Computer Science • 17 (4) 2016 http://dx.doi.org/10.7494/csci.2016.17.4.519

Piotr Nawrocki
Mikołaj Jakubowski
Tomasz Godzik

NOTIFICATION METHODS IN WIRELESS SYSTEMS

Abstract *Recently, there has been an increasing need for secure, efficient, and simple notification methods for wireless systems. Such systems are meant to provide users with precise tools best suited for work or leisure environments, and a lot of effort has been put into creating a multitude of applications. At the same time, however, not much research has been made into determining which of the available protocols are best suited for each individual task. A number of basic notification methods are presented here, and tests have been performed for the most-promising ones. An attempt has been made to determine which of the methods have the best throughput, latency, security, and other characteristics. A comprehensive comparison is provided, which can be used to select the right method for each individual project. Finally, conclusions are provided, and the results from all of the tests conducted are discussed.*

Keywords notification methods, wireless systems, energy consumption

Citation Computer Science 17 (4) 2016: 519–539

1. Introduction

The purpose of this paper is to analyze and test several selected notification methods for wireless platforms. This paper is an expanded version of a paper [6] presented at the Federated Conference on Computer Science and Information Systems, Lodz, Poland, 2015. The reason for this research is the need to determine the best way of sending simple as well as more-advanced messages about the events involved in the operation of grid systems or telemetric networks. This makes it possible to use the optimal approach in numerous projects that need to inform users about their current status. This aspect is currently of the utmost importance for the industry, as such notification methods enable developers to engage users much more deeply and keep them in constant contact with their leisure and work interests. These considerations have guided us throughout our research and affected all of our decisions on the selection and ways of testing the methods in question.

Several protocols and methods were considered based on their purposes and current industry standards. The main candidates were:

- CoAP (Constraint Application Protocol);
- Modbus;
- XMPP (Extensible Messaging and Presence Protocol);
- XMPP over SOAP (Simple Object Access Protocol);
- MQTT (Message Queuing Telemetry Transport);
- MQTT-SN (Message Queuing Telemetry Transport for Sensor Networks);
- AMQP (Advanced Message Queuing Protocol);
- Cloud notification systems (Google Cloud Messaging, Urban Airship);
- SMS (Short Message Service);
- Restful HTTP (Hypertext Transfer Protocol);
- SMQ (Simple Message Queries);
- ADM (Amazon Device Messaging);
- SIMPLE (Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions);
- STOMP (Simple (or Streaming) Text Oriented Message Protocol);
- DDS (Data Distribution Service for Real-Time Systems);
- AllJoyn.

Of course, these are not all of the protocols that could have been used for wireless notifications, but those listed appear to hold the most promise; therefore, the purpose is to discern their usefulness in the best way possible.

In addition to the protocols and methods listed above, other solutions were investigated, (such as the Apple push notification or Line application that, for various reasons, were not considered further). The Apple push notification technology is a useful technology, but it is proprietary (i.e., limited to Apple devices), so that is why it was decided to test more-universal solutions first. There are also solutions (applications) that use their own protocols; a good example is the Line application, which uses a proprietary protocol. Testing this solution was considered; however, there are significant difficulties with accessing the documentation for this protocol.

2. Related Work

Wireless systems are a relatively new field of study, and searching a specific topic such as comparing available notification methods does not return many related work results. Some of the protocols have been covered in separate articles; while these took the sending of notifications into account, tests were not always conducted in wireless systems.

The one available article [4] that compared notification methods only covered cloud systems [7] and applications. It discussed the following methods: C2DM (Google Cloud to Device Messaging the predecessor to GCM), Xtify, XMPP, and Urban Airship. As that article was relatively new at the time of our research, one might think that the information contained there would still be relevant, but it turned out to already be outdated. Meanwhile, Google has redesigned and rebranded its notification system, and Xtify was purchased by IBM. Only Urban Airship is still on the market in the same configuration as before. The article is more a comparison of available commercial products than a real-world testing suite. As expected, our conclusion was that the fastest protocol of the four tested was XMPP, but it had a characteristic that differed slightly from the others.

Another article [13] only tested the MQTT protocol. The authors believed that it was the best-possible choice and only aimed to describe its main features and capabilities. Only a single simple test as well as its averaged results together with the amount of data transferred and power consumption over a period of time were provided. In conclusion, the authors described the MQTT protocol as being both lightweight and perfect for mobile platforms.

In [11], the authors investigated XMPP in the field of collaborative applications. Its main purpose was to assess the usefulness of XMPP in exchanging location data between mobile clients and web servers. No testing was conducted, but a thorough description of XMPP and the Android platform was provided, while also taking into account the ways of integrating them. The article described XMPP as a general-purpose messaging protocol that is easily extensible.

In [16], the authors present measures to embed the Modbus protocol into a ZigBee stack. They monitor the real-time information from the ZigBee wireless sensor network and use some instructions to control the remote device in a friendly interface.

Important aspects in the context of notification methods are SLA parameters [5] and the power consumption of battery-powered devices. In [12], the authors discussed the problem of sending notification data using GPRS connectivity from remote telemetry stations [1]. They proposed the concept of adaptive message aggregation that extends the MQTT-SN protocol, adjusting its behavior to the GPRS (General Packet Radio Service) connectivity profile in order to decrease the energy consumption related to data transmission.

3. Notification methods

The following section generally describes and analyses the possible notification methods for wireless devices mentioned in the introduction. As a result of this analysis, it was decided to select some of them in order to perform the thorough tests later described in this paper.

3.1. SMS

It is possible to use the Short Message Service as a notification mechanism. An application would have to intercept the SMS messages received by an Android phone and analyze them to check whether they contain notifications from the system. One could just use simple text messages without a dedicated client application, but this would severely limit the functionality available to users.

