Exploiting Edge Computing for Adaptive Data Update in Internet of Things Networks

Andrea Petroni*
andrea.petroni@uniroma1.it

Andrea Lacava*
lacava.1663286@studenti.uniroma1.it

Pierluigi Locatelli*
locatelli.1668471@studenti.uniroma1.it

Gianluigi Nero*
nero.1657819@studenti.uniroma1.it

Marcello Pediconi**
marcello.pediconi@aenduo.com

Francesca Cuomo*
francesca.cuomo@uniroma1.it

* Department of Information Engineering, Electronics and Telecommunications
La Sapienza University of Rome, Italy
** Aenduo s.r.l., Rome, Italy

Abstract

With the advent of the pervasive Internet of Things (IoT) era it is expected to have billions of entities simultaneously connected to the network, sharing heterogeneous data to support disparate applications. Such scenario will therefore open new challenges as for network management and information exchange rules. In this context, the increasing data volume may especially lead Cloud-based services to be suffering from overload and data traffic consumption increase when serving a huge number of devices. A potential approach to address this problem is to edge computing, including all those enabling technologies able to move large part of computing close to the data sources, proving several benefits in terms of latency reduction, bandwidth optimization and security [RGXZ17][MTPC19]. Another aspect impacting the performance is the optimization of the amount of data volumes transmitted by the IoT devices. This task is accomplished by specific data synchronization protocols and algorithms that are responsible for information exchange between devices and cloud. In this direction, we consider a decentralized IoT cloud framework where devices connect to the data center through an IoT gateway. Moreover, we present a mechanism for data synchronization that considers *Octodiff*, a well known tool for data compression, combined with an adaptive algorithm specifically tailored to limited, variable, IoT traffic volumes. By investigating the performance of the proposed architecture, we show how the traffic amount generated by IoT cloud-services can be conveniently reduced.

1 Introduction

People's lives and habits will reach new breakthroughs thanks to the introduction of the Internet of Things (IoT). Being surrounded anytime and anywhere by billions of *things* communicating and sharing information with one another represents the latest step of the interaction between humans and technology.

Such a revolutionary paradigm involves very heterogeneous devices, from wearables, smartphones and computers associated to daily, personal activities, up to sensors and machinery systems employed for monitoring and control smart buildings, industry and cities [AGM^{+}15]. In this direction, as each IoT device output is characterized by specific structures and features, different kind of data are going to be simultaneously transmitted through the network. Some exam-

Editors: E. Calvanese Strinati, D. Charitos, I. Chatzigiannakis, P. Ciampolini, F. Cuomo, P. Di Lorenzo, D. Gavalas, S. Hanke, A. Komninos, G. Mylonas.
In: Proceeding of the Poster and Workshop Sessions of AmI-2019, the 2019 European Conference on Ambient Intelligence, Rome, Italy, 13-11-2019, published at http://ceur-ws.org.

ples are represented by the small size spot measures typically produced by sensors for environmental monitoring, otherwise high quality and large size videos and pictures shared between computers. Moreover, data traffic is driven by the specific application that it is related to, requiring appropriate quality of service. For instance, activities like video streaming need to be supported by high rates and low latency, while many others are oriented to energy saving, especially when battery power devices are involved. In general, taking into account all of these concerns is important in order to minimize the overall traffic load, but when dealing with the IoT it becomes essential as the presence of a large number of devices entails a significant growth of the amount of data volumes.

The efficient management of network resources is recognized to be very challenging, especially in cloud-based networks where the presence of a huge amount of devices simultaneously performing data synchronization may cause bandwidth saturation. Therefore, it follows that effective protocols for remote data update are necessary to minimize the amount of data exchanged through the network and, consequently, to optimize the overall traffic load. Moreover, the reduction of information to be transmitted returns also to energy saving for all that IoT devices that are typically battery powered.

Unfortunately, data synchronization algorithms originally developed for computer systems badly fit to the IoT since they are not able to handle heterogeneous traffic volumes and their performance do not consider device power consumption. An example is given by the *rsync* algorithm [TM96] representing the core of many, well known, cloud services like Dropbox. Given two parties each one storing different version of the same file (an old one and a newer one), the *rsync* algorithm performs a single round synchronization by recognizing the matching parts between the two file versions, so that the information to be exchanged is just that one necessary to make the old file version updated. In general, good performance are obtained when the size of files under processing is quite large, on the other hand, the algorithm suffers from inefficiency when dealing with small amount of data. So, *rsync* may not represent the best solution in the IoT case where few data are sporadically transmitted (for instance, sensors for environmental monitoring sending their spot measurements to the data center). Some other algorithms have been proposed in the literature with the aim to improve the performance of *rsync*, such as that one presented in [YIS08] where reconciliation techniques are applied to minimize the bandwidth consumption. Other solutions consider instead multiple round procedures in order to optimize the generated traffic amount like in [SNT04], but unfortunately undesired latency is introduced due to the bi-directional communications occurring between the two parties involved in the synchronization.

