevant for such service. Then the actual architecture and technological details of NotX are presented. The last part of paper is dedicated to discussing the development process used to drive NotX development and possible directions of further work on NotX.

II. Business requirements

NotX's first deployment hence first real use case is to serve as a notification service to Takeplace [11] information system to send various notifications including:

- emails with password change/registration
- rescheduling of presentation (this is typically delivered by SMS or voice)

Notification is sent dynamically via appropriate engine according to user settings and global NotX settings. Voice and SMS engines are a very fast way to notify the user but uses of these engines are charged so their use is limited and must be controlled.

Voice notifications can be delivered into cellular network or to SIP (Session Initiation Protocol - signalization protocol for internet phone communication). Motivation for SIP can be found in [7].

Because it is anticipated that the use of notifications will be massive and certain groups of users will be repetitively notified (for example attendees of certain conference) we demand *tagging* of users. Information system developer (IS developer) should be able to send notifications to either specific user or to specific tag.

Required operations to be performed via NotX are:

- tag (userid, tag)
- unTag (userid, tag)
- sendNotification (dest, msgType, templateName, placeholderVals)

The *tag* parameter in *tag* and *unTag* is text with '.' characters permitted, e.g. *ConfernceA.attendee* or *ConferenceA.speaker*. The first part of *tag* parameter up to '.' is called *domain*. The tag does not have to include the domain, the domain is

used only for billing and statistical purposes. The *userid* is unique identifier of user to be tagged. The *sendNotification* operation is used to send notification itself. The parameter *dest* is used to specify to which entity the message should be sent. It can be either userid prefixed with ':' or it can be tag. When tag is used in this parameter the message will be sent to all tagged users. Parameter *msgType* is used to add more semantic to message, e.g.: *important*. Administrator of NotX can use this semantic parameter to configure NotX to send all *important* messages via predefined engine, e.g for TTS (text to speech synthesis). Next parameter *templateName* is used to specify which message should be sent, for example template for registration approval *registration_approval* is template of message that is sent when registration process is successful. Lastly *placeholderVals* is associative array that is used to inject values into template.

Takeplace itself is a distributed web application, and has many different developers that are experienced in various technologies (ranging from PHP to Servlets) hence every one of these developers is used to a different way of accessing services. NotX should take this into account and make it as easy as possible to access NotX service.

Next important requirement is concerned with internationalization. Because academical conferences are usually attended by participants from various countries it is convenient for them to receive notifications in their own language. This is important equally for voice, sms and email notifications.

Because NotX uses charged services like cellular network the NotX has to keep track of sent notifications with information about domain to which they were sent.

Regarding nonfunctional requirements, the most important is to handle peaks of notifications with persistent fail-over. Usually if there is one big notification for all participants of major conference there can be thousands of various messages sent via email, SMS or voice synthesizer (TTS). It's not necessary to deliver all notifications at once but system should not render unresponsive or shouldn't crash and all messages should be delivered.

III. Architecture

NotX is developed to be a scalable platform and protocol independent Service. Currently the NotX is deployed to serve as a service for Takeplace so thousands of messages can arrive per second at peak hours.

Necessary attribute of reusable software service is its platform independence. That's why NotX and its components are built using Java programming language. NotX itself is web application that is built using build tool Maven [2].

The main output of the build process are two WAR (Web Application Archive) files: Notx.war and Communication.war which is web application that exposes protocols used as an interface into NotX. These main components are depicted in Figure 1. These WAR archives correspond to main components of NotX architecture - the core logic itself (NotX) and communication module (the Communication.war).

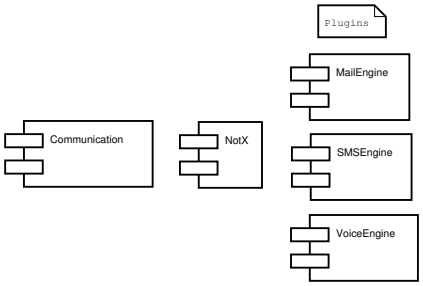

Fig. 1. Components of NotX

JMS (JSR 914) is specification for messaging API between loosely coupled components of information system. We are using Apache Active MQ implementation which supports persistent fail-over. Considering fail-over there are several fails that can happen during NotX's lifecycle:

1) Problem with external engine provider (SMS or voice)
2) Problem with connectivity to communication module with NotX
3) Bug in NotX logic
4) Any fail of hardware while processing notification

To address all of these problems our architecture is queue centric as seen in overall design in figure 2. After receiving request for notification the communication module immediately sends the request into persistent queue.

