Shared Dictionary Compression in Publish/Subscribe Systems

Christoph Doblander, Tanuj Ghinaiya, Kaiwen Zhang, Hans-Arno Jacobsen
Middleware Systems Reseach Group (MSRG)
Technische Universität München
{doblande, ghinaiya, zhangk, jacobsen}@in.tum.de

ABSTRACT

Publish/subscribe is known as a scalable and efficient data dissemination mechanism. Its efficiency comes from the optimized routing algorithms, yet few works exist on employing compression to save bandwidth, which is especially important in mobile environments. State of the art compression methods such as GZip or Deflate can be generally employed to compress messages. In this paper, we show how to reduce bandwidth even further by employing Shared Dictionary Compression (SDC) in pub/sub. However, SDC requires a dictionary to be generated and disseminated prior to compression, which introduces additional computational and bandwidth overhead. To support SDC, we propose a novel and lightweight protocol for pub/sub which employs a new class of brokers, called *sampling* brokers. Our solution generates, and disseminates dictionaries using the sampling brokers. Dictionary maintenance is performed regularly using an adaptive algorithm. The evaluation of our proposed design shows that it is possible to compensate for the introduced overhead and achieve significant bandwidth reduction over Deflate.

CCS Concepts

•**Networks** → **Network protocols; Cloud computing;**
•**Information systems** → *Data compression; Data exchange;*

Keywords

Publish/Subscribe, Middleware, Compression

1. INTRODUCTION

In the context of mobile applications, publish/subscribe (pub/sub) [17] is used as the dissemination substrate between the back end infrastructure and the smartphone applications to deliver notifications. As an example, Facebook Messenger uses MQTT [1], a pub/sub protocol, to communicate with the server.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DEBS '16, June 20-24, 2016, Irvine, CA, USA

© 2016 ACM. ISBN 978-1-4503-4021-2/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2933267.2933308

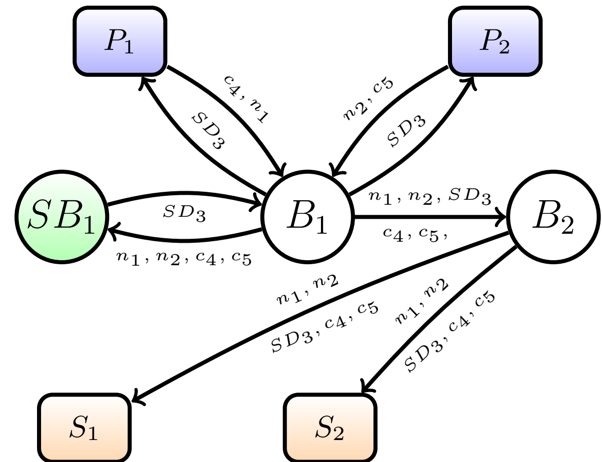

Figure 1: Notification delivery in pub/sub overlay

In this paper, we address the problem of high bandwidth usage for notifications in pub/sub. Mobile phone data connections are often metered. The available bandwidth also tends to be lower in rural areas than in city centers. Reducing bandwidth usage reduces costs and allows the use of collaborative applications even in areas where bandwidth is sparse.

While advanced pub/sub routing algorithms do provide some form of bandwidth economy [19], few works tackle the problem explicitly by compressing notifications. Popular compression methods, such as GZip or Deflate, can generally be applied for moderate performance gains. However, we demonstrate how further compression can be obtained by employing Shared Dictionary Compression (SDC) in pub/sub.

Our approach adds a new type of broker, subsequently called *sampling brokers*, to the pub/sub design. A sampling broker is responsible for sampling notifications to create dictionaries, maintenance of said dictionaries over time and disseminating them in the overlay network. An adaptive algorithm is employed for periodical maintenance which creates a new dictionary with specific parameters when it is beneficial to do so over keeping an older version, and fault-tolerance is provided by sending the dictionaries to a caching service. Figure 1 shows a high level overview of a broker overlay extended by the sampling broker SB_1. Publishers $P_{1,2}$ send messages to the subscribers $S_{1,2}$. After the notifications $n_{1,2}$ are published, the adaptive algorithm decides

that the bandwidth can be reduced by employing a dictionary. A new dictionary SD_3 is sampled and sent through the pub/sub overlay. The dictionary can then be used by $P_{1,2}$ to compress the notifications $c_{4,5}$ and $S_{1,2}$ can uncompress the notifications. The figure is used as an example throughout the whole paper.