This approach has several major issues. First, the cost of sending multiple messages to numerous clients could be immense. Second, it is not guaranteed that the message will be delivered on time or (sometimes) even the same day. What is more, all text messages have a maximum undelivered period (which cannot exceed 7 days), and this means that some notifications would not be delivered at all.

3.2. Google Cloud Messaging

In order to simplify the development of applications and to extend phone battery life, Google has created a simple built-in notification system for the Android platform, which only maintains a single connection at any time.

This approach has some obvious drawbacks. First, the number of messages sent concurrently is limited to four per application, and there is no guarantee that the message will be delivered (especially while the service is shared). Secondly, there is no specified maximum delay, which is not acceptable for most modern systems. Moreover, in posts like [8], it is claimed that the method is not all that well documented, and it is not easy to make an application work reliably with Google Cloud Messaging (GCM). Another problem with GCM is that some people do not trust Google to not abuse its capabilities, citing privacy or security concerns. One must also keep in mind that GCM can be used by some malware applications, as described in [2].

3.3. Restful HTTP

Another possible solution would be to use a RESTful HTTP service based on a pull queue model [3]. Such an implementation would have to pull notifications from the server at certain intervals or when the user turns on the application. Currently, creating such a service is a very simple process and does not require additional knowledge from most developers (which is the main advantage of this approach).

However, using this method is very inefficient, as it is not clear at what intervals such requests should be made. Using too long of an interval between requests may result in multiple notifications being sent all at once, making the older messages meaningless. Conversely, if the intervals were too short, it would use too much of the device's resources. Moreover, much of the workload is shifted to the wireless device, and the amount of data sent between server and client is sometimes doubled.

Some ideas for REST notification systems are discussed in [15]; however, using a pure REST approach is highly discouraged. Using the AMQP/REST mixed approach seems much more plausible.

3.4. Modbus

The Modbus protocol was developed in the 1970s by Modicon, Incorporated. It was created for use with its programmable logic controllers; but since then, it has become one of the industry standards for connecting industrial electronic devices. It is the most-mature protocol among all of the solutions selected, and it is the most lightweight as well.

Modbus works in the master-slave paradigm. There is a single master that can send queries consisting of sequences of actions to perform. Each action is interpreted by one of the slaves (sensor, disc, etc.) and can be either a data read or write. Data comes in two formats: coils and registers. A coil is a simple boolean value, and a register has the size of 16 bits. If a larger value is needed, it has to be split between multiple coils or registers. Each coil and each register has its own address, and it is up to both communicating parties to understand what is stored in each. Please note that, if both master and slave use the same technology, there are no problems with endianness, as values on each side will be written and read in the same order.

It is important to note that Modbus is an application-level protocol and only describes a messaging structure. It supports multiple underlying protocols such as TCP/IP or serial; however, the former is of most interest for this article, as it is most-common and battle-tested. Moreover, the TCP/IP protocol lines up perfectly with the scope of this article.

Requests from the master are sent in data frames whose sizes and fields vary from one Modbus implementation to another. In TCP/IP, each data packet contains a data frame that has the MBAP Header and Modbus TCP/IP Protocol Data Unit fields. The Protocol Data Unit contains a set of instructions to be performed. The MBAP Header is the exact ID of the device that the request is addressed to, as many slaves can reside on the same IP address.

3.5. XMPP

XMPP is basically an open technology for real-time communication, using XML (Extensible Markup Language) as the base format for exchanging information. It was designed to be easily extensible, and one of its main uses are publish-subscribe systems. Throughout its history, it was used by companies such as Google (in the Google Talk communicator), by Microsoft (in Skype), and by Facebook (in WhatsApp Messenger).

The idea behind XMPP is similar to that of e-mail, with a distributed server network in which each and every server can create its own service. The XMPP standard enables message encryption, and XML support allows for the use of such technologies as SOAP or EDI (Electronic Data Interchange).

A standard that is tightly coupled with XMPP is SOAP over XMPP, which can be tested using the same means, as sending a SOAP message is basically sending some content over XMPP. This standard provides effective and reliable messaging – both asynchronous and synchronous.

XMPP is a general-purpose protocol that is easily extensible. It was only designed to meet mobile platform requirements and was not expected to outperform any other protocols. However, its flexibility makes it a choice worth considering. In [11], a few add-ons are mentioned, like group chats or streaming services with a possibility to transfer files.

3.6. SOAP over HTTP

SOAP is a lightweight protocol for message exchanges that is independent from the system platform programming language. Its specification does not define a specific transport layer protocol, but most implementations use HTTP. It is important to mention, however, that HTTP is of no use for asynchronous messaging; because of this, SMTP is often used instead. The protocol makes it possible to send many short messages.

In the discussion on the use of SOAP in notification systems, the following solutions should be considered: polling, both endpoints having their SOA interfaces, using WS-notification, and using the message-queueing solution encapsulated in HTTP.

All of the solutions above have been analyzed, and none of them are easy to adapt to the needs of wireless notification systems. The first solution requires the client to make requests at certain points in time, which generates a lot of unnecessary traffic and is quite resource-heavy on small devices. The second idea is better but would not work for most wireless devices, as not all requests would pass from the server to the device (because such HTTP requests are often blocked). A good solution is to use WS-notification, but the problem with making requests from the server is still present. What is more, it is not a standard supported by all web servers. The final solution uses queueing, but it involves a lot of unnecessary technology, especially given that there are ready-made queueing mechanisms that do not have to be encapsulated in HTTP requests.

3.7. MQTT/MQTT-SN

MQTT is a publish-subscribe lightweight messaging protocol based on TCP/IP. It was designed to be open, simple, lightweight, and easy to implement, since it was intended to be used in constrained environments with limitations such as expense, low bandwidth, unreliable network, limited processor, or memory resources.