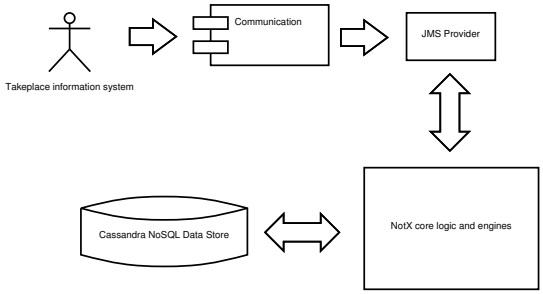

Fig. 2. Overall design

The most important operation of NotX is *sendNotification*. We will describe core logic behind this operation in more detail. As noted in business requirements this operation takes four parameters: *dest*, *msgType*, *templateName*, *placeholderVals*. Important logic takes place when *sendNotification* is called and destination is set to some specific tag, for example *ConferenceA.attendees*. Following steps take place after request has arrived to communication module:

1) Communication module recognizes the request as notification request and puts new notification request message A into message queue
2) NotX logic starts processing A by looking up N users which are to be notified by notification in A. Then NotX generates N messages $\{A_1, ..., A_N\}$ and puts them back into message queue.

3) Note that up to this point there was no interaction with any engine. Now NotX will be continuously receiving messages $A_x \in \{A_1, ..., A_N\}$ from message queue and each such message is processed as follows:
   a) Find language L of user for whom A_x is dedicated.
   b) Look up template for A_x according to L
   c) Now NotX injects *placeholderVals* into template and uses selected engine to notify the user. If this whole process is successful then notification statistic is saved into data store.

If there would be any kind of problem with data store or connectivity the messages are kept in message queue for administrator to manually decide how to deal with them.

Step number two is very important. The generation of $A_1, .., A_N$ messages helps to more evenly distribute load on the system and also helps traceability of the system. For example when notification request is to be processed for *ConferenceA.attendees* that can mean notification of 1000 users. When even 1 notification fails it is beneficial to know which one failed and why. After bug fixing it's important to be able to swiftly retry sending exactly the same notification as failed previously.

A. Protocol access

Requirement for protocol access may occur in many contexts. NotX provides following interfaces (as depicted in Figure 3):

- JSON (JavaScript Object Notation) interface via HTTP POST for simple notification sending
- Thrift interface for higher level languages (framework for cross-language service development)
- native Java libraries
- web services

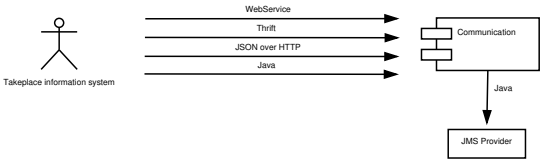

Fig. 3. Protocols

Adding new communication protocol is fairly easy. It means modifying Communication module, which resides in directory *src/notx-communication*. When adding new protocol, it is necessary to implement all NotX methods. Each method implementation usually just creates standard JMS message and puts it into MQ. Then only modification of *Communication-Main* class will make sure that after launch of communication module the interface into NotX will be functional.

B. Data storage

NotX uses data store for:

- information about users - contacts and tags
- statistics
- fail-over

Each user has his contacts stored in data store. This way NotX is able to send notifications by any engine for this particular user.

Statistics are saved mainly to charge users of paid services and performance tinkering.

Data storage is also used as a fail-over mechanism. Whenever a notification message is not sent successfully it is saved into data store and can be viewed via web interface with exception that caused the failure. It's possible to send specific failed notifications back to the message queue to retry the sending.

The data store itself is implemented using Cassandra NoSQL database. Decision to choose this storage type was led by need for multi-platform and highly scalable data store.

IV. Development process

The development process of NotX was driven by SCRUM methodology. This agile process introduced by Schwaber and Shuterland [8] suits development of the NotX best because requirements were from start more about searching of possibilities instead of launching repeatable processes.

SCRUM itself is being used with two week sprints (sprint is one iteration in SCRUM). Each sprint starts with sprint planning, where spring goal is presented (major functionality, or tangible goal that is to be produced by this sprint) and product backlog items for this sprint are presented (product backlog item is high level business requirement). SCRUM itself doesn't give many hints how to specify backlog items, but there are publications addressing this issue e. g. [9] which introduce *user stories* into SCRUM.

Then development proceeds and at the end of the sprint *sprint review* takes place where output of an iteration is presented. In NotX settings the sprint review and sprint planning took place same day, usually on Wednesday.

Product backlog as well as sprint backlog are kept in Open Office spreadsheet. This low tech approach always yields less administration and more focus on actual work. From backlog it can be derived how much work was spent on specific product backlog item each day and how well estimated the task was.