To motivate and evaluate our work, we consider the following representative use case. We imagine a scenario where mobile phones use an application to communicate with a back end, which provides various services. Upon starting the application on a smartphone, a map centered around its current location is displayed. Additionally, the application subscribes to nearby notifications in order to refresh its view of the map. All user-facing events are sent via the mobile phone network. While fast mobile connections are readily available in city centers, the same cannot be said for rural areas. Providing the user a real-time view of the map requires a high incoming event throughput rate, which may not be desirable if the phone is operating on a metered data plan. We evaluate this use case using the dataset of the DEBS 2015 Grand Challenge [18], which contains taxi trips records in NYC for a year. We measure the bandwidth savings when these events are sent to a mobile phone that displays the information on a map in real-time.

The contributions of this paper are:

1. We present our solution *Simple SDC for pub/sub* (SPSS): a fault-tolerant pub/sub design with Shared Dictionary Compression for efficient publication traffic reduction.

2. We introduce the use of *sampling* brokers for generating, maintaining, and disseminating dictionaries.

3. We provide an adaptive algorithm for dictionary maintenance, which considers the benefits of generating a new dictionary vs. the dictionary sharing overhead with varying parameters.

4. We evaluate our algorithm using several real world datasets by comparing to Deflate. Our evaluation is implemented on top of a MQTT broker using a smartphone client.

The following section, Section 2, provides background material on compression and pub/sub. In Section 3, we look at relevant literature on compression in distributed systems. Section 4 presents our design for pub/sub using SDC. We then compare our approach to the state of the art in Section 5. We finally conclude in Section 6, where we offer our final observations and outline future work.

2. BACKGROUND

Pub/sub is an effective method to disseminate data while decoupling data sources and sinks [17]. Event sources publish notifications (also called publications) on a topic and brokers route the notifications to interested subscribers. In a topic-based model, subscribers express their interest in a topic. The events belonging to that topic are then routed through the broker overlay network to the interested subscribers. In a content-based system, subscribers can additionally express predicates in their subscriptions which further filter publications belonging to a topic, by comparing those predicates with attribute-value pairs embedded in the publications. Note that our solution supports all types of matching.

SDC leverages similarity between notifications to improve compression ratios. One of the first papers [22] about dictionary-based compression achieves 60%-70% compression for English documents using a small dictionary. This idea is further explored in [11] and subsequently extended in [12]. In this paper, we use *Shared Dictionary Compression* (SDC) to refer to a combination of dictionary-based compression and multiple passes of Huffman Coding. Using this method, references to the dictionary are represented very efficiently. We use the open-source library FemtoZip [2], which implements said technique.

Dictionary based compression is proposed for HTTP [14]. SDCH is a proposal for a HTTP/1.1-compatible extension to enable inter-response data compression. LinkedIn reported [9] an average bandwidth reduction of 24% on top of GZIP. Brotli [10], a recent Internet-Draft, also contains a static dictionary sampled from a multi-lingual web corpus to improve compression ratios specifically for web pages.

BigTable [15], a distributed storage system for structured data, uses a custom compression scheme, which includes sampling a dictionarySampling a dictionary. First, *Data Compression Using Long Common Strings* [11] is used. In the second pass, a small window is used, similar to Deflate. They report a 10-to-1 reduction in space. The main reason is that web pages from a single host share large amounts of boiler plate which is identified in the first run and included in the dictionary. Dictionary-based compression is also used in main memory column stores [13]. The dictionary can also be used to index and to rewrite queries. This approach reduces the memory consumption of the database.

Compression schemes, like GZip [8], Snappy or Deflate [7], use a sliding window and leverage the redundancy within that window to compress a notification. The redundancy within a window tends to be low compared to the redundancy between two subsequent notifications. Therefore, we argue that SDC reduces bandwidth beyond what is possible with traditional compression methods.

Deflate [7] uses a combination of LZ77 and Huffman coding. GZIP [8] uses Deflate as the compressed data format. GZIP appends a header and a CRC32 checksum to Deflate. When small notifications are compressed, it is possible that the compression gains do not outweigh the additional large header. Thus, we employ Deflate, not GZIP, as a baseline for comparison.

3. RELATED WORK

To the best of our knowledge, there is no paper that explores the use of SDC within pub/sub. Consequently, we extend the scope of our related work to general bandwidth reduction mechanisms using delta compression [21] and deduplication [23].

In delta compression [16], one notification is described in terms of another notification.