The entire protocol is based upon a central message broker, which distributes messages published on a topic to all of the interested consumers. The "MQ" part of the name comes from "Message Queueing"; however, this protocol does not support queuing by default. It has three types of quality of service for message delivery: "At most once", "At least once", and "Exactly once". It also has a mechanism that can be used to inform interested parties about an abnormal disconnection using the "Testament" and "Last Will" features.

What is interesting is the fact that MQTT has already been used in numerous applications. The first implementation of GCM(C2D)[1] used precisely this protocol. DeltaRail's latest version of their IECC (Integrated Electronic Control Center) also uses MQTT for communications within its signaling system (which is covered in [14]).

This standard was created by IBM; because of this fact, the IBM MQTT client Java library was used for testing, and the Mosquitto open source message broker was utilized for distributing messages. Mosquitto's simple construction allowed us to create a bash script, sending a set number of messages. The Android client connects to the broker using the IBM library and is fed the messages sent by the script.

MQTT-SN is a variation of MQTT designed to be used in sensor networks. In particular, it is supposed to be lightweight and easily implementable on small devices (e.g., in non-TCP/IP[2] networks).

3.8. CoAP

CoAP[3]. is a specialized web transfer protocol for use with constrained nodes and networks based on UDP (User Datagram Protocol). Its main task is to allow for communication between small devices such as sensors, switches, etc. It was designed on the basis of HTTP in order to simplify its architecture and allow for multicast. It also provides simple mapping between CoAP and HTTP, which can be used to create RESTful services. The messages are sent in a binary format; their size is limited by the maximum size of a datagram. Messages can be sent with acknowledgements or without (depending on the designer's needs). Although it is a relatively new standard, it already has some additional features proposed like "Observable", which makes it possible to notify all subscribed clients about changes to the resource.

3.9. AMQP

AMQP is an open-standard application layer protocol for message-oriented middleware that uses a binary format to send its messages. It was designed to solve the problem of interoperability between heterogeneous systems and message brokers. It was first used in 2006 by JP Morgan. It offers both point-to-point and publish-subscribe messaging types.

[1] Android Developer Central – GCM Advanced Topic – `http://developer.android.com/google/gcm/adv.html`.

[2] Transmission Control Protocol/Internet Protocol.

[3] CoAP RFC 7252 – `http://tools.ietf.org/html/rfc7252`.

The most important advantage of AMQP is the fact that it is independent of programming languages and platforms (unlike most messaging standards); for example, JMS (Java Message Service) [9]. Moreover, it offers several types of quality of service in terms of delivery guarantees; these types are "at-most-once", "at-least-once", or "exactly-once" guarantees. It also allows for the encryption of messages, which is especially important in the case of valuable scientific data. Currently, it is a widely used standard and has a large number of implementing libraries, like Apache Qpid, RabbitMQ [10], or StormMQ.

3.10. SMQ

SMQ (Simple Message Queries) is a proprietary Real Time Logic protocol that embraces the publish/subscribe broadcast design pattern. The main field for which it is designed is the IoT (Internet of Things). Its main advantages are said to be that the requirement for the dynamic creation of topic names and secure design has been eliminated. Moreover, it is said to be lightweight and fast, which is really important for the resource-constrained devices of the IoT. For transferring messages between the browser client and the broker, SMQ uses an http/https connection, which is then morphed into an HTTP WebSocket. However, for the exchange of data between devices and the broker, the HTTP/HTTPS connection is morphed into a persistent TCP connection. The main difference between SMQ and such protocols as MQTT and AMQP is that the SimpleMQ broker translates each topic name into a randomly created 32-bit number. Real Time Logic provides client libraries for ANSI-C to us in devices (SMQ client and SharkMQ client) and JavaScript for browsers (SMQ.js). The protocol supports security features like SSL/TLS.

The protocol was ruled out from our comparison because it has no Java support (in contrast to most other notification methods). Obviously, Android has native code support (which may allow to use SMQ); but, to date, no such attempts have been made (as far as we know).

3.11. ADM

Another protocol that can be used for wireless device notifications is ADM (Amazon Device Messaging) which allows to send push notifications to all Amazon devices and to the applications they are running. Its creators describe it as a simple and efficient protocol for general use. It uses OAuth 2.0 to verify whether a server can send a notification to a client; when the message is passed to this device, it is encrypted with SSL. It is important to note that ADM is just a simple transport mechanism and cannot transform data in any way. All communication must be effected using JSON objects. One great advantage of ADM is the ability to wake the device when delivering a message; however, it does not make any guarantees about delivery or order of messages.

While the Amazon Device Messaging API is a really simple and effective solution, it is restricted by its lack of QoS as well as the fact that it is restricted almost exclusively to Kindle Fire devices (which are not the most popular wireless devices). Because of these considerations, this notification method was not selected for testing in this comparison.

3.12. SIMPLE

SIMPLE (Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions) is a set of extensions for the Session Initiation Protocol (which is used to set up, initiate, and manage media sessions). SIMPLE defines additional SIP methods to handle data transport. Similar to XMPP, it is an open standard and has a very broad range of uses. The main advantage of this solution is said to be the unification of voice, video, and data messaging. It can as easily be used as a notification method, as it allows users to register for presence events and receive notifications when they occur. SIMPLE uses XML to serialize data and provides encryption for its messages.

While SIMPLE provides much of the needed functionality for notification methods, it is a much-more-complex solution than necessary. In addition to the above facts, it is very difficult to find any materials (except for the RFC document and a brief mention on Wikipedia). Ultimately, because of these two considerations, the SIMPLE protocol was not included in our comparison.

