Digital Communications and Networks xxx (xxxx) xxx

Contents lists available at ScienceDirect

Digital Communications and Networks

journal homepage: www.keaipublishing.com/dcan

On the performance of web services, google cloud messaging and firebase cloud messaging[☆]

Guido Albertengo, Fikru G. Debele, Waqar Hassan [*], Dario Stramandino

Department of Electronics and Telecommunications, Politecnico di Torino Corso Duca Degli Abruzzi, 24, Turin, 10129, Italy

A R T I C L E I N F O

Keywords:
Web Services
Google Cloud Messaging
Firebase Cloud Messaging
REST
SOAP

A B S T R A C T

Smartphones and other connected devices rely on data services, such as Web Services (WS), Google Cloud Messaging (GCM) and Firebase Cloud Messaging (FCM), to share the information they collect or use. Traditionally, these services were classified according to the average number of bytes transmitted or their delivery time. However, when dealing with battery-operated devices, another important parameter to be taken into account is their power consumption. Furthermore, software designers and developers often do not consider the efficiency of a data communication system, but are simply concerned about ease-of-use and response time. In this paper, we compare FCM, GCM and two types of WS, namely Simple Object Access Protocol (SOAP) and REpresentational State Transfer (REST) WS in terms of delay, data efficiency, and power consumption. The final outcome is that RESTful WS outperforms all others, making GCM and FCM a viable alternative only when the amount of data to be transmitted is very limited, or when the mobile application requires the advanced services offered by FCM or GCM only.

1. Introduction

Web Services (WS) [1], Google Cloud Messaging (GCM) [2] and Firebase Cloud Messaging (FCM) [3], which replaced GCM in 2016, are designed to exchange messages across the Internet. With respect to fixed hosts, where energy and data efficiency are minor concerns, in any mobile scenario available, energy is always limited and network data traffic is usually capped. In both scenarios, applications may even have time constraints to be fulfilled. Due to the constant increase of mobile users connected to the Internet, smartphone applications are rapidly replacing websites in the Consumer-to-Business (C2B) and even in the Business-to-Business (B2B) markets. Therefore, the selection of messaging technique can no more neglect how energy is used and must also take into account the message delivery time, which affects the application response time. To address this problem, we set up a test bed, using mobile devices to evaluate one-way delay, round trip time, data usage, packet overhead and battery duration for the above mentioned messaging techniques. The results we presented in the following sections can be used to assist application designers and developers in their technical choices.

Traditionally, websites and applications used WS to exchange data with servers according to the classical client-server communication scheme. This approach in general requires quite a lot of time to design the WS, implement and test it, and then implement and test the corresponding client. This increases the time to market and the development cost of the application. In terms of infrastructure, WS requires an always-on server with a public IP address, which can be either a dedicated computer or a virtual machine in the Internet cloud. Nowadays, this latter solution is usually preferred due to its higher reliability and lower overall cost.

Cloud-based messaging solutions, such as Google's GCM and FCM as well as Apple's Push notification [4] service, usually require fewer resources in terms of time, development cost and infrastructure with respect to WS. All of them provide a Software Development Kit (SDK) and Application Programming Interfaces (APIs) so that developers do not need to define low level programming logic. Moreover, since all of these services are cloud-based, mobile applications and data servers can exchange data directly, in a peer-to-peer fashion, without the need for any dedicated server to handle messages, as in the WS case. All of these factors make these services more reliable and usually cheaper than WS.

[☆] Note: The manuscript was extended from the conference proceeding “On the performance of Web Services and Google Cloud Messaging, Computer and Information Technology (CIT), 2017 Proceedings of 17th International Conference on (pp. 363–367). IEEE, Helsinki, Finland, August 2017.”

[*] Corresponding author.
E-mail address: waqar.hassan@polito.it (W. Hassan).

https://doi.org/10.1016/j.dcan.2019.02.002
Received 5 February 2018; Received in revised form 9 February 2019; Accepted 21 February 2019
Available online xxxx

Moreover, developers can use and test all features with comfortable usage quotas without paying any charges when using GCM or FCM only. As a conclusion, these services are no more expensive than WS in both development and production.

Therefore, from the application provider's perspective, there are no reasons to keep on using WS, but the situation could be very different from the customer's perspective. In fact, they are usually very sensitive about these issues:

1. How much battery does this application use?
2. Is this application fast and responsive?
3. How much data does it use?

This paper aims to answer these questions in order to help application developers choose the messaging technique which exploits the best trade-off between development cost and user's satisfaction. This is, to the best of our knowledge, the first attempt to compare the out-of-the-box performances offered by WS, GCM and FCM. Source code for all main classes needed for the tests conducted for this paper is publicly available at https://github.com/hassanwaqar00/ws-vs-gcm-vs-fcm.