REAP [21], another data differencing algorithm (MultiDiff), performs significantly better than simple delta encoding. The patch can reference multiple other preceding notifications. The protocol uses windows of notifications, which can be referenced by the diff. To address out-of-order notifications, the delta is always computed to notifications within a certain preceding window; the subscribers are then

instructed to cache this window. The notifications in the window can also be diffs of preceding notifications; hence, at-least once guarantees are mandatory. Our work targets a different kind of environment, where publishers and subscribers often join or leave the network, which is the case when a mobile phone user opens or closes an application. In addition, our approach should not require ordering guarantees that are difficult to fulfill in cyclic pub/sub broker overlays. In content-based pub/sub, this approach has another weakness, since subscribers to the same topic receive different publications due to their unique predicates. In this case, the deltas would have to be re-encoded between communicating peers introducing additional computational overhead.

Deduplication algorithms are used to reduce bandwidth usage in the context of database replication [23]. For the same reasons as delta compression, we do not consider this approach.

4. SDC IN PUB/SUB

Our approach, called Simple SDC for pub/sub (SSPS), introduces several new components. We introduce a new broker class, called Sampling Broker (SB), which can be a separate broker instance or a role assigned to an existing broker. Additionally, we introduce a caching service (CS) to provide fault tolerance. The responsibility of a SB is to create dictionaries, monitor the compression ratio, and maintain the dictionary over time. The CS caches the dictionaries, such that newly joining subscribers or publishers can acquire required dictionaries.

Dictionary sampling: To enable SDC in pub/sub, we first have to sample notifications to create dictionaries. In SSPS, we propose a single dedicated SB per topic, with a dictionary generated for each. A SB carries the main computational load of our approach, which is the sampling of publications. To balance the computational load of many topics, consistent hashing [20] of the topics could be used to assign the SB to a physical instance. A SB subscribes to the topic and accumulates the notifications in a ring buffer with a fixed size (B_{size}). A predefined hash function is used to generate the dictionary based on the stored notifications.

Sharing of the dictionary: After the dictionary is sampled, it is added to the cache and published on the corresponding dictionary topic. Note that all publishers and subscribers sending and receiving data on a topic are subscribed to the corresponding dictionary topic and will receive the appropriate dictionary. After a fixed expiry timestamp T_{exp} attached to each dictionary elapses, no publisher is allowed to compress a notification using this dictionary. L_{max} is a predefined time interval, which is well above the worst case maximum publisher-subscriber latency and the maximum clock skew within the network. After $T_{exp} + L_{max}$ the subscriber is also allowed to dismiss the dictionary. Each dictionary has an identifier, which is referenced to in the compressed notifications.

Caching service: The caching service provides fault tolerance for the dictionaries. Every SD is cached until $T_{exp} + 2 \times R \times L_{max}$. The CS is essentially a fault-tolerant key-value store.

Continuous dictionary maintenance: Before a dictionary expires, the SD has to either increase the expiry time or sample a new dictionary. The SB is only allowed to increase the expiry time until $T_{exp} - L_{max}$. First, the expiry in the caching service is increased, then a new notification on the dictionary topic is published, which increases the expiry of the SD at the publishers and subscribers.

New publishers: We have to consider two cases. The first occurs when a publisher creates a new topic and starts publishing, the second case occurs when the publisher starts publishing on an existing topic. In the first case, the publisher starts sending notifications on the topic and also subscribes to the dictionary topic. When the SB has created a dictionary, it will be published on the dictionary topic which belongs to the notification topic. The publisher always compresses the notifications using the dictionary whose expiry is the farthest in future. In the second case, the publisher can start sending the notifications without compression and eventually acquire the SD from the CS. No additional latency is incurred since the publisher can always publish uncompressed notifications.

New subscribers: When a subscriber subscribes to a topic, it also issues a subscription to the corresponding dictionary topic, which is calculated using a predefined hash function on the original topic. When the subscriber receives a compressed notification and the dictionary is not available, the dictionary has to first be acquired from the caching service. In this case, an additional latency of L_{max} can occur when receiving the first message.

Dictionary maintenance: The dictionary has to be resampled to maintain high bandwidth savings. Hence, we propose an adaptive algorithm, which probes the notifications to detect if a new dictionary would improve the compression (see Section 4.2). Each dictionary has an identifier, which is unique within a topic and within the timespan $T_{exp} + 2 \times L_{max}$. In our experiments, a single byte is used to represent this ID. Figure 1 shows an example of the algorithm in practice. The subscript numbers on the notifications denote the sequence. The assumption is that the subscribers $S_{1,2}$ are already connected, but no notification have been sent so far. The publishers $P_{1,2}$ start to publish the notifications $n_{1,2}$. SB_1 and $S_{1,2}$ receive the notifications. SB_1 begins to sample the notifications and publishes a new SD_3 on the corresponding dictionary topic. $P_{1,2}$ and $S_{1,2}$ are subscribed to the topic and receive D_3. From now on, $P_{1,2}$ can publish compressed notifications $c_{4,5}$. The subscribers $S_{1,2}$ can decompress the notifications using the cached dictionary.