More recently, the problem of data update has been addressed with a view to IoT scenarios and applications. In this context, the heterogeneity and variability of IoT data are addressed in [ZL16] by proposing a remote synchronization based on timestamp and bitmap to improve the transmission efficiency. Furthermore, the work in [PCS$^+$18] introduces an *rsync*-inspired algorithm where the knowledge of data characteristics are exploited to adaptively tune the synchronization parameters and optimize the amount of information to be transmitted.

Finally, the architecture itself characterizing the cloud services may represent a limit for the performance in the IoT context. Indeed, devices at the edge of the network may be very far from the datacenter node, so the process of remote data synchronization could suffer from long latency, bandwidth inefficiency, bottlenecks and excessive device power consumption. A potential solution to all those issues is the edge computing [SCZ$^+$16] that envisages the development of a de-centralized structure where many computing tasks are moved at the network edge, so closer to the IoT devices. The exploitation of this novel approach is described in [WZL$^+$19] where a fog computing-based technique is proposed for data synchronization in the IoT.

Following this direction, on the basing of principles of edge computing, we investigate a cloud architecture where devices are connected to an IoT gateway, placed at the network edge, and performing remote files synchronization by interacting with the main data center. Specifically, we consider a data synchronization mechanism given by the combination of *Octodiff* with a rephrased version of the adaptive algorithm AC-*rdiff* introduced in [PCS$^+$18], in order to evaluate the benefits of the presented solution on the network traffic load and on the reduction of the processing time requested by the synchronization procedure.

The paper is organized as follows. In Sec. 2 the network scenario is introduced, reporting the essentials of *Octodiff* and AC-*rdiff*. The architecture and features of the proposed file synchronization mechanism are detailed in Sec. 3. Sec. 4 reports some numerical results, and finally the conclusion is drawn in Sec. 5.

2 Reference framework

Let us refer to an IoT scenario where multiple devices (sensors, vehicles, machinery and so forth) periodically access the network to store their data and measurements on a cloud server. As previously mentioned, the potentially large number of entities simultaneously in-

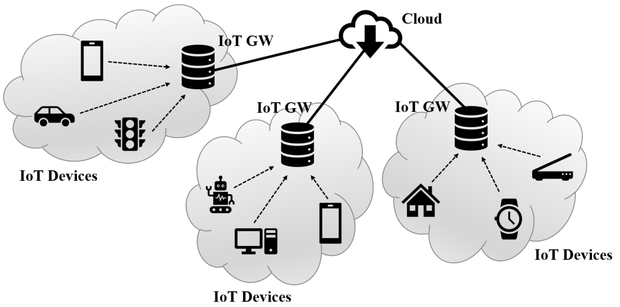

Figure 1: Reference cloud network architecture.

teracting with the cloud may create a bottleneck at the server node, so we consider a de-centralized architecture like that one described in Fig. 1 where many IoT gateways (IoT GWs) placed at the network edge are responsible for data collection and update. The proposed scenario is realized on the basis of the principles of edge computing, where data processing is performed near the devices, reducing latency and providing energy saving to battery powered IoT entities. Therefore, each IoT GW is supposed to have a storage capacity and computing power, so that data synchronization can be operated minimizing the interaction with the cloud central node (communications for network control and management may be considered anyway). Giving further details about the communication protocols involving IoT GWs and the central node goes behind the scope of this work, however we were inspired by a medical IoT application were medical devices communicate with short range links to a GW that on its turn is interconnected to the Internet. We focus the attention on remote data synchronization aspects. In this direction, we report below the essentials of *Octodiff* and the adaptive algorithm in [PCS+18] that are combined to implement the synchronization mechanism.

2.1 Octodiff

Octodiff is an implementation of remote delta compression developed by Microsoft. It relies on the *rdiff* algorithm that is a particular version of *rsync* [TM96]. By referring to Fig. 2, *Octodiff* generates a *delta* file containing all the information necessary to update the old file version stored at server side up to the latest one provided at the client side. By identifying the IoT GW as the client and the cloud as the server (Fig. 2), the main steps of the procedure are summarized as follows (see [Gitff] for further details):

1. The cloud organizes its old version of the file F_{OLD} in blocks of size equal to d_{c}, referred as *chunks* in the rest of the paper. Then, chunks are compressed, checksums (named signatures in the followings) are generated and a resulting list of signatures S_{OLD} are sent to the IoT GW.