2. Architectures and comparison

In this section, we briefly describe the operation and architectures of WS, GCM and FCM. All of them are designed for message exchanges, but unlike WS which follows the classic client-server communication paradigm, GCM and FCM force all messages to pass through servers (managed by Google), making them distinct in terms of architecture and, of course, out-of-the-box features. These dissimilarities should likely imply a difference in performance. On the other side, these same servers enable GCM and FCM to provide value-added services, such as push technology, data encryption and collapsible messages [5], which can not be directly provided by any WS. However, in order to compare these services on a fair base, we used a common test scenario composed of a mobile terminal and a data server. The mobile, using the above-mentioned services, sends a block of data to the server which in turns sends back the same data block.

2.1. WSs

WSs are programmatic interfaces for communication between software agents using the HyperText Transfer Protocol (HTTP). Software agents that provide services are referred to as *service providers*. They are responsible for publishing programmatic interfaces and description of the services to *discovery agencies*. The published services are discoverable and consumable over the web by other software agents called *service requesters*, who need to discover the published services from discovery agencies before they can start using them. Once services are discovered, service requesters interact with providers through message exchanging. Fig. 1 depicts the WS architecture modelling the interactions among the service provider, service discovery service, and service requester.

As far as the messages exchanged between the service requester and provider are concerned, different WS implementations use different protocols and message structures. The two well-known WS architectural styles named REpresentational State Transfer (REST) [7] and Simple Object Access Protocol (SOAP) [8] are currently used on the Internet.

REST is an architectural style that defines a set of recommendations for designing loosely coupled applications that use the HTTP protocol for data transmission. In its purest form, a RESTful WS uses HTTP methods (such as POST, GET, PUT and DELETE), is stateless to improve performance by saving bandwidth and minimising server-side application state caching, uses Uniform Resource Identifier (URI) to address resources, is data-driven and transfers data structures by serialising them in eXtensible Markup Language (XML) or JavaScript Object Notation (JSON) [9].

SOAP is a protocol defining strict rules for messaging and Remote Procedure Calls (RPCs) using XML format that uses any application layer

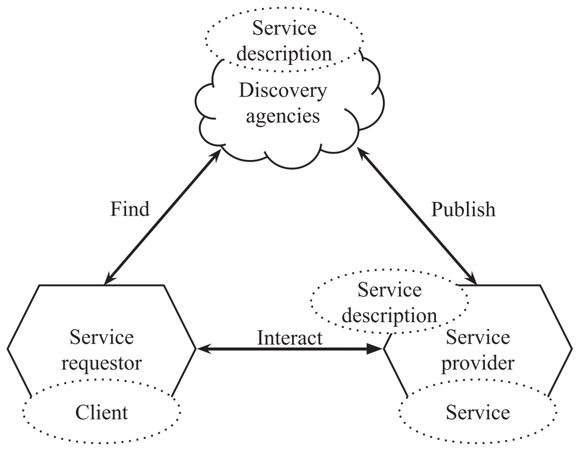

Fig. 1. WS message exchange pattern [6].

protocol and is usually coupled with Web Services Description Language (WSDL). In its purest form, it also uses HTTP methods, is stateless (but can be made stateful), function-driven and transfers data structures by serialising them only in XML [10].

2.2. GCM

GCM is a service provided and maintained by Google. Although not recommended for new applications, it allows developers to exchange messages among a single server and one or more client devices. GCM handles all aspects of messaging, including queuing, delivery and addressing. It is completely free to use for developers and supports client devices running Android, iOS and Chrome. Messages can be sent downstream (*i.e.,* from the server to clients) or upstream (*i.e.,* from a client to the server). Downstream messages can be distributed in three different ways:

- to a single client deviceFor
- to a group of devices
- to all devices that subscribed to a topic

Downstream messages are usually used to alert users, start a background process on the client's device or chat messaging. Upstream messages may be used as acknowledgements, pushing data or chat messaging. There are three fundamental components in the GCM architecture:

1. GCM connection server
2. Application server based on:
   (a) HTTP
   (b) eXtensible Messaging and Presence Protocol (XMPP)
3. Client application

The GCM architecture is presented in Fig. 2.