To our knowledge there are no major modifications of SCRUM methodology for web development (also in [10] no consistent difference was reported in decision making for web projects). There are however some subtle differences when developing system such as NotX in general (not just with agile practices):

- External tools spike first
- Sprint review should contain technological details
- Sprints to refactor code has to be more explicitly specified
- More focus on automatized integration testing

It's essential that each external tool like TTS system or SMS gateway that are used to carry out notifications are spiked first before adding any product backlog items that are dependent on this TTS. We recommend to have a sprint in which external tools are examined. Such a sprint helps in planning next

sprints because developers of NotX can help product owner to prioritize and estimate product backlog items that will include external tool usage.

Sprint review should include technological details because product owner represents technologically experienced users (developers of information system).

Sprints to refactor code has to be specified very explicitly with carefully formulated sprint goal. Sprint goals of these sprints shouldn't be vague or not measurable like: create more readable code. But there should be measurable goals e. g.:

- write automated test that will fire up in-memory database and perform CRUD (Create Read Update Delete) operations
- rewrite logic of configuration loading and present this new design using class diagram and sequence diagram at sprint review

Focus on integration testing comes from the fact that NotX itself uses several external systems and lot of logic is simple orchestration between JMS provider and external engines. This makes unit testing less effective.

All points above can be addressed with SCRUM by managing content of product backlog and sprint reviews.

V. Further work

In future, we plan to extend NotX to be publicly available service to be used by any IS developer. Technically it is possible right now because NotX supports lot of protocols for communication. There are, however, some missing functionalities like IS developer registration or billing reports.

Last important way to add more functionality to NotX is extending its communication module. There are many possibilities:

- Facebook engine—engine that notifies directly into accounts wall or private message
- Twitter engine—sends the notification to twitter
- FTP/SCP engine—puts the notification on FTP server or via SCP on some server
- IRC engine
- Skype engine—calling by skype. We didn't tested feasibility of this option yet.

The design of NotX, especially fact that it stores user information is preparation of NotX to become fully publicly available service that won't disclose users contacts. By managing contacts NotX is also single point where user can change his contacts and single point in which user can cut off potential notifications from various sources.

VI. Conclusion

In this paper we reported state of NotX - Service with capability of sending notifications via various engines. NotX gives value added to information system developers by taking burden of setting up infrastructure to send SMS, voice and email notifications. Additionally NotX helps with contacts management as it stores the contact information about users and doesn't reveal those contacts to IS developer.

NotX reduces time to integrate interesting functionality for any new information system with low development time and bring out of the box governance capabilities like failover, statistics and large scale notification sending in various languages. While doing all this NotX doesn't disclose any information about the user except his identifier that will be used to send notification.

Acknowledgment

The authors would like to thank Pavol Grešša for refining architecture of NotX and also to Lukáš Rychnovský for ideas from CEP and experience with building large scale distributed application that he shared.

References

[1] Kyuchang Kang, Jeunwoo Lee and Hoon Choi, "Instant Notification Service for Ubiquitous Personal Care in Healthcare Application" in *International Conference on Convergence Information Technology* 2007 pp. 1500-1503

[2] Apache Maven Project http://maven.apache.org/

[3] Schmandt, C. and Marmasse, N. and Marti, S. and Sawhney, N. and Wheeler, S. "Everywhere Messaging" in *IBM Syst. J.*, vol. 39, issue 3-4, July 2000, p. 660-670

[4] J. Jeng and Y. Drissi, "PENS: A Predictive Event Notification System for e-Commerce Environment" in *The Twenty-Fourth Annual International Computer Software and Applications Conference*, October 2000.

[5] Chi Po Cheong; Chatwin, C.; Young, R.; "An SOA-based diseases notification system" in *Information, Communications and Signal Processing, 2009. ICICS 2009. 7th International Conference* on, vol., no., pp.1-4, 8-10 Dec. 2009 doi: 10.1109/ICICS.2009.5397519

[6] Mohamed, Nader Al-Jaroodi, Jameela Jawhar, Imad A generic notification system for Internet information in *Information Reuse and Integration, 2008. IRI 2008. IEEE International*

[7] A. Sadat, G. Sorwar, M. U. Chowdhury, "Session Initiation Protocol (SIP) based Event Notification System Architecture for Telemedicine Applications" in *1st IEEE/ACIS International Workshop on Component- Based Software Engineering, Software Architecture and Reuse (ICISCOMSAR' 06)*, pp. 214-218, July 2006.

[8] Ken Schwaber, Mike Beedle Agile *Software Development with Scrum* Prentice Hall, 2001

[9] Mike Cohn *User Stories Applied For Agile Software Development* Addison-Wesley, 2010 ISBN:0-321-20568-5

[10] Carmen Zannier and Frank Maurer Foundations of Agile Decision Making from Agile Mentors and Developers in *Extreme Programming and Agile Processes in Software Engineering*, June 2006, LNCS 4044, p. 11–20

[11] http://www.takeplace.eu/en