3.13. STOMP

STOMP (Simple [or Streaming] Text Oriented Message Protocol) is a very simple text-based protocol for messaging middleware. It provides an interoperable wire format to allow any client to communicate with any STOMP broker independent of the programming language or system platform. It was designed to be as simple as possible, and in many instances, it is very similar to HTTP. In contrast to AMQP or MQTT, it only covers a small subset of commonly used messaging operations instead of providing a complete solution. A basic STOMP frame consists of a command, a set of optional headers, and an optional body. All messages are sent to destinations on STOMP servers. What those destinations are is dependent on the exact implementation; however, they most commonly correspond to a topic, an exchange, or a queue. It uses UTF-8 encoding by default, but it can also be used to carry binary data.

As this protocol is very frequently used in the same way as AMQP or MQTT, it was decided to not include it in the comparison of notification methods. Perhaps when further work is necessary, its tests will be also included; however, it is currently beyond the scope of this paper.

3.14. DDS

DDS (Data Distribution Service for Real-Time Systems) is a middleware specification created by the Object Management Group that defines an Application Level Interface and the behavior of a Data-Distribution Service (DDS) that supports Data-Centric Publish-Subscribe (DCPS) in real-time systems. Its purpose is to enable scalable, real-time, dependable high-performance and interoperable data exchanges between publishers and subscribers. This data flow is regulated by QoS contracts established between both DataWriters and DataReaders. Among the more-important features of the DDS specification is the presence of a built-in discovery service that makes it possible to dynamically discover the existence of both publishers and subscribers (which, in turn, simplifies complex network programming for distributed applications. As DDS is not a protocol itself, the OMG suggests using the RTPS (Real-Time Publish Subscribe), which supports all of the requirements of the DDS specification. Among the available DDS libraries is OpenDDS, which is written in C++; however, Java and JMS bindings are also provided in order to allow it to be used in the JVM environment.

The main idea of the DDS specification was to assist the creation of large distributed applications that could interoperate independently of the platform or programming language used. And while this kind of assistance is very useful, the protocol appears somewhat complex to use in simple wireless notifications (especially since the DDS specification is intended for more than just wireless systems.

3.15. AllJoyn

AllJoyn is an open-source software framework used for communication between devices and applications in order to enable the Internet of Things. Each AllJoyn application can advertise its services using one of two mechanisms: About Annoucements and Well-Known Name. The former is the recommended mechanism for advertising, while the latter has more lower-level functionalities and does not provide as much metadata. The framework also takes care of creating sessions between different applications, which can be point-to-point or multi-point. The AllJoyn application has the option of accepting or denying remote connection requests. Moreover, its API supports all major platforms and several languages: Java, C/C++, and Objective-C.

The research presented in this article is closely related to the IoT, and AllJoyn appears to provide a comprehensive solution for managing multiple devices and applications. However, at the time of writing, AllJoyn was still not a widely accepted framework, and there were several issues with connections over the Internet.

4. Tests

There are currently three main mobile operating systems available on the market (Android, iOS, and Windows Mobile) as well as numerous devices that support them. As it would be neither possible nor sensible to test each and every one of them, only one testing platform and device was chosen.

Google's Android system was selected as the wireless platform for testing purposes because of the considerable availability and open nature of the solution. All major protocols and methods selected have working implementations for this system.

As a mobile device, the Nexus 5 (LG D821) was used with Android version 4.4.3 using the standard Dalvik engine. At this point, Android Runtime was already available, but it did not seem ready for serious testing as of yet.

In order to conduct all tests, a server platform was needed (which consisted of an Asus laptop with an Intel i5-3320M processor and 8GB of RAM with Ubuntu 12.04LTS and Oracle Java 1.7.0_60 installed). All data (from a server platform) was transmitted over the Wi-Fi network using the 802.11g standard (54 Mbps).

For time-related test cases, ClockSync[4]. was used to synchronize with time servers on the wireless device. All applications launched their message connectors in separate threads. Services were not used, so all memory-usage diagrams show the combined values of the connector and activity screen.

All useful notification methods should meet most of the specifications listed below:

- the financial cost should be low – mostly for open source and university projects;
- it should be possible to transmit more than just simple text – to enable interaction between the application and the main system;
- energy and memory usage should be minimal – the solution is to be used on wireless devices;
- message contents should be secure – confidential information could be transmitted;
- minimal message loss – important data could be transmitted;
- minimal delay – fast interaction is sometimes needed.

As a result of analyzing the available notification methods (see Section 3) and taking the above assumptions into account, it was decided to test the following protocols: Modbus, XMPP, SOAP, MQTT, CoAP, and AMQP.

In order to conduct testing for each protocol, the following solutions were used:

- XMPP – in order to prepare the XMPP test, the Smack library was used to implement the mobile client. It has been ported to Android in a version called Asmack. ejabberd was used as a message broker. The second client, which sent messages to the mobile client, was implemented in Python using the SleekXMPP library. The mobile solution was plain and simple, with its task limited to keeping an open connection to the broker.
- SOAP – an attempt was made to test a basic polling mechanism using a simple Python SOAP server[5]. and a basic Android client[6]. This involved making a number of requests for stress testing and a single request to measure single-message performance. After obtaining the initial results, this method was discarded, as it was more than ten times slower than any other method and used a lot of resources for polling (which is unacceptable for wireless devices). It was decided to concentrate efforts on other solutions specifically intended for such devices. The results collected are shown along with the other protocols tested but are not

[4]ClockSync – `http://amip.tools-for.net/wiki/android/clocksync`.

[5]Python simple and lightweight SOAP Library (A.K.A. soap2py) – `https://code.google.com/p/pysimplesoap/wiki/SoapServer`.

[6]A simple SOAP client for Android – `https://code.google.com/p/droidsoapclient`.

included in graph comparisons (in Figure 1), as they were much worse than for any other test conducted.