The *GCM connection server* receives messages from an application server and sends them to the client applications or vice versa. Google provides two types of connection servers, which differ from the messaging protocol they use, namely, HTTP and XMPP. They slightly differ in terms of features they support (refer to Table 1) and can be used separately or together at the same time. The XMPP-based server is commonly known as Cloud Connection Server (CCS). The *application server* communicates with the selected GCM connection server to send/receive data to/from the client application. It is responsible for communicating with GCM connection server using properly formatted requests, handling requests and exponential back-off (which is an error handling approach in which a client periodically retries a failed request with increasing delays between requests), storing API key and managing

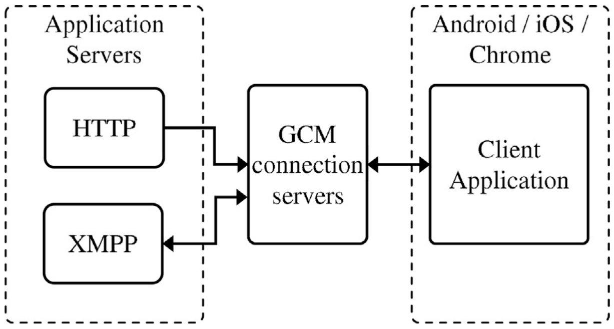

Fig. 2. GCM general architecture.

client registration tokens. The *client application* must register with GCM service to obtain a unique identifier called a registration token, which will be later on used to identify and authenticate the client when it sends messages to the GCM server.

2.3. FCM

FCM is a cross-platform solution for messages and notifications for Android, iOS, and web applications. FCM is provided and maintained by Firebase, a company now owned by Google [11]. FCM inherits GCM's core infrastructure but simplifies the client-side development (for example, developers no longer need to write their own registration or subscription retrying logic in the client application). Most of what has been said earlier about GCM remains true for FCM. The way FCM handles messages is very similar to GCM except in this case messages pass through FCM servers instead of GCM connection servers. Although GCM has not been deprecated by Google so far, developers are strongly encouraged to upgrade their applications to FCM.

The fundamental components in the FCM architecture are:

1. FCM connection server
2. Trusted environment with:
   (a) Application server based on:
      i. HTTP
      ii. XMPP
   (b) Cloud functions
3. Client application

The FCM architecture is presented in Fig. 3. FCM supports notification and data messages. Notification messages are automatically handled by the FCM SDK to show a notification on behalf of the client application. They contain a predefined set of user-visible keys (and an optional data payload of custom key-value pairs). Data messages, on the other hand, have only custom key-value pairs and are completely handled by the client application. FCM allows collapsible and non-collapsible message delivery. A non-collapsible message is a message that is always delivered to the device. A collapsible message is a message that may be replaced by a new message, which supersedes it, if it has yet to be delivered to a

Table 1
Comparison of features supported by CCS and HTTP-based GCM connection servers.

Features	CCS	HTTP
Upstream/Downstream messages	Upstream and Downstream	Downstream
Messaging (synchronous/ asynchronous)	Asynchronous	Synchronous
Message format	JSON encapsulated in XMPP messages	JSON or plain text sent as HTTP POST
Multicast downstream	Not supported	Supported in JSON message format

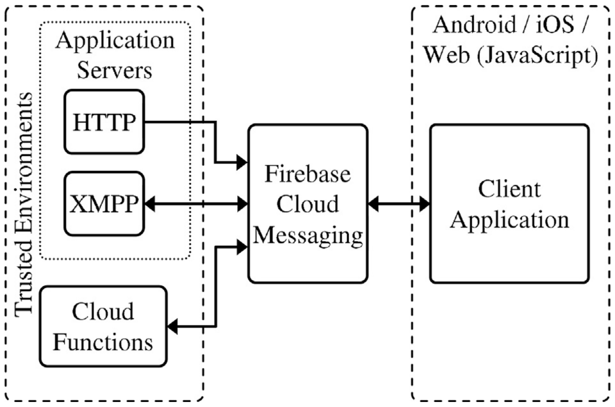

Fig. 3. FCM general architecture.

device [12].

2.4. Comparison

As explained in the previous sections, WS strongly differs from GCM and FCM in terms of architecture and out-of-the-box features. Value-added features such as encryption, push-based messaging, etc., are not intrinsically supported by WS (although they can be implemented by custom software). GCM and FCM offer encryption, push-based messaging, native Android and official iOS SDK support. In terms of architecture, GCM and FCM have a third entity between the client and the application server or trusted environment, which adds some delays in the communication path between a mobile terminal and an Internet-located server.

We make our comparison on the out-of-the-box performance of WS, GCM and FCM in a simple client server scenario, which is the only one where the additional services offered by GCM or FCM will not bias the choice of a developer. In more complex scenarios, the features only provided by GCM and FCM usually make them the only viable solutions. Finally, notice that the time delays due to the presence of servers in GCM and FCM are included in all of their time measurements.

3. Performance metrics

In order to analyse the performance of the above mentioned messaging techniques, we used the following metrics:

- *One-Way Delay (OWD)*: the time taken for a packet to be transmitted across a network from source to destination [13].
- *Round Trip Time (RTT)*: the time taken for a packet to be transmitted plus the time taken for an acknowledgement of that packet to be received [14].
- *Network Data Usage (NDU)*: the amount of data transmitted over the network.
- *Packet Over Head (POH)*: the data overhead due to protocols and serialisation. Computed as $packet\ overhead = packet\ size - payload\ length$ [15].
- *Battery Duration (BD)*: the number of hours a device's battery lasts while constantly using WS, GCM or FCM, all of them at the same messaging rate.