4.1 Analysis of overhead

Dictionary publication: The dictionary size is bounded to a multiple of the average uncompressed notification size. In our evaluation, we consider multipliers of up to 21 of the original notification size + Huffman Trees. All publishers $|P|$ and subscribers $|S|$ have to receive the dictionary. The total bandwidth used for spreading a new dictionary is $(|P|+|S|) \times SD_{size}$.

Memory consumption of dictionaries across the overlay: There can be situations when multiple dictionaries are active at the same time. This situation occurs when the bandwidth savings of the new dictionary pay off faster than waiting for the expiry of the old dictionary. This depends on the behavior of the dictionary maintenance algorithm and how it adapts to changes in the content of publications. In general over time, the sampling frequency and estimation of T_{exp} should be stable. Hence the additional memory consumption of the SD is on the publisher and subscriber SD_{size}.

Dictionary sampling time: Sampling a dictionary is at worst an $O(N^{3/2})$ operation [11]. For practical workloads, the average case is $O(N)$. N stands for the total number of bytes of the notifications in the buffer, $\sum_{i=0}^{bufferlength} = |N_i|$. Our experiments confirm this observation. Once the dictionary is sampled, it is limited to a specific size. The cost of trimming a dictionary to a certain size is negligible compared to the sampling time.

4.2 Adaptive algorithm

We propose an adaptive algorithm to choose $SD_{multiplier}$, B_{size}, and T_{exp}. Our proposed heuristic is conservative in the sense that a new dictionary is only spread when it is certain that the cost of spreading can be amortized.

Overview: Every time a dictionary expires (T_{exp} is reached), the algorithm first computes the amount of bandwidth reduction that can be achieved if a new dictionary would be published. If the gain from a new dictionary is higher than a certain threshold, the dictionary is published, otherwise the existing dictionary is prolonged. In addition, the algorithm changes the parameters of the dictionary creation ($SD_{multiplier}$ or B_{size}) for the bandwidth evaluation at the next expiry time.

Bandwith reduction: To calculate the amortization time, we need to estimate the bandwidth reduction (BR) of a new dictionary. BR is estimated by splitting the recorded messages into a training and validation set. Splitting the data into two independent sets is a technique known from machine learning. In case the set is not split, the calculated bandwidth reductions tend to be not reproducible similar to when a machine learning method is validated on the training data. A dictionary is sampled on the training set and the bandwidth reduction is derived by comparing the uncompressed validation set with the compressed validation set. We decided to take 70% for training and 30% of the notifications for validation.

We can calculate the bandwidth reduction (BR) of the new dictionary using the uncompressed size of the notifications TB and the size of the compressed notifications CTB using Equation 1:

$$BR = 1 - \frac{CTB}{TB} \tag{1}$$

Amortization time: The time needed to amortize the dictionary $T_{amortize}$ is calculate using the current rate (R) and the estimated bandwidth reductions (BR). The rate, see Equation 2, is derived by dividing the total size of the notifications by the time span between the first and last message in the buffer.

$$\begin{aligned} TS &= t_{buffersize} - t_0 \\ R &= \frac{\sum_{i=0}^{buffersize} |n_i|}{TS} \end{aligned} \tag{2}$$

The amortization time then takes the size of the dictionary and the bandwidth reduction into account using the estimated rate R (see Equation 3):

$$T_{amortize} = \frac{|SD|}{R \times BR} \tag{3}$$

Dictionary parameters: When the dictionary is sampled, two parameters are taken into account: $SD_{multiplier}$ and B_{size}. When use of a dictionary is prolonged, the $SD_{multiplier}$ and B_{size} parameters are increased and taken into account for the next evaluation. It could be that an increase of the dictionary size or sampling window size result in greater bandwidth reduction. Every time the configuration of a dictionary is changed, the counter *adaptations* is increased. After a certain amount of tries with increasing variations, B_{size} is no longer increased since the computational costs become too high. This parameter should instead be chosen according to the average notification size.

Expiry time: Equation 4 shows how the expiry time of a dictionary is calculated. A dictionary should at least amortize 10×, this value being chosen experimentally. Additionally, we added a factor based on how often different parameters of the dictionary were chosen. Each time the parameters of the dictionary are changed, the adaptation counter is increased. The computational cost of the dictionary sampling grows with increasing B_{size} and the compression cost of the publishers increase when the $SD_{multiplier}$ is increased.

$$T_{exp} = T_{amortize} * adaptations^3 * 10 \tag{4}$$

5. EVALUATION

We present three experiments. First, we evaluate the compression potential and computational cost of SSPS using varying permutations to find the best configuration. Then, we present an evaluation of the adaptive dictionary maintenance algorithm and discuss the trade-off between the computational cost and bandwidth reduction in view of the practical limits assessed in the first experiment. Finally, we present a practical use case evaluation using an experimental implementation on top of MQTT [5].

