# Demo Abstract: MOS: A Bandwidth-Efficient Cross-Platform Middleware for Publish/Subscribe

Christoph Doblander, Simon Zimmermann, Kaiwen Zhang, Hans-Arno Jacobsen
Chair of Application and Middleware Systems
Technische Universität München
{doblande, zimmerms, kzhang, jacobsen}@cs.tum.de

## ABSTRACT

Shared dictionary compression is known as an efficient compression method for pub/sub. In practice, bandwidth reductions of more than 80% are achievable for JSON or XML data formats. Compared to other compression techniques such as GZip or Deflate, a dictionary is needed to compress and decompress messages. Generating a dictionary is a CPU-expensive task and sharing it introduces bandwidth overheads. Furthermore, the dictionary is continuously maintained to keep the compression performance high. We developed MOS: a cross-platform middleware for managing shared dictionary compression tasks. This includes dictionary propagation, compression/decompression, and periodic maintenance. We provide a developer API to interact with the MQTT-based pub/sub infrastructure. Our demo shows an example application built on top of MOS which shows the performance of the shared dictionary compression scheme.

## 1. INTRODUCTION

The back-end infrastructure of smartphone applications often uses a pub/sub middleware to disseminate notifications in a scalable way [6]. As an example, the Facebook Messenger application uses MQTT [2], a pub/sub protocol, to communicate with the servers.

Mobile phone data connections are often metered and the available bandwidth in rural areas tends to be lower than in city centers. To reduce data usage of applications further than what is possible using GZip or Deflate, we propose a shared dictionary compression (SDC) scheme for pub/sub. Using that technique, bandwidth reductions of more than 80% can be achieved by amortizing the overhead of dictionary propagation [5]. To leverage SDC, the developer would have to keep track of the dictionaries at the publisher and subscriber sides and additionally maintain the dictionary over time by implementing a sampling broker. We introduce a new middleware, called MQTT On Steroids (MOS), which manages the SDC protocol and provide the developer with a standard pub/sub API. Hence the developer can benefit from the bandwidth reductions without additional complexity in the codebase. Figure 1 shows an overview of the abstracted logic of the framework. MOS encapsulates the SDC logic on top of MQTT [4] and the compression library, and provides a pub/sub API for the Cordova framework. Cordova [1] allows the developer to create a mobile phone application using Javascript and HTML. From a single codebase, it is possible to target multiple platforms like Apple IOS or Android. A library in C++, FemtoZip [3], implements methods to compress and decompress messages using a dictionary. To make this library available within Cordova, it is wrapped using Java on Android or Swift on IOS.

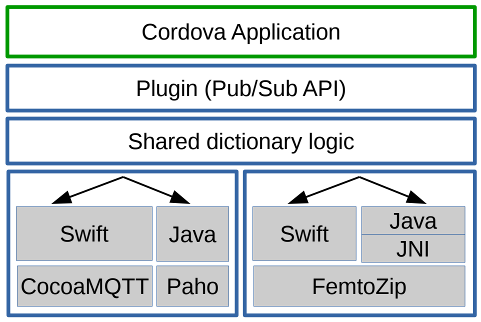


Figure 1: Pub/sub API using SDC

In this demo, we show the capabilities of our SDC middleware in an example application. The application leverages the framework to allow the user to publish/subscribe on multiple smartphone platforms. The user interface shows the bandwidth savings, notification rate, and other technical information of the connection to the broker.

## 2. MIDDLEWARE FOR SDC

The MOS middleware contains three parts: the client, the sampling broker (SB) and the caching service (CS) which runs on top of MQTT [4]. The MOS client library is used by the developer to enable bandwidth savings on mobile phone clients. The MOS SB maintains the dictionaries and the MOS CS acts as a key-value store where newly joining publishers and subscribes can request dictionaries.