4. Methodology

To evaluate the performances, we implemented a test bed for each of the three communication systems. In all of them, the client was a smartphone, running an Android application, whereas the server hosted a Java application. The smartphone was directly connected using Wireless Fidelity (Wi-Fi) 802.11n to a router, which in turn was connected to the Internet through a 100Mbps Ethernet link. This solution was chosen to minimise the transmission delay and to be able to monitor the data packet traffic using a Wireshark [16] instance running on the router itself. Another Wireshark instance running on the server monitored the same data packet traffic on the other side of the communication path. The Wireshark traces were used to evaluate NDU and POH.

For the evaluation of OWD, the client and the server required a common time reference. We used the Network Time Protocol (NTP) to synchronise them. For OWD and RTT, we executed 10 tests per payload length where each test consisted of the transmission of 500 messages. NTP time synchronisation between the smartphone and the application server was achieved as follows:

1. The smartphone queried an NTP server (time.google.com) 100 times to get 100 time-offsets: their average was taken as the actual time-offset.
2. The smartphone requested the application server to perform time synchronisation with the same NTP server (time.google.com), again using 100 iterations and taking an average at the end.
3. The application server reported a successful time synchronisation to the smartphone.
4. The smartphone started the OWD test. After every 100 test messages (*i.e.,* time duration <1 min), the smartphone and application server re-synchronised the local time-offset (step 1) until all test messages were sent (500 messages per test with 10 tests).

For BD, four tests were performed with different payload lengths. BD tests collected a device's battery statistics every 3 mins until the battery drained completely. Notice that BD is the duration of the battery in a scenario where a device continuously uses the messaging technique. This allows to estimate the duration of the battery in an endurance test, which is a situation where the energy consumption of the smartphone's other applications or of its operating system is minimised. In a more realistic scenario, a device may use a messaging technique discontinuously or periodically with a long period. It is, however, realistic to assume that the ranking of messaging techniques achieved using BD will be valid also for discontinuous or periodic requests.

WS test beds always used HTTP POST method with XML serialisation for SOAP WS and JSON serialisation for RESTful WS. The testing scenarios for each message exchange technique are described in the following sections.

For OWD, RTT, NDU, and POH, tests were performed with payload lengths between 64 and 4055 bytes. This latter value was chosen to comply with a requirement of GCM and FCM that were designed to transfer a data message with maximum key/value pairs size up to 4096 bytes. During the tests, the maximum length of a value that we were able to bundle with a key (named ”PAYLOAD”) was 4055 bytes. Using a value size longer than that resulted in a GCM "Message Too Big" error [17,18]. To have conformity among different tests, therefore, we fixed the maximum payload length for all messaging systems we tested to 4055 bytes.

During all tests, there were no other network loads on the 100 Mbps link connecting our router to the Internet. All mobile devices run the official versions of Android with all available updates from the manufacturer and only factory default applications.

The WS, GCM and FCM servers were hosted on an Ubuntu 14.10 computer with Intel Core 2 Duo 2.56 GHz, 6 GB Random-Access Memory (RAM) and a 100 Mbps Ethernet link to the Internet. The operating system was completely up-to-date and during the course of the tests, the system was only used by the system or test related processes.

4.1. WS test bed

In order to evaluate the delay performance (*i.e.,* OWD and RTT), the smartphone sent a message every 500 ms, which contained a predefined payload to the server and recorded the current time. The server received

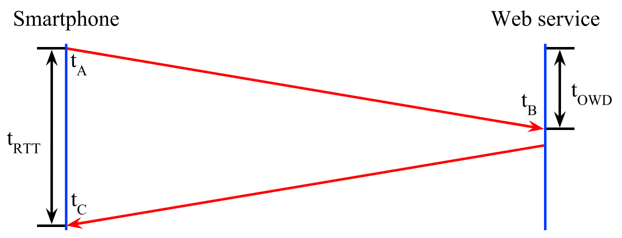

Fig. 4. Message delivery timeline for OWD and RTT evaluation in WS.

the message and recorded the current time for OWD computation. Then, it echoed the same message back to the smartphone which computed the RTT value upon reception, as shown in Fig. 4. A time interval of 500 ms was chosen since it was higher than the highest OWD value observed during preliminary tests, while still allowing to execute a single test in a reasonable time (*i.e.,* 250s).