5.1 Datasets

Compression performance depends on the redundancy within a notification and between notifications. For that purpose, we took several real world datasets with varying degrees of redundancy. The DEBS15 dataset contains taxi trips from New York. The original dataset is published in CSV. To study the effect of different formats we converted the notifications into JSON, XML and Google Protobuf. Google Protobuf [3] generates an efficient binary notification based on a field description. The Twitter dataset is acquired from the Twitter API and contains Tweets including metadata from New York. We extracted only the content of the tweets to measure the compression performance on a highly variable text without any schema overhead. The Air Quality dataset comes from the NYC Open Data Portal [6]. This is used as an example of an IoT dataset, which includes many sensor readings. The EPEX dataset is extracted from the energy spot market auctions from the European Power Exchange. We use this dataset to test our solution with financial data, which mostly contains numbers and some repetitive information like contract types.

5.2 Compression potential using SDC

We emulated the connection of a publisher to the broker and evaluated multiple configuration dimensions. The size of the buffer B_{size} is based on how many preceding notifications the dictionary has sampled. The window W_{size} denotes how often the buffer is sampled. The maximum size of the

dictionary is set to a multiple of the average notification size. Equations 5,6 show the permutations. As an example, the configuration W_{300}, B_{100}, SD_2 means that every 300 notifications, a dictionary is built using the last 100 notifications and the dictionary size is limited to 2× the average notification size. At least 20k notifications are employed from each dataset.

$$size = \{50, 100, 200, 300, 500, 800, 1300, 2100, 3400, 5500\} \quad (5)$$

$$multiplier = \{0.3, 0.5, 1, 2, 3, 5, 8, 13, 21\} \quad (6)$$

The evaluations are conducted on a machine with 4×Intel Xeon CPU E7-4850 v3 @ 2.20GHz.

The emulation uses the library FemtoZip [2]. FemtoZip is a SDC library which can be used to build dictionaries. We used the class `FemtoZipCompressionModel` to generate the dictionaries from the notifications. The compression model creates also an optimal Huffman tree. The dictionary size is the sum of the size of both components. The dictionary part is limited to a multiple of the average notification size, see Equation 6.

The heatmaps in Figure 2 show the sweet spot of each dataset when the update frequency of the dictionary is set to 5500. The y-axis shows the dictionary multiplier, the x-axis the buffer size. The overall tendency is that bandwidth savings increase as more notifications are sampled. However after a certain size, it does not make sense to further grow the buffer. The same applies to the dictionary. While in most cases the best bandwidth savings are achieved with the biggest dictionary sizes, at a certain threshold the gain in bandwidth savings are not substantial.

Table 1 shows the best configuration for SDC. Column *Mean size* is the mean size of the notifications. Column *Best B/W* denotes how long the buffer should be and after how many notifications the dictionary should be refreshed. Column *SD size* shows how many additional bytes the average overhead per message is and column *SD OH* shows the introduced overhead in average per notification. The columns *Deflate (size)* and *SDC (size)* show how big in average the message is including the average overhead of the protocol. The last two columns show how big the bandwidth savings are. The overhead is largely introduced by the dictionary. Note that we added one byte per message for the protocol overhead to denote if the message is uncompressed or which dictionary identifier is used. The Deflate evaluation has no dictionaries and thus does not introduce any protocol overhead to the notifications.

An interesting observation is that there is nearly no difference in the bandwidth usage between XML, JSON and Protobuf when SDC is used. This is not the case when Deflate is used. The schema overhead and common attribute combination are promoted in the shared dictionary, which is similar in all schema variations. Therefore, the developer can choose the notification format which is most convenient without worrying about data size.

The overall tendency in our evaluation shows that it makes sense to sample bigger dictionaries and exchange them less frequently. The best dictionary multiplier in the EPEX Spot market data stream is only 3. This has to do that there is a limited amount of variation in the text properties, while the rest of the properties is numerical.

Figure 3 summarizes the performance comparison between SDC and Deflate. One observation is that small notifications are less compressible using Deflate, since there is less repetition within a notification. Tweets also have less repetition within a single tweet, hence the Deflate performance is low. Using SDC, where common words and tags are promoted to the SD, the bandwidth savings are higher.