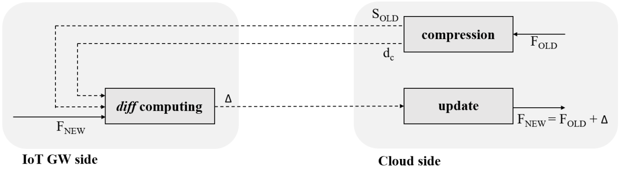

Figure 2: Octodiff scheme for remote file synchronization.

2. The IoT GW uses the list of signatures S_{OLD} to compute the differences between its latest file version F_{NEW} and F_{OLD} stored at cloud side. The output delta file Δ, containing the information about the differences, is transmitted to the cloud.

3. The cloud applies the changes reported in Δ to F_{OLD}, finally obtaining an exact copy of F_{NEW}.

It is worth highlighting that transmitting the signatures S_{OLD} (generated from an Adler32-based rolling checksum and a hash functions SHA1) instead of the entire file basis F_{OLD} brings benefits in terms of bandwidth saving. A similar optimization is given by transferring the smaller file Δ in place of the whole F_{NEW}. However, the performance of *Octodiff* strictly depends on a convenient choice of d_{c}, that is the parameter that drives not only the split of F_{OLD}, but also the differences computation at the IoT GW side. In fact, if the dimension of d_{c} is not sufficiently smaller than the entire file size, the processing at IoT GW side may generate a file Δ containing overhead information that is unnecessary to the cloud for file update. So, additional and undesired data traffic is generated. In *Octodiff* the chunk size is set by default to 2048 *bytes*, even though values from 128 to 31744 *bytes* are accepted. The analysis of the original *rsync* reported in [TM96] identifies the optimal value for the chunk size as equal to 512 or 1024 *bytes*.

2.2 Adaptive data synchronization algorithm

All the *rsync*-based algorithms are known to be very efficient especially when processing rather large size data (significantly larger than the chunk size). On the other hand, as previously outlined, providing good performance becomes harder when small files are handled. In fact, the *rsync* algorithm works according to a fixed chunk size, the value of which may be sometimes inappropriate especially when dealing with heterogeneous data. This occurrence is typical in the IoT, so a *static* approach to data synchronization is ineffective. In this direction, the algorithm AC-*rdiff* introduced in [PCS+18] shows how the chunk size can be conveniently tuned in order to match the characteristics of the file under processing. Specifically, the algorithm considers the following steps:

1. The differences between two file versions F_{NEW} and F_{OLD} are computed, generating the output file Δ as performed in *rsync* (and, therefore, as in *Octodiff*) to be transmitted from the client to the server for completing the update.

2. By analyzing the file Δ it is possible to infer information about the portions of F_{NEW} and F_{OLD} that are matching. So, basing on the distribution of the matching blocks along the file, a new optimal chunk size is estimated to be used in the next synchronization procedures.

More in detail, the file Δ is composed of *i)* a sequence of literal bytes, that is all the information recognized as new and that have to be necessarily sent to the server for its file version update, *ii)* a list of *tokens*, that is the indexes referring to the position of matching blocks within the file. The set of tokens represents the input for the algorithm responsible for the chunk size optimization. By analyzing the list of tokens, a rough estimate of the matching blocks spatial distribution is obtained. So, loosely speaking, the presence of matching blocks placed one next to the other leads the chunk size to be incremented. On the other hand, a sparse distribution of matching blocks makes the chunk size basically reduced. Finally, the new estimated chunk size is saved in the header of the file Δ and transmitted to the cloud in order to be set up for the next synchronization procedure.

So, the parameters ruling the synchronization mechanism are tuned basing on the characteristics and spatial distribution of changes occurring in the file. Preliminary results have shown the effectiveness of this approach especially when dealing with data of a size in the order of few kilobytes (that case may represent a typical output of an IoT sensor for environmental monitoring).

3 Adaptive Octodiff synchronization

Interestingly, the chunk size adaptation rule driving AC-*rdiff* can be easily combined with any synchronization mechanism relying on the principles of *rsync*. In this direction, the study here proposed is on the integration of AC-*rdiff*, conveniently modified, with *Octodiff* tool.