The OWD and RTT were evaluated as $t_{OWD} = t_B - t_A$ and $t_{RTT} = t_C - t_A$, respectively, where all values are measured in ms. The steps to compute the NDU were as follows:

1. The smartphone sent a message of a predefined payload length, and the Wireshark instance running on the router traced the packet.
2. The server received the packet and the Wireshark instance traced it.
3. The server sent a packet with the same payload back to the smartphone and Wireshark traced it again at both server and client sides.
4. In the end, all Wireshark traces were examined to evaluate NDU and POH.

For BD, the scenario was similar to that of NDU, except that Wireshark was not required. The smartphone sent a message every 500 ms and logged the battery parameters (*i.e.,* battery charge percentage, voltage and temperature) every 3 mins. The consumption statistics collection continued until the smartphone battery completely discharged.

4.2. GCM test bed

The communication scenario of GCM was different from the previous one since for GCM there were three entities in the system. To evaluate the time performance, the smartphone sent a message, again every 500 ms, that contained a predefined payload and its GCM registration ID [19] to the CCS and recorded the current time instance. The CCS forwarded the message to the application server which recorded the reception time. The application server echoed, again through the CCS, the same message back to the smartphone which computed the RTT value upon reception, as shown in Fig. 5.

The delay and RTT for GCM were evaluated using the same formula as in the case of WS. For the evaluation of NDU, the scheme was very similar too, except that the message passed through the CCS before reaching the application server. For the estimation of BD, a slightly different mechanism was used, since the CCS disconnected the communication between the smartphone and the application server in a couple of hours (randomly between 2h and 4 h). Instead of sending periodic upstream messages (like in the case of WS), we sent periodic downstream messages. In detail, the

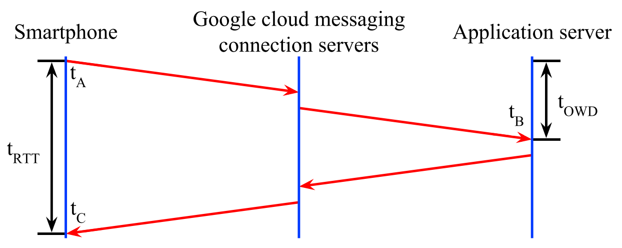

Fig. 5. Message delivery timeline for OWD and RTT evaluation in GCM.

approach we took to overcome this problem is as follows:

1. The smartphone sent an initiation message with its GCM registration ID to the CCS along with the identification number of the application server.
2. The CCS received the message and forwarded it to the application server.
3. The application server received the message and started sending messages, every 500 ms, with a predefined payload length to the smartphone.
4. The CCS received the messages and forwarded them to the destination smartphone.
5. The smartphone received the messages sent from the application server.
6. Periodically (every 3 mins, same as BD tests for WS), the battery parameters were saved locally and sent to the application server, where they were logged.

Steps 1–6 were repeated until the smartphone discharged completely.

4.3. *FCM test bed*

The communication scenario of FCM was identical to that of GCM for the evaluation of OWD, RTT, NDU and POH. For the estimation of BD, the scenario was similar to that of WS since the application server disconnection bug was finally fixed in the implementation of FCM that we used in our tests. Due to the presence of this bug, BD tests of GCM were done by sending periodic downstream messages, but the test for FCM used upstream messages as in the case of WS.

5. Test bed devices and configurations

The devices used in all tests can be divided into two categories, namely mobiles and servers. All details regarding them and the software used are reported in the following sections.

5.1. *Mobiles*

For OWD, RTT, NDU and POH analysis, we used these devices:

- Samsung Galaxy SII (GT-I9100) running Android 4.1.2 with 1.2 GHz dual-core Central Processing Unit (CPU) and 1 GB of RAM
- HTC One S (Z520e) running Android 4.1.1 with 1.2 GHz dual-core CPU and 1 GB of RAM

For BD analysis, we used two similar devices:

- Asus Google Nexus 7 Wi-Fi (ME370T) running Android 5.1.1 with 1.5 GHz quad-core CPU, 1 GB of RAM and 4325 mAh LiPo battery
- Asus Google Nexus 7 Cellular (ME370TG) running Android 5.1.1 with 1.5 GHz quad-core CPU, 1 GB of RAM and 4325 mAh LiPo battery

As far as softwares are concerned, for SOAP WS, we used ksoap2-android library [20], which is a lightweight and efficient client library supporting consumption and serialisation of SOAP WS. For RESTful WS, we used Java's HttpURLConnection. All GCM and FCM payloads were encrypted and WS payloads were not encrypted due to the fact that this is their default behaviour.