- MQTT/MQTT-SN – a broker that works with the MQTT/MQTT-SN protocol is the RSMB (Really Small Message Broker) from IBM; and while it is quite easy to find, locating an appropriate client library (especially for MQTT-SN) is much more difficult. The one that is available for MQTT-SN[7]. is written in the C programming language, so it was of no use whatsoever for Android devices. The project also appeared to have been abandoned (no recent contributions). A further search led to a library written in Python[8], which was then used to implement a client that would be run using the QPython interpreter. After a certain amount of research and testing, a stage was reached where messages were delivered from the broker to the device (but never reliably). Attempts to change QoS settings failed, as it appeared that the client-library implementation was not complete as of yet. For these reasons, MQTT-SN was excluded from testing, and only MQTT was tested. It appears that the protocol is not yet mature enough to be used on a larger scale and that it has no reliable or finished implementations.
- CoAP – there are a limited number of implementations. The main Java libraries are jCoAP and nCoAP. jCoAP is not up to date with the RFC (Requests for Comments) 7252, so nCoAP was therefore used (which additionally implements the "Observable" feature). To test the protocol, a simple server was created with a time service and a mobile client that was sending GET requests to the server. At first, it was intended to use the "Observable" feature; but during stress testing, it turned out that messages cannot be sent too often using this implementation due to errors. Although "Observable" can be quite useful (especially in wireless systems), it was consequentially decided to have each notification sent as an answer to a separate GET request.
- AMQP – RabbitMQ was selected, which is one of the best-documented and most-popular libraries. For testing purposes, a simple Python script was developed that can send a set number of simple messages containing timestamps and an Android client application. The client connects to the RabbitMQ message broker, and the Python script is then used to send messages.
- Modbus – for testing purposes, the j2mod library was chosen (which is a still-maintained fork of the well-known jamod library). Both the master and slave were implemented using the same technology (although the server was written in Scala instead of Java). It is important to note that it is not just a one-way protocol in contrast to RabbitMQ, for instance. As an additional requirement, data had to be buffered in registers in order to make sure that nothing was lost and the response was correct.

[7]MQTT-SN client in C – `https://github.com/njh/mqtt-sn-tools`.

[8]MQTT-SN client in Python – `http://git.eclipse.org/c/mosquitto/org.eclipse.mosquitto.rsmb.git/tree/rsmb/src/MQTTSClient/Python`.

To assess the efficiency and usefulness of the notification methods selected, several test cases were created and run for each protocol.

4.1. Time per message

Each of the six applications designed to check how much time it takes to process a single message while stress testing was used with different numbers of concurrent messages sent. Set sizes of 10, 50, 100, 200, 300, 400, and 500 messages were chosen. Each message contained its timestamp in order to enable the calculation of the exact delay. Figures 1 (nCoAP), 2 (MQTT), 3 (RabbitMQ), 4 (XMPP), 5 (SOAP over HTTP) and 6 (Modbus) show how much time it took to process a single message for different stress test set sizes.

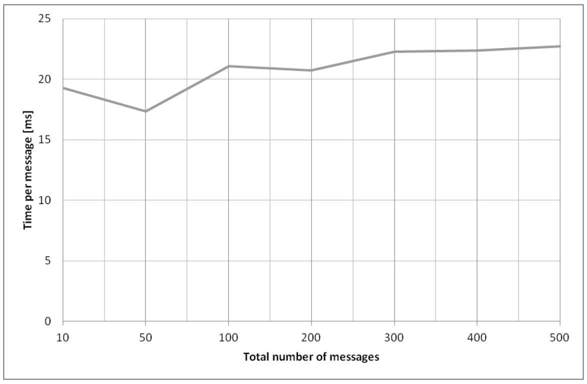

Figure 1. nCoAP

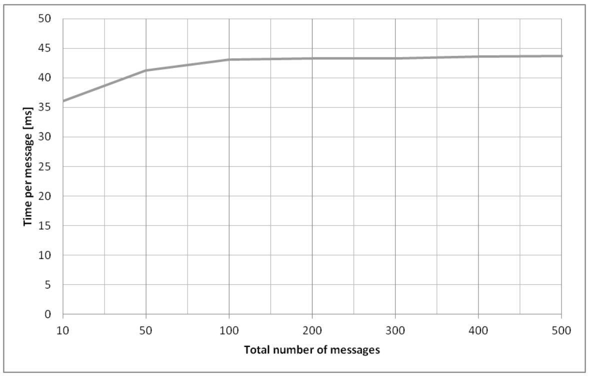

Figure 2. MQTT

Based on the graphs generated, it can be stated that Modbus is the fastest in terms of performance (probably because it is the simplest protocol in this compari-