First, it is worth highlighting that AC-*rdiff* is performed on the current file update procedure, and returns an optimized parameter (the chunk size) to be applied in the next synchronization event. So, the processing performed at the IoT GW side considers two steps, that is the computation of files differences by generating the file Δ and then the chunk size optimization. A reference architecture describing this mechanism is essentially given by the cascade of the block for delta computing (named *diff computing*) and the block for chunk size optimization, as depicted in Fig. 3. The processing operated by the cloud, that is file compression and update, essentially is the same as in the original *Octodiff*. Given the storage and processing capacity supposed for the IoT GW we assume that this node as able to store both F_{NEW} and F_{OLD} in its memory. So, the file Δ can be computed internally, that is without the need to receive the old file version reference from the cloud. In this way, the downlink communication from the cloud to the IoT GW could be significantly reduced, saving therefore network traffic. The cloud would always deal with the file update operations, while data compression and transmission would only concern sporadic cases like for example failure events where the connection with the IoT GW needs to be restored. The computing requested by the IoT GW is instead summarized in the following steps. The synchronization starts when the IoT GW collects all the information related to F_{NEW} and F_{OLD}. The differences between the two files are processed and the corresponding output file Δ is generated. The adaptive chunking is then performed using the information (the tokens list) stored in Δ, returning the new chunk size estimate d_{opt}. Finally, both the file Δ and d_{opt} are sent to the cloud to let it perform its file version update. It is worth highlighting that the cloud must receive from the IoT GW the information about d_{opt} so that it can correctly set up the parameters for the next synchronization procedure.

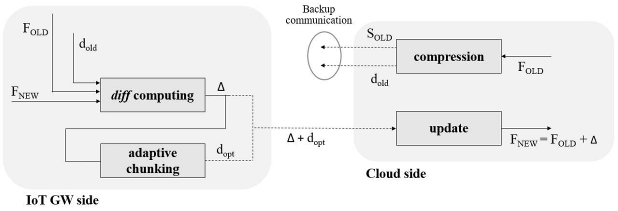

Figure 3: Proposed scheme for remote file synchronization.

To implement the AC-*rdiff* in the *Octodiff* framework the algorithm has been rephrased in some parts, with the corresponding pseudocode being detailed below. Specifically, the input is represented by the current chunk size and a vector gathering the token indexes, d_{old} and $\mathcal{P}_{\mathrm{t}}$ respectively. By computing the distance between consecutive token indexes (row 10) the distribution of matching chunks along the file is inferred and multiple partial estimates of the optimal chunk size are obtained (row 13-18). The new chunk d_{opt} is returned by the average on the partial esti-

Algorithm 1 AC-*rdiff* Algorithm

```
Input: $\mathcal{P}_{\text{t}}$ Token index vector; $d_{\text{old}}$ current chunk size
if size($\mathcal{P}_{\text{t}}$) $\leq 1$ then
    $d_{\text{opt}} := d_{\text{old}}$
else
    $W_m :=$ [] Chunk size partial estimates vector
    count := 0 Chunk estimates counter
    $N_{ac} := 1$ Adjacent matching chunks counter
    $N_{ud} := 0$ Expected updates counter
    for m = 1:M-1
        $\Phi_m := \mathcal{P}_{\text{t}}(m) - \mathcal{P}_{\text{t}}(m-1)$
        if $\Phi_m == d_{\text{old}}$ then
            $N_{ac} := N_{ac} + 1$
            $W_m(\text{count}) := d_{\text{old}} + \mu_{\text{UP}} N_{ac}$
        else
            count := count + 1;
            $N_{ac} := 1$
            $N_{ud} := \lceil \frac{\Phi_m}{d_{\text{old}}} - 1 \rceil$
            $W_m(\text{count}) := d_{\text{old}}$ - $\mu_{\text{DOWN}} N_{ud}$
            count := count + 1;
            $N_{ud} := 0$
        end
    end
    $W_m(W_m \geq 2d_{\text{old}}) := 2d_{\text{old}}$
    $W_m(W_m \leq 0.5d_{\text{old}}) := 0.5d_{\text{old}}$
    $d_{\text{opt}} := \dfrac{\text{sum}(W_m)}{\text{size}(W_m)}$
end
return $d_{\text{opt}}$
```

mates collected in W_m. The speed of adaptation of the new chunk size is ruled by two step sizes, namely μ_{UP} and μ_{DOWN}. However, in order to avoid excessive variations of the output, d_{opt} is allowed to range from half the old chunk size d_{old} and 2 times d_{old}. Furthermore, as described in the previous section, the chunk size driving *Octodiff* can assume values from 128 to 31744 *bytes*, so the output of AC-*rdiff* is anyway forced to meet this constraint even when the actually estimated d_{opt} is out of the considered range.