GCM requires devices running Android version 2.2 or higher. FCM requires devices running Android version 4.0 and higher. Both GCM and FCM require the device to support Google Play Services. The version of ksoap2-android library used for these test beds uses Java version 1.5 which makes it compatible with all versions of Android higher than 4.0. Java's HttpURLConnection is part of Android's API since API level 1 (Android version 1.0). All libraries and API's used for these tests are compatible with the latest available version of Android (as of October

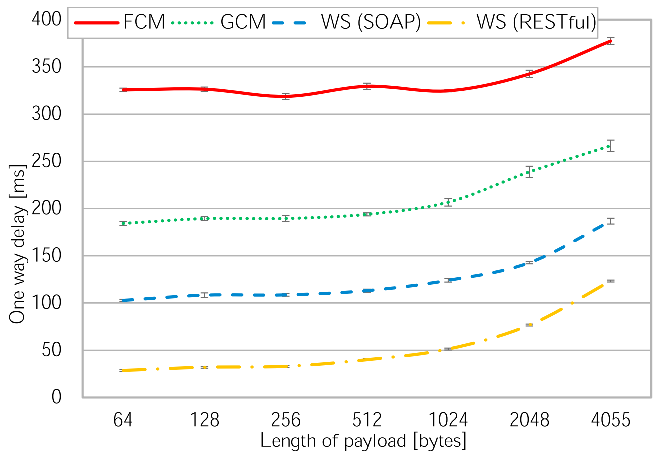

Fig. 6. OWD comparison.

2018, version 9.0).

5.2. *Servers*

To implement a CCS application server for GCM, we used Smack library [21] (as suggested by Google guidelines) which is an open-source XMPP client library. For the implementation of XMPP application server for FCM, we used Smack library's version 4.2.1.

For both WSs, we used Java EE's built-in libraries for implementation. Additionally, for RESTful WS, JSON parsing was done using an open-source implementation for JSON called JSON-java library [22].

6. Experimental results

Test results are shown for each performance parameter.

Fig. 6 summarises OWD averages for all tests. For all payload lengths, on average, SOAP WS took 127 ms, RESTful WS took 55 ms, GCM took 210 ms and FCM took 335 ms for a packet to travel from source to destination. RESTful WS showed the best performance while FCM was the worst (followed by GCM), which can be explained by the presence of an extra hop in the communication path. All communication techniques shared a general trend of performance: they slowed down as the size of payload increased, likely due to packet segmentation and increase in CPU computation time.

Fig. 7 shows RTT averages for all tests. For all payload lengths, on average, SOAP WS took 177 ms, RESTful WS took 100 ms, GCM took 531 ms and FCM took 697 ms for a packet to travel from source to destination and the acknowledgement from the destination back to the

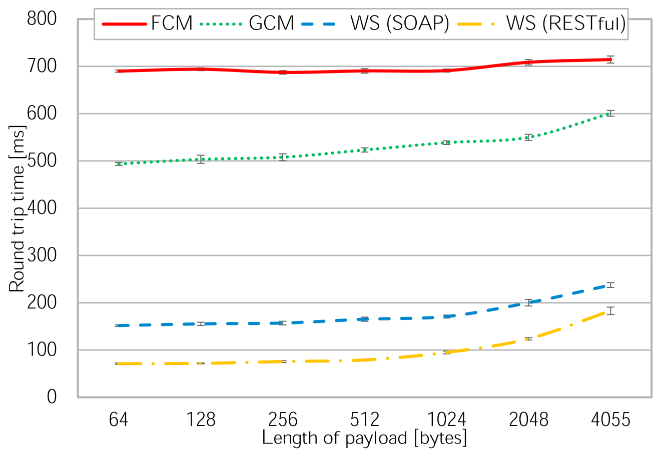

Fig. 7. RTT comparison.

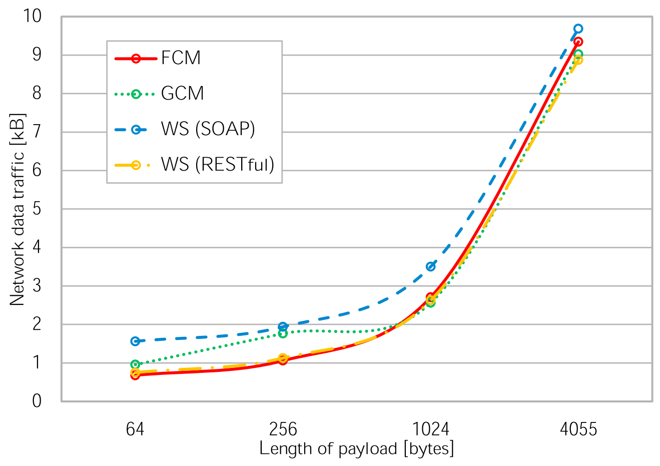

Fig. 8. NDU comparison in RTT scenario.