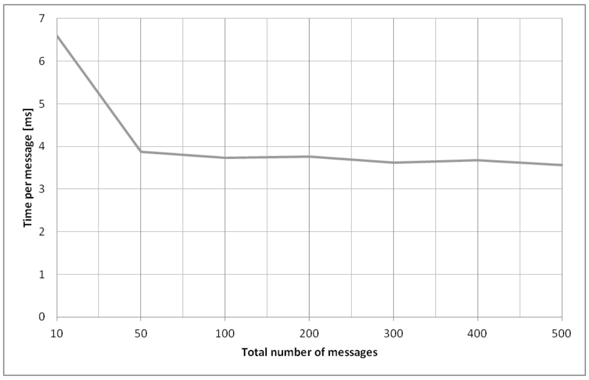

Figure 3. RabbitMQ

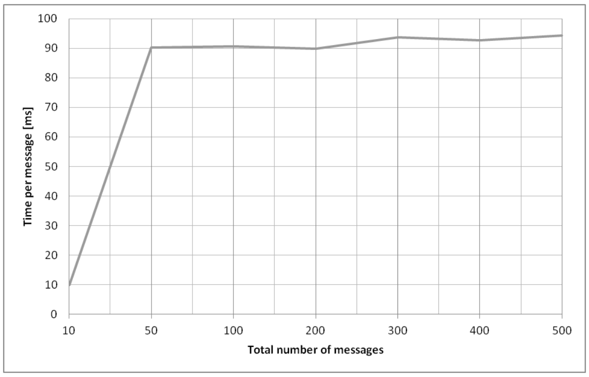

Figure 4. XMPP

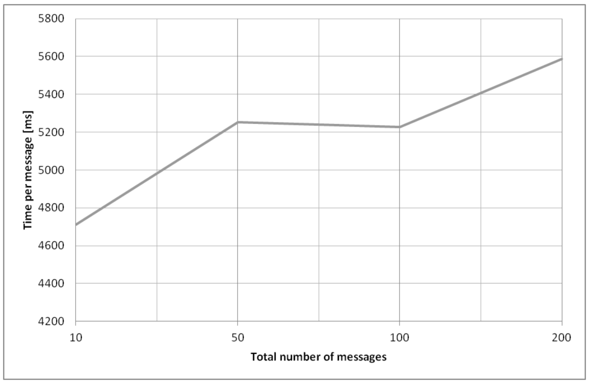

Figure 5. SOAP over HTTP

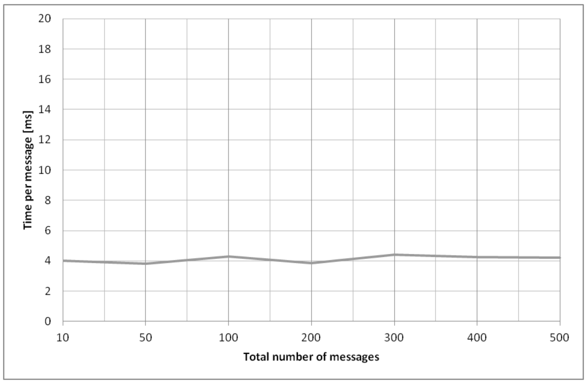

Figure 6. Modbus

son). Its results remain steady irrespective of the number of messages sent at once. RabbitMQ is similarly very fast, and its performance actually improves as more messages are sent at the same time; however, it does not wait for a response (in contrast to Modbus). The nCoAP solution was also quite effective; however, keep in mind that each message was sent in response to a GET request, so it could be faster still. Both MQTT and XMPP exhibit quite long sending times for larger numbers of messages. The SOAP over HTTP protocol is definitely the slowest solution. In this test, Modbus and RabbitMQ stand out as the leaders in the group.

4.2. Resource usage

After performance, the second-most-important criterion was resource usage. It is crucial to use as few device resources as possible on a wireless platform in order to consume less power and allow for greater efficiency. In this section, peak memory usage (shown in Table 1 – "RAM (Random Access Memory) usage peak") and CPU (Central Processing Unit) power consumption (by using PowerTutor tool [17]) were measured, as presented in Figures 7 (nCoAP), 8 (MQTT), 9 (RabbitMQ), 10 (XMPP), 11 (SOAP over HTTP) and 12 (Modbus), while sending 1000 messages concurrently to be processed by each of the mobile clients developed.

It is clearly visible that nearly all protocols used similar amounts of memory, with the outliers (MQTT and Modbus) exhibiting 10-MB-lower RAM usage than the others. It is clear that Modbus again outperforms the rest, probably because it is the simplest protocol of all compared.

A much larger difference can be seen in power consumption levels. These seem to be strongly correlated with each individual protocol's processing time. nCoAP and RabbitMQ consumed the least power; Modbus does not appear to fall far behind. About twice the power was consumed by XMPP and SOAP over HTTP. The worst result was achieved by the MQTT protocol.

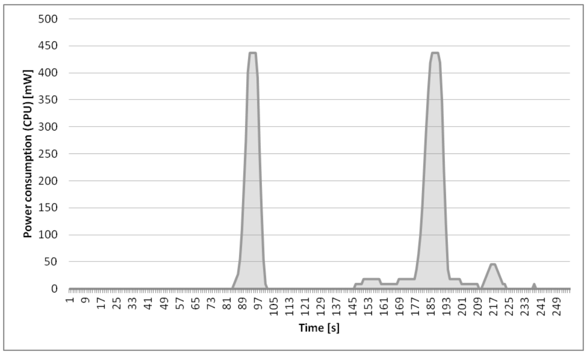

Figure 7. nCoAP (2 consecutive runs shown).

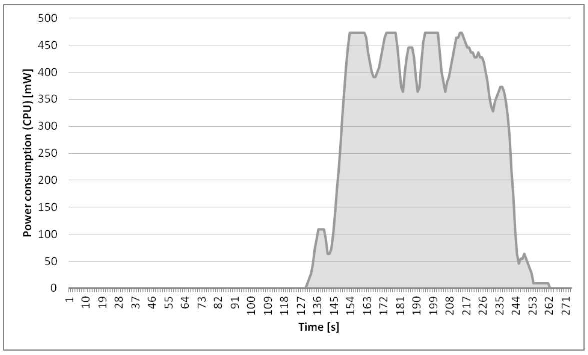

Figure 8. MQTT

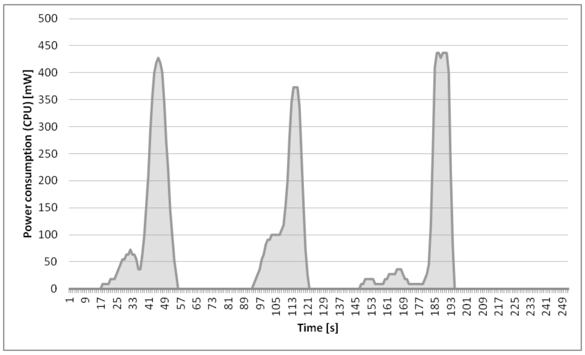

Figure 9. RabbitMQ (3 consecutive runs shown).