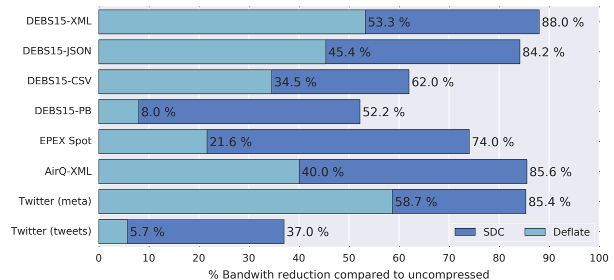

Figure 3: % bandwidth reduction incl. overhead

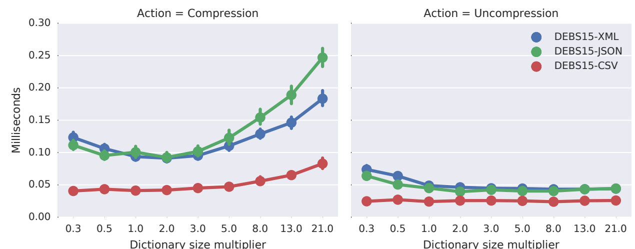

Figure 4: Compression/Uncompression time

5.3 Computational costs of SDC

In the emulation experiment, we record wall clock time of the individual operations. Our experiment employs one fixed thread per core. Figure 4 shows that the time to compress a single message is linear to the size of the dictionary. Note the near-exponential scale on the x-axis. The uncompression process is performed in constant time. Figure 2 shows that at a certain size of the dictionary not a lot more bandwidth reductions manifest. Additionally, the computational power needed to compress messages grows. Hence it makes sense that the adaptive algorithm probes if a bigger dictionary pays off instead of defaulting to a bigger $SD_{multiplier}$.

5.4 Adaptive algorithm

We tested the adaptive algorithm in an emulation that publishes at a fixed rate of 100 notifications/sec. We simulated the DEBS data stream over the first 100k notifications.

Figure 5 shows the behavior of the algorithm over time. When a dictionary expires, a decision is made to either create a **New SD** or to **Prolong SD**, shown by the blue and green dots. The decision is based on the **Current Savings** and the estimated **New Savings**. The line **Total BW reduction** shows the bandwidth reduction over time taking into account the introduced overhead. The initial dictionary holds for more than 50k notifications and not enough new savings could be acquired to justify spreading a new

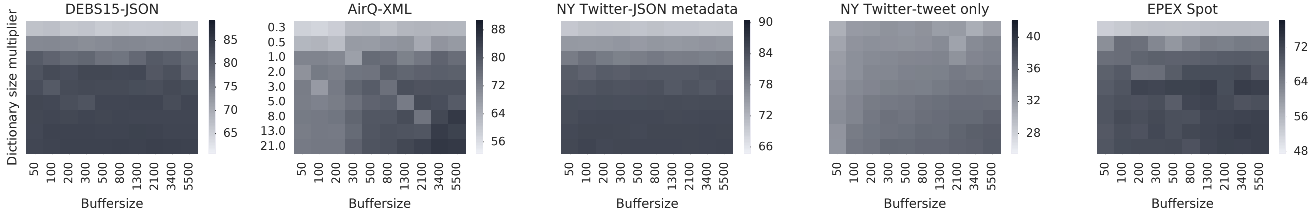

Figure 2: Buffer size vs. dictionary multiplier, update freq. set to 5500

Dataset	Mean size	Best $B/W/M$	SD size	SD OH	Deflate (size)	SDC (size)	Deflate (%)	SDC (%)
DEBS15-XML	711.41	2100/5500/21	22460.0	5.08	332.36	85.18	53.3	88.0
DEBS15-JSON	530.41	2100/5500/21	18659.0	4.39	289.68	83.99	45.4	84.2
DEBS15-CSV	191.41	500/5500/21	11540.0	3.1	125.2	72.75	34.5	62.0
DEBS15-PB	174.96	500/5500/21	11183.0	3.03	161.02	83.64	8.0	52.2
EPEX Spot	94.72	1300/3400/3	7810.0	3.3	73.79	24.59	21.6	74.0
AirQ-XML	536.53	3400/5500/21	18798.0	4.42	321.52	77.43	40.0	85.6
Twitter (meta)	2995.74	100/5500/21	69739.0	13.68	1241.52	438.74	58.7	85.4
Twitter (tweets)	84.87	5500/5500/21	9304.0	2.69	80.19	53.44	5.7	37.0

Table 1: Shared Dictionary compression vs. Deflate

dictionary. After 50k of notifications the content of the notifications changes which causes a drop of the bandwidth savings with the dictionary. The adaptive algorithm reacts by creating a new dictionary and changing the configuration. Alternating once the $SD_{multiplier}$ and once the B_{size} is increased.