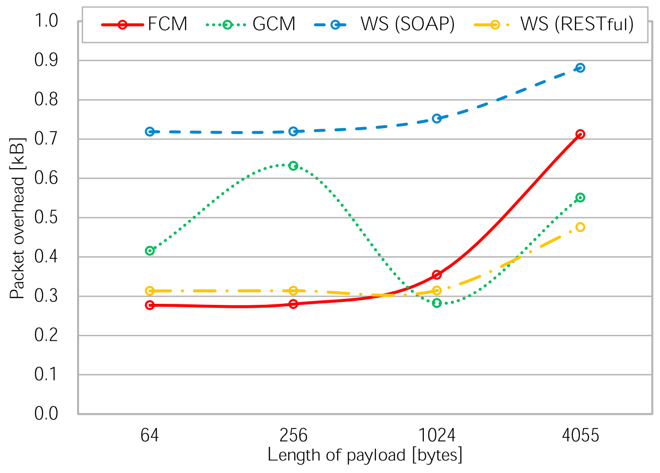

Fig. 9. Packet overhead comparison.

source. RESTful WS again showed the best performance while FCM was the slowest, which can be explained as before. Also, the general trend of performance slowed down when larger payload sizes were still present.

Fig. 8 reports NDU for multiple payload lengths, where each point is the data traffic, measured on the wire, at the smartphone in RTT scenario. In other words, each point is the sum of inward and outward data traffic at the smartphone end. For example, if the smartphone sends a message (upstream) with a payload of p bytes, the outward traffic at smartphone end is expected to be $p+x$ bytes and the inward traffic at smartphone end should be $p+y$ bytes. The total data traffic (reported in the figure) is expected to be $2p+x+y$ bytes. Overall, for these tests with four payloads in RTT scenario (upstream message followed by echoed downstream message), RESTful WS used 13.4 kB, GCM used 14.3 kB, FCM used 13.7 kB and SOAP WS used 16.7 kB of data on the network at the smartphone end. RESTful WS used the minimum amount of data while SOAP WS used the maximum, as it was expected due to the use of JSON instead of XML for serialisation. GCM and FCM used encryption and fall between RESTful WS and SOAP WS.

Fig. 9 shows the average POH for multiple payload lengths, which is given by: $(x+y)/2$ bytes. For all payload lengths, SOAP WS had 18%–92%, RESTful had 11%–83%, GCM had 12%–87% and FCM had 15%–82% POH. The overall trend was that the size of POH remained more or less the same for smaller payloads, but as the payload size increased, the POH size increased as well. This happened due to packet segmentation applied to larger payload sizes. Also, at smaller payload sizes (64 bytes and 256 bytes), FCM used less POH compared with others, while at larger

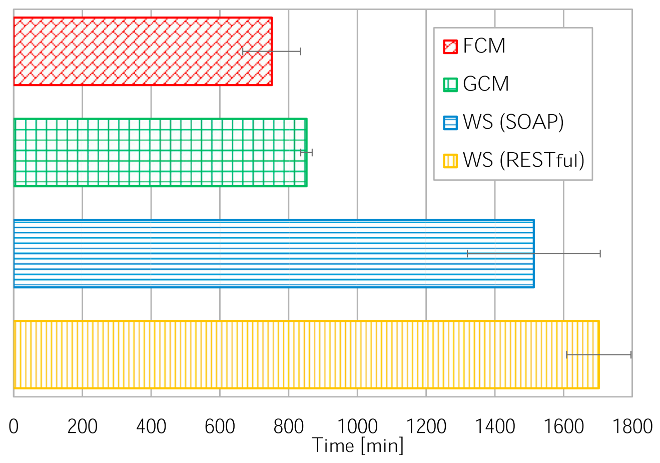

Fig. 10. BD comparison.

payload sizes (1024 bytes and 4055 bytes), RESTful WS used less POH. Overall, on average, RESTful WS showed the minimum while SOAP WS showed the maximum NDU and POH, with FCM falling very close to RESTful WS.

It is worth to notice that for a payload size of 256 bytes GCM exhibits an inconsistent behaviour with respect to all other payload sizes. This anomaly was investigated and confirmed by our tests but not explained due to the lack of a comprehensive description of the GCM protocol. What we observed is that for this particular payload size, the packet overhead is very high as if some paddings were used to increase the overall GCM message size. However, notice that FCM, which replaced GCM, no more exhibits this behaviour, which could suggest the presence of a bug in GCM later fixed in FCM.

Fig. 10 provides BD averages for two tests per test device (*i.e.,* four complete battery cycles). The battery showed the longest duration using RESTful WS and the shortest one using FCM. This was likely due to the need to maintain an active connection with the intermediate server. The temperature of the devices was between 25 °C and 30 °C during all tests.