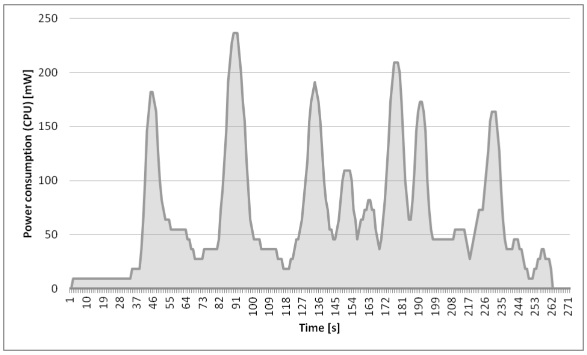

Figure 10. XMPP

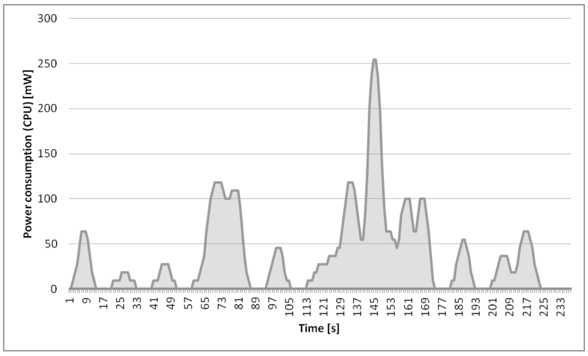

Figure 11. SOAP over HTTP

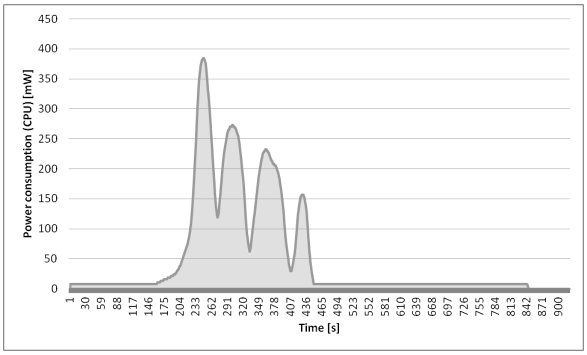

Figure 12. Modbus

4.3. Reliability

Message-sending reliability was also tested, as it is among the most important issues to be tackled in wireless notification applications. Especially important is the issue of what happens to messages in a queue when the connection to the client is lost. This was simulated by reconnecting to a Wi-Fi network while sending a set of 1000 messages. All protocols were tested using default settings. It turned out that nCoAP and Modbus were the only ones not to deliver all of their messages. These two do not provide any recovery mechanisms by default, because all messages are sent on a request-response basis. Developers must be sure to use the correct settings for each protocol, as QoS is not usually switched on by default.

4.4. Ordering

This test case was meant to show whether protocols deliver messages in the same order in which they are sent. Similar to the first test, a set of messages containing timestamps was sent, and the comparison of arrival times of successive messages made it possible to determine whether they were correctly ordered. Only nCoAP changed the order of messages, which is most probably caused by using UDP. All other protocols delivered messages in the correct order, even during high load.

4.5. Average delay

The final test case was used to calculate the average delay when sending a single isolated message using each of the five protocols (Table 1 – "Average delay"). It turns out that nCoAP is the fastest when it comes to sending individual messages, and SOAP over HTTP is the slowest among all of the protocols tested.

5. Conclusions

The results shown in Figure 13 clearly demonstrate that, in terms of the maximum number of messages delivered per second, RabbitMQ and Modbus are the leaders; however, when it comes to minimal delay, nCoAP tends to be able to deliver single messages much quicker. This means that, if large numbers of notifications are to be sent, RabbitMQ could be used; while in a sparse notification system, Modbus should perhaps be recommended. In the power consumption test, the best results were also achieved by RabbitMQ and nCoAP (with Modbus not far behind).

It should also be noted that not all protocols are able to easily pass through firewalls and NATs like XMPP (which is the most complete of all of the protocols tested). When it comes to RAM usage, MQTT turned out to require the least amount of megabytes, which can be of great importance on wireless devices. A summary of test results is shown in Table 1.

In conclusion, the most-promising solution seems to be RabbitMQ; however, none of the protocols proposed clearly outperformed all of the others in all test cases. Modbus seems to be as fast as RabbitMQ, but it does not provide any reliability of

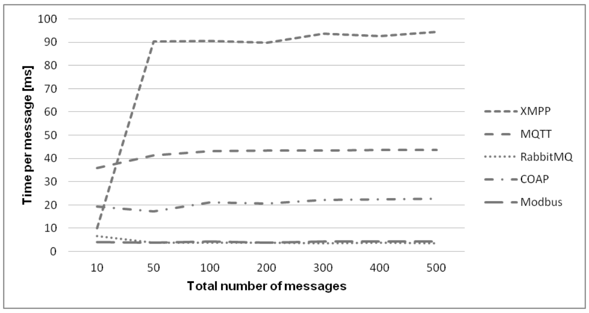

Figure 13. Comparison of protocols.

notification delivery (which is a huge drawback). This test suite only demonstrates the performance of some implementations currently available, and the results might change for future releases, different platforms, and/or different devices. Before any protocol is selected, it is important to specify the needs of the project in question and then compare the protocols to determine which one best suits these needs.

As this article only used the Android system, more research could be carried out using different platforms (such as Windows Mobile or iOS) in order to confirm how much the results depended on using a specific solution. Moreover, using other wireless standards like LTE would help to improve the quality and usefulness of research in regards to possible throughput and delay.