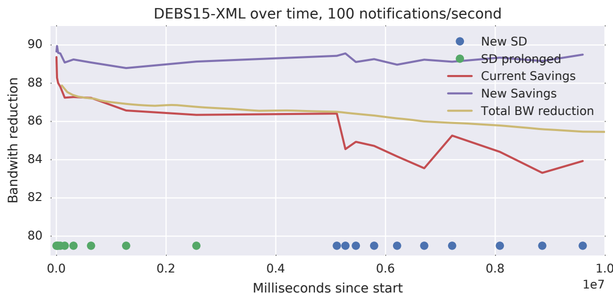

Figure 5: Adaptive algorithm over time

The adaptive algorithm could not reach the optimal results we have shown in the evaluation of the compression potential (see Table 2). While the JSON and XML variants are roughly $\approx 4\%$ worse than the best parameter combination, the CSV variant is more than $\approx 20\%$ worse. The adaptive algorithm starts without knowing what the best parameter combination is and over time tries to achieve higher bandwidth savings. Every time a dictionary is sampled, a computational cost is incurred, hence it is not feasible to try out too many combinations for the sampling broker. Nevertheless in all three cases the results are better than Deflate.

5.5 MQTT experiment

The prototype is implemented on top of MQTT [5]. An existing MQTT-compliant broker [4] is extended to handle the sampling broker role. We measure the throughput in a real-world environment which includes a hosted server at a provider, an Android application as a subscriber and another server as publisher.

Dataset	Best in eval.	Deflate	Adaptive
DEBS15-XML	88.0 %	53.3 %	85.46 %
DEBS15-JSON	84.2 %	45.4 %	80.17 %
DEBS15-CSV	62.0 %	34.5 %	41.65 %

Table 2: Adaptive algorithm

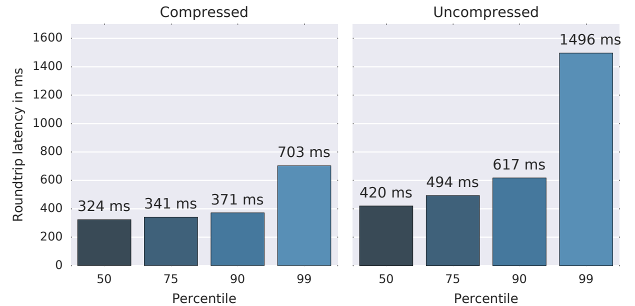

Figure 6: Latency

The Android phone is switched to 2G Mode. The signal quality is between -91 and $-82dBm$, which explains the flakiness of the connection and of our results. Ping latency is between $290-650ms$, which models our motivating rural scenario.

Figure 6 shows end to end latency at percentile in this experiment. Every 100 notifications, the subscriber responds to a notification by publishing on a separate topic. The subscriber receives this notification and derives the latency. The results show a significant reduction in latency especially in the higher percentiles.

Figure 7 shows the throughput rate in our experiment. First, 5000 uncompressed notifications are sent with the time span between the first and last message recorded (see

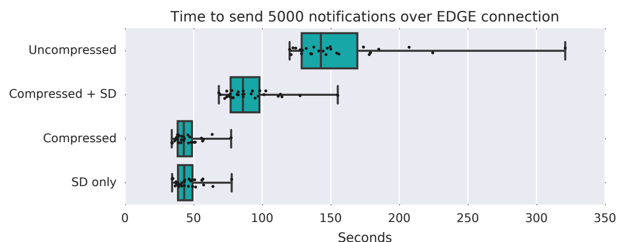

Figure 7: Throughput

bar **Uncompressed**). We then measured the time taken to send the dictionary (bar **SD only**). The dictionary multiplier is set to 1. We then send 5000 compressed notifications using the dictionary and also measured the time interval between the first and last message (bar **Compressed**). The bar **Compressed + SD** shows the time to transmit both the dictionary and 5000 compressed notifications. The chart shows that using SDC, ≈ 40% less time is needed to send 5000 notifications. Potentially, the dictionary can be used for another 5000 notifications which would amortize the dictionary overhead even more.

6. CONCLUSION AND FUTURE WORK

The evaluation has shown that by leveraging redundancy between notifications using a shared dictionary, additional bandwidth reductions can be achieved over traditional compression techniques.

SDC is especially beneficial in sensor networks or Internet of Things (IOT). Sensor events typically contain metadata like location, timestamp, units, identifiers and multiple values which are readings from the environment (e.g., temperature, humidity, etc.). In multiple consecutive notifications this metadata will be repeated and would be promoted to the dictionary and only be exchanged a single time. SDC can therefore compensate for the inefficiencies of sensor data.