The SB creates the dictionaries and observes the bandwidth reductions. The SB executes an adaptive algorithm which decides to either sample a new dictionary or to extend the time to live (TTL) of the old dictionary. Reasons

*MIDDLEWARE '16 Trento, Italy*
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

| Format | Deflate | Adaptive algorithm |
|---|---|---|
| DEBS-CSV | 34.5% | 41.65% |
| DEBS-XML | 53.3% | 85.46% |
| DEBS-JSON | 45.4% | 41.65% |

Table 1: Bandwidth reduction [5]

for a new dictionary include changes over time in the content of notifications which could make a dictionary obsolete. The adaptive algorithm chooses the parameters for the dictionary, the size, and over how many historic notifications should be sampled. The size of the dictionary impacts compression performance and bandwidth overhead and the length of historic messages impacts the time to sample a dictionary. The adaptive algorithm is a heuristic which tries to find a balance between the two parameters [5]. The client library is split up in several layers (see Figure 1). It encapsulates a MQTT client and the library FemtoZip [3] which is used for dictionary-based compression. The SDC logic keeps track of current active dictionaries and removes them when TTL is reached. Figure 1 shows the interactions between client and MQTT broker. Each topic has a corresponding dictionary topic. As an example, when the developer subscribes to a topic or publishes a new notification, the library additionally subscribes to the corresponding dictionary topic. In case a compressed notification is received, the library uncompresses the notification transparently using the correct dictionary. Also when a new notification is published, it is transparently compressed in case a dictionary is available for this topic.

Our demonstration, see Figure 3 is done with an example dataset from the DEBS 2015 Grand Challenge [7]. Table 1 shows the achieved bandwidth savings using the MOS middleware when using the dataset. One interesting observation is that the final compressed size of notifications across formats is nearly the same, with at most 88% reduction for XML. Therefore, the developer can choose the notification format which is most practical or convenient without worrying about data size.

## 3. REFERENCES

[1] Apache Cordova. https://cordova.apache.org/.

[2] Facebook Messenger. https://www.facebook.com/notes/10150259350998920.

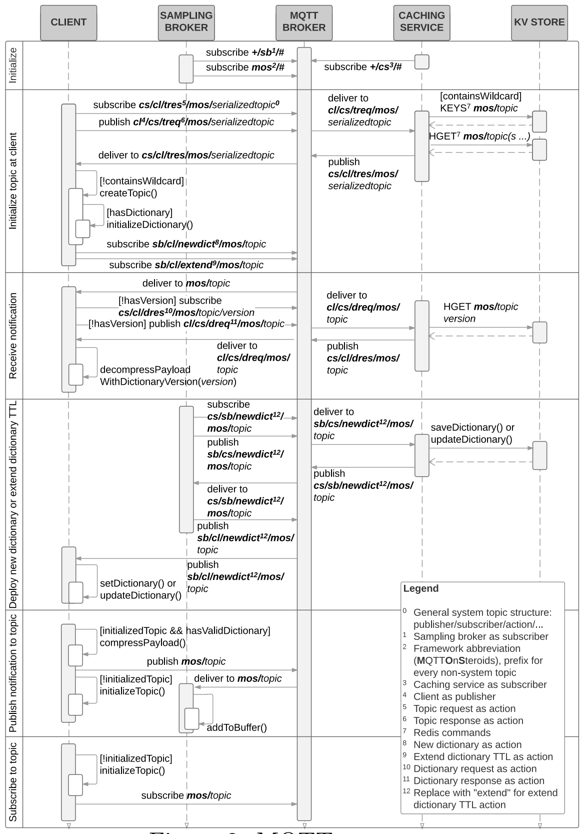
[3] FemtoZip. https://github.com/gtoubassi/femtozip.

[4] MQTT. http://mqtt.org/.

[5] C. Doblander, T. Ghinaiya, K. Zhang, and H.-A. Jacobsen. Shared Dictionary Compression in Publish/Subscribe. In *Proceedings of the 10th ACM International Conference on Distributed Event-Based Systems*, DEBS '16, 2016.

[6] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES Publish/Subscribe System. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. IGI Global, 2010.

[7] Z. Jerzak and H. Ziekow. The DEBS 2015 Grand Challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 266–268, New York, NY, USA, 2015. ACM.