Table 1
Overview of protocol properties

	nCoAP	**RabbitMQ**	**MQTT**	**XMPP**	**SOAP**	**Modbus**
RAM usage peak [MB]	47	44	35	46	38	37
Average delay [ms]	91	185.5	339.6	192.3	972.2	308.1
Ordered	no	yes	yes	yes	yes	yes
Reliable	no	yes	yes	yes	yes	no
Content	binary	binary	binary	text	text	binary
Based on	UDP	TCP	TCP	TCP	TCP	TCP
SSL support	no	yes	yes	soon	yes	no

Acknowledgements

The research presented in this paper was partially supported by the National Center for Research and Development (NCBiR) under Grant No. PBS1/B9/18/2013 and by the Polish Ministry of Science and Higher Education under AGH University of Science and Technology Grant 11.11.230.124 (statutory project).

References

[1] Brzoza-Woch R., Konieczny M., Kwolek B., Nawrocki P., Szydlo T., Zielinski K.: Holistic Approach to Urgent Computing for Flood Decision Support. *Procedia Computer Science*, vol. 51(0), pp. 2387–2396, 2015, ISSN 1877-0509, international Conference On Computational Science, ICCS 2015 Computational Science at the Gates of Nature.

[2] Duckett C.: Android malware utilising Google Cloud Messaging service. `http://www.zdnet.com/android-malware-utilising-google-cloud-messaging-service-7000019427/`, 2013.

[3] Ghinamo G., Vadala F., Corbi C., Bettassa P., Risso F., Sisto R.: Vehicle navigation service based on real-time traffic information: A RESTful NetAPI solution with long polling notification. *Ubiquitous Positioning, Indoor Navigation, and Location Based Service (UPINLBS), 2012*, pp. 1–8, 2012.

[4] Hansen J., Grønli T.M., Ghinea G.: Towards cloud to device push messaging on Android: Technologies, possibilities and challenges. *International Journal of Communications, Network and System Sciences*, vol. 5(12), pp. 839–849, 2012.

[5] Kosinski J., Nawrocki P., Radziszowski D., Zielinski K., Zielinski S., Przybylski G., Wnek P.: SLA Monitoring and Management Framework for Telecommunication Services. *Fourth International Conference on Networking and Services, ICNS 2008.*, pp. 170–175, 2008.

[6] Nawrocki P., Jakubowski M., Godzik T.: Analysis of notification methods with respect to mobile system characteristics. M. Ganzha, L. Maciaszek, P. M., eds., *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, *Annals of Computer Science and Information Systems*, vol. 5, pp. 1183–1189, IEEE, 2015, `http://dx.doi.org/10.15439/2015F6`.

[7] Nawrocki P., Soboń M.: Public cloud computing for Software as a Service platforms. *Computer Science*, vol. 15(1), 2014, ISSN 2300-7036, `http://journals.agh.edu.pl/csci/article/view/519`.

[8] Oldenburg R.: Keeping Google Cloud Messaging For Android Working Reliably [Technical Post]. `http://blog.pushbullet.com/2014/02/12/keeping-google-cloud-messaging-for-android-working-reliably-techincal-post`, 2014.

[9] Richards M.: Understanding the Difference Between AMQP and JMS. *NFJS Magazine*, 2011.

[10] Rostanski M., Grochla K., Seman A.: Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. M.P. M. Ganzha L. Maciaszek, ed., *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, *Annals of Computer Science and Information Systems*, vol. 2, pp. 879–884, IEEE, 2014, `http://dx.doi.org/10.15439/2014F48`.

[11] Schuster D., Koren I., Springer T., Hering D., Söllner B., Endler M., Schill A.: *Creating Applications for Real-Time Collaboration with XMPP and Android on*

Mobile Devices. Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications, IGI Global, 2012.

[12] Szydlo T., Nawrocki P., Brzoza-Woch R., Zielinski K.: Power aware MOM for telemetry-oriented applications using GPRS-enabled embedded devices – levee monitoring use case. M.P. M. Ganzha L. Maciaszek, ed., *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, *Annals of Computer Science and Information Systems*, vol. 2, pp. 1059–1064, IEEE, 2014, `http://dx.doi.org/10.15439/2014F252`.

[13] Tang K., Wang Y., Liu H., Sheng Y., Wang X., Wei Z.: Design and Implementation of Push Notification System Based on the MQTT Protocol. *2013 International Conference on Information Science and Computer Applications (ISCA 2013)*, Atlantis Press, 2013.

[14] Wood D., Robson D.: Message broker technology for flexible signalling control. *Proc. ASPECT 2012 Conference*, 2012.

[15] Wylie K.: REST Requires Asynchronous Notification. `http://kirkwylie.blogspot.com/2008/12/rest-requires-asynchronous-notification.html`, 2008.

[16] Yanfei L., Cheng W., Chengbo Y., Xiaojun Q.: Research on ZigBee Wireless Sensors Network Based on ModBus Protocol. *IFITA '09, International Forum on Information Technology and Applications*, vol. 1, pp. 487–490, 2009.

[17] Zhang L., Tiwana B., Qian Z., Wang Z., Dick R.P., Mao Z.M., Yang L.: Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pp. 105–114, ACM, New York, NY, USA, 2010, ISBN 978-1-60558-905-3, `http://doi.acm.org/10.1145/1878961.1878982`.

Affiliations

Piotr Nawrocki
AGH University of Science and Technology, `piotr.nawrocki@agh.edu.pl`

Mikołaj Jakubowski
AGH University of Science and Technology, `mkl.jakubowski@gmail.com`

Tomasz Godzik
AGH University of Science and Technology, `tomek.godzik@gmail.com`

Received: 14.02.2016
Revised: 4.05.2016
Accepted: 12.05.2016