The approach shows that it is possible to trade computational resources vs. bandwidth. While the available bandwidth for mobile phones has improved, computational power has improved even more: even low-end mobile phones contain four or more cores.

We see SSPS as a first step towards a dynamic cloud-based pub/sub system which supports SDC. Our current requirement for dedicated sampling brokers which manage dictionaries is promising, but is a simplistic assumption for the cloud.

Dictionary maintenance: Our current proposed algorithm assumes a fixed rate. To compensate for variations, it uses pessimistic assumptions about the amortization time of the dictionary. Ideally, the algorithm should publish a new dictionary when the rate is low.

Dictionary sampling: In content-based pub/sub, publications are matched based on subscription predicates. It may therefore be beneficial to maintain several active dictionaries for the same topic, depending on the nature of the content-based subscriptions.

Scheduling of dictionary sampling: In large-scale pub/sub systems, thousands of publishers and subscribers exchange notifications. Dictionary sampling becomes a non-negligible computational cost, which has to be scheduled accordingly. A scheduler process can take into account the dictionary expiry as a deadline and plan upfront which dictionaries to sample. The sampling cost is a factor of notification size and size of the buffer and can be accurately estimated up front. The algorithm could be optimized for minimal bandwidth usage of mobile subscribers.

Content-based pub/sub matching: Content-based pub/sub could be extended to match notifications without uncompressing. By taking the schema into account when creating the dictionary, we could already match the dictionary entries. Using this information, it is possible to skip uncompression and match directly on the compressed data.

Overlay bandwidth: SSPS introduces additional CPU load to publishers and subscribers because the notifications have to be compressed and uncompressed. Another variant of the solution could acknowledge that and only reduce the bandwidth between the brokers. Using this approach data center replication could be made more efficient without affecting computational performance.

Combination with databases: Databases use already dictionary based compression techniques to reduce disk space or improve latency. Notifications are often stored in column stores for further analysis. Ideally the pub/sub can be combined with the database and the dictionary could be reused in the column store to reduce the computational load.

Iterative sampling: The algorithm behind SD sampling currently works in batch mode. This means that every amount of time, a new dictionary has to be sampled from scratch. A incremental algorithm, potentially building upon approximate data structures like Bloom filters, can make the batch dictionary generation unnecessary and hence reduce load spikes.

7. ACKNOWLEDGEMENTS

This work was supported by the Alexander von Humboldt foundation.

8. REFERENCES

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

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

[3] Google Protocol Buffers. https://developers.google.com/protocol-buffers/.

[4] Moquette. https://github.com/andsel/moquette.

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

[6] NYC OpenData. https://data.cityofnewyork.us/.

[7] RFC1951 - DEFLATE Compressed Data Format Specification version 1.3. https://tools.ietf.org/html/rfc1951.

[8] RFC1952 - GZIP file format specification version 4.3. https://www.ietf.org/rfc/rfc1952.txt.

[9] Shared Dictionary Compression for HTTP at LinkedIn. https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin.

[10] J. Alakuijala and Z. Szabadka. Draft: Brotli Compressed Data Format. http://www.ietf.org/id/draft-alakuijala-brotli-08.txt, 2015.

[11] J. Bentley and D. McIlroy. Data compression using long common strings. In *Data Compression Conference, 1999. Proceedings. DCC '99*, pages 287–295, Mar 1999.

[12] J. Bentley and D. McIlroy. Data compression with long repeated strings. *Information Sciences*, 135(1âĂŞ2):1 – 11, 2001. Dictionary Based Compression.

[13] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 283–296, New York, NY, USA, 2009. ACM.

[14] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter. A Proposal for Shared Dictionary Compression over HTTP. https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf, 2008.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[16] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7(2):192–214, Apr. 1998.

[17] 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.

[18] 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.

[19] S. Ji, C. Ye, J. Wei, and H.-A. Jacobsen. MERC: Match at Edge and Route intra–Cluster for Content-based Publish/Subscribe Systems. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 13–24, New York, NY, USA, 2015. ACM.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[21] E. Skjervold and M. Skjegstad. Reap: Delta compression for publish/subscribe web services in manets. In *Military Communications Conference, MILCOM 2013 - 2013 IEEE*, pages 1488–1496, Nov 2013.

[22] H. White. Printed english compression by dictionary encoding. *Proceedings of the IEEE*, 55(3):390–396, March 1967.

[23] L. Xu, A. Pavlo, S. Sengupta, J. Li, and G. R. Ganger. Reducing replication bandwidth for distributed document databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 222–235, New York, NY, USA, 2015. ACM.