7. Conclusion

A set of tests in a simple client-server scenario were done for four messaging techniques: SOAP WS, RESTful WS, GCM and FCM. The tests evaluated time-related performance, data usage and battery duration. The results confirmed that RESTful WS outperforms other communication techniques in all evaluated performance matrices. FCM showed the worst performance for 3 out of 5 evaluated parameters (*i.e.,* OWD, RTT and BD). However, it is important to mention that FCM and GCM support more features out-of-the-box than WS, such as push technology, data encryption, native Android support as well as official iOS SDK support by Google. As far as the question is concerned, if WS is replaced by Google's messaging services (GCM and FCM), the answer depends on the particular requirements of the application. Google's messaging services perform really well as far as value-added features and smaller payloads are concerned. However, for scenarios where the delivery of larger payloads is required or where there are time constraints, WS should be the preferred solution. Although no tests were done for iOS, it is still realistic to assume that the rankings in all scenarios would not change.

References

[1] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services: Concepts, Architectures and Applications, Springer, 2004, pp. 123–125. Ch. Web services.

[2] Google, Google cloud messaging, Website, 2016. https://developers.google.com/cloud-messaging/. (Accessed 10 March 2016).

[3] Google, Firebase cloud messaging, Website, 2017. https://firebase.google.com/docs/cloud-messaging/. (Accessed 25 November 2017).

[4] Apple, Local and remote notification programming guide: APNs overview, Website, 2018. URL, https://developer.apple.com/library/archive/documentation/Networ

kingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html#//apple_ref/doc/uid/TP40008194-CH8-SW1. (Accessed 6 February 2019).

[5] Google, Google cloud messaging - messaging concepts and options, Website, 2017. URL, https://developers.google.com/cloud-messaging/concept-options#common-message-options. (Accessed 10 December 2017).

[6] W. W. W. C, W3C, Web services architecture, Website, 2002. URL, https://www.w3.org/TR/2002/WD-ws-arch-20021114/. (Accessed 25 November 2017).

[7] L. Richardson, S. Ruby, D.H. Hansson, Ch. RESTful, Resource-oriented architectures, RESTful Web Services, vol. 1, O'Reilly Media, 2008, pp. 13–14. URL, http://www.oreilly.com/catalog/9780596529260/.

[8] J. Snell, D. Tidwell, P. Kulchenko, Programming web services with SOAP, Ch. Describing a SOAP service, O'Reilly Media, 2002. xiii + 244. URL, https://books.google.it/books?id=ALo1LxID5q0C.

[9] A. Rodriguez, Restful web services: the basics, IBM developerWorks 33, URL, https://developer.ibm.com/articles/ws-restful/.

[10] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana, Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI, IEEE Internet Computing 6 (2) (2002) 86–93, https://doi.org/10.1109/4236.991449. URL, https://ieeexplore.ieee.org/abstract/document/991449.

[11] Wikipedia, Firebase, Website, [Accessed 9 Nov. 2018]. URL https://en.wikipedia.org/wiki/Firebase.

[12] Google, Documentation: about FCM messages, Website, 2017. URL, https://firebase.google.com/docs/cloud-messaging/concept-options. (Accessed 25 November 2017).

[13] J. Walrand, S. Parekh, Communication networks: a concise introduction, in: Of Synthesis Lectures on Communication Networks, vol. 3, Morgan & Claypool Publishers, 2010, pp. 1–192, https://doi.org/10.2200/S00254ED1V01Y201002CNT004. Ch. Delay, http://www.morganclaypool.com/doi/abs/10.2200/S00254ED1V01Y201002CNT004.

[14] A. Farrel, Network management know it all, Morgan Kaufmann Know it All Series, Elsevier Science, 2008, pp. 135–136. Ch. Active Network Monitoring, http://www.amazon.com/dp/0123745985.

[15] K. Arai, L. Sugiyanta, Agent based approach of routing protocol minimizing the number of hops and maintaining connectivity of mobile terminals which move one area to the other, vol. 6018, LNCS, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 305–320, https://doi.org/10.1007/978-3-642-12179-1-27. URL, https://doi.org/10.1007/978-3-642-12179-1.

[16] Wireshark Foundation, Wireshark, [Version 2.2.1]. URL https://www.wireshark.org/.

[17] Google, XMPP connection server reference, Website, 2017. URL, https://developers.google.com/cloud-messaging/xmpp-server-ref. (Accessed 25 November 2017).

[18] Google, Documentation: firebase cloud messaging XMPP protocol, Website, 2017. URL, https://firebase.google.com/docs/cloud-messaging/xmpp-server-ref. (Accessed 25 November 2017).

[19] Google, Registering client apps, Website, [Accessed 10 Mar. 2016]. URL https://developers.google.com/cloud-messaging/registration.

[20] Simpligility, ksoap2-android, [Version 3.6.2]. URL https://simpligility.github.io/ksoap2-android/.

[21] Ignite Realtime, Smack, [Version 4.2.1]. URL https://www.igniterealtime.org/projects/smack/index.jsp.

[22] Stleary, JSON-java, [Release 20160810]. URL https://github.com/stleary/JSON-java.