CLIENT
SAMPLING BROKER
MQTT BROKER
CACHING SERVICE
KV STORE
Initialize
subscribe +/sb¹/#
subscribe mos²/#
subscribe +/cs³/#
Initialize topic at client
subscribe cs/cl/tres⁵/mos/serializedtopic⁰
publish cl⁴/cs/treq⁶/mos/serializedtopic
deliver to cl/cs/treq/mos/serializedtopic
[containsWildcard] KEYS⁷ mos/topic
HGET⁷ mos/topic(s ...)
deliver to cs/cl/tres/mos/serializedtopic
publish cs/cl/tres/mos/serializedtopic
[!containsWildcard] createTopic()
[hasDictionary] initializeDictionary()
subscribe sb/cl/newdict⁸/mos/topic
subscribe sb/cl/extend⁹/mos/topic
Receive notification
deliver to mos/topic
[!hasVersion] subscribe cs/cl/dres¹⁰/mos/topic/version
[!hasVersion] publish cl/cs/dreq¹¹/mos/topic
deliver to cl/cs/dreq/mos/topic
HGET mos/topic version
deliver to cl/cs/dreq/mos/topic
publish cs/cl/dres/mos/topic
decompressPayload WithDictionaryVersion(version)
Deploy new dictionary or extend dictionary TTL
subscribe cs/sb/newdict¹²/mos/topic
deliver to sb/cs/newdict¹²/mos/topic
saveDictionary() or updateDictionary()
publish sb/cs/newdict¹²/mos/topic
deliver to cs/sb/newdict¹²/mos/topic
publish cs/sb/newdict¹²/mos/topic
publish sb/cl/newdict¹²/mos/topic
publish sb/cl/newdict¹²/mos/topic
setDictionary() or updateDictionary()
Publish notification to topic
[initializedTopic && hasValidDictionary] compressPayload()
publish mos/topic
[!initializedTopic] initializeTopic()
deliver to mos/topic
addToBuffer()
Subscribe to topic
[!initializedTopic] initializeTopic()
subscribe mos/topic
Legend
⁰ General system topic structure: publisher/subscriber/action/...
¹ Sampling broker as subscriber
² Framework abbreviation (MQTTOnSteroids), prefix for every non-system topic
³ Caching service as subscriber
⁴ Client as publisher
⁵ Topic request as action
⁶ Topic response as action
⁷ Redis commands
⁸ New dictionary as action
⁹ Extend dictionary TTL as action
¹⁰ Dictionary request as action
¹¹ Dictionary response as action
¹² Replace with "extend" for extend dictionary TTL action

Figure 2: MQTT sequence

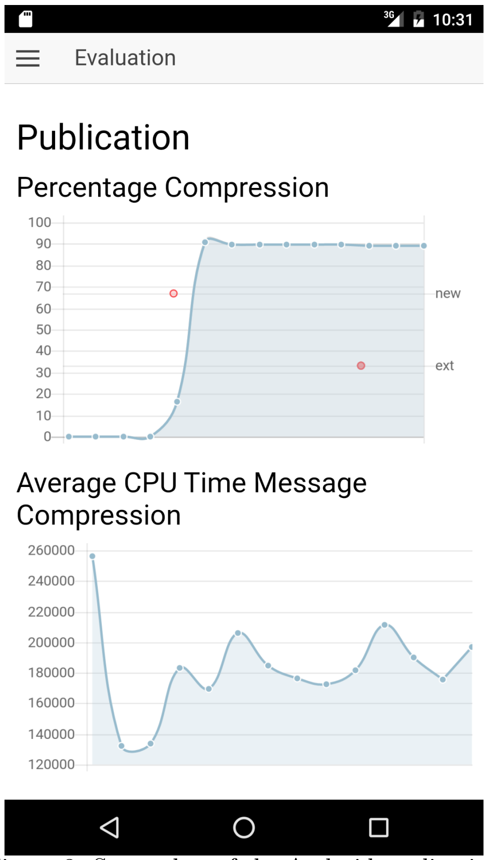


Figure 3: Screenshot of the Android application

We recorded a demonstration of our mobile applications. On the left side you can see the IOS Emulator, on the right side the same application running in an Android Emulator. The terminal at the bottom of the screen shows the sampling broker. https://www.dropbox.com/s/8wtrwaxzeplaq4b/middleware_demo.mp4?dl=1

For the real demo we would need access to Wifi. We will show the mobile application on a mobile phone. In case the Wifi is interrupted we plan to have a backup recording on a laptop.