4 Numerical results

The effectiveness of remote synchronization algorithms strictly depends on the characteristics of data under processing and on the changes that they may be subject to. It is worth noting that the measurements produced by devices are typically gathered into files characterized by an header and a payload or, in more specific cases, into structured formats like for instance JSON and XML. The knowledge of how the information to be transmitted is organized could be helpful in order to conveniently set up the synchronization mechanism, however the heterogeneity of data and devices in the IoT may not always allow the achievement of optimal performance.

In this direction, we have investigated the efficiency of the proposed adaptive chunking based *Octodiff*, referred as A-*Octodiff* in the rest of the paper, comparing its performance with those provided by the classical *Octodiff* working with static parameters, referred as S-*Octodiff*. Specifically, by running the remote file synchronization on simulated data, we show how the flexibility characterizing A-*Octodiff* results to be useful for optimizing of the network traffic load, but also for reducing the processing time.

We preliminary considered the remote file synchronization as run on simulated files, the dimension of which has been initially chosen to be about 50 kB. Such a value is in line with the typical output generated by some IoT devices. Specifically, the performance of A-*Octodiff* and S-*Octodiff* have been measured in terms of generated traffic percentage, given by:

$$T_{\%} = \frac{\dim(\Delta)}{\dim(F_{NEW})} \times 100$$

that is the ratio between the amount of data actually transmitted (the file Δ) and the dimension of the entire latest file version (the file F_{NEW}). The results come from the average on 100 simulations (that is, 100 file synchronization procedures), considering different file update percentages expressing how much of the current file has changed with respect to its previous version.

The spatial distribution of updates within the file have been first modeled as uniform and organized in blocks of variable size (from tens to hundreds of bytes). This kind of distribution represents the worst case for S-*Octodiff* and, in general, for all the synchronization mechanisms working with static parameters since the updates essentially occur randomly in the file. The synchronization mechanisms have been implemented with different starting chunk sizes, but while these values remain static in S-*Octodiff*, the adaptation provided by A-*Octodiff* make them change as the simulations go on. S-*Octodiff* considers by default the chunk size as equal to 2048 *bytes*. By looking at the results reported in Fig. 4 it is possible to appreciate how the use of such static value provides bad performance as the generated traffic percentage is in the order of 35%, increasing up to the 50% as the file update percentage grows (error bars referring to a 95% confidence interval are highlighted for all the measurements). The reason is that the chunk size is too large with respect to the file size, therefore it is very hard to recognize matching blocks of bytes. On the other hand, with A-*Octodiff*, by adapting the chunk size to an optimal value, the amount of generated traffic results to be significantly

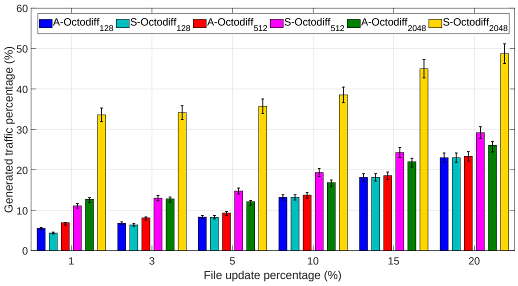

Figure 4: Performance of different remote synchronization mechanisms on 50 kB files (randomly distributed updates).

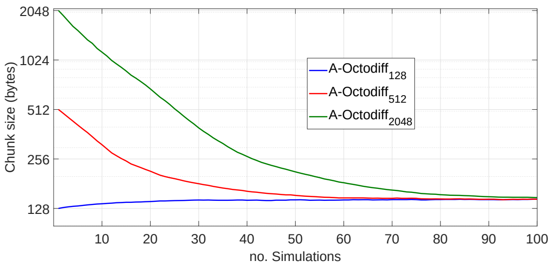

Figure 5: Chunk size adaptation during 50 kB files update (randomly distributed updates).

reduced down to the 15%-25%. The same simulations have been run considering a starting chunk size equal to 512 bytes that has resulted to be more suited to the considered scenario, in fact S-*Octodiff* returns an average traffic percentage ranging from 13% to 30%. However, the adaptive mechanism driving A-*Octodiff* allows the achievement of even better performance, as the generated traffic percentage is reduced by the 6%-7% with respect to S-*Octodiff*. Finally, the remote file synchronization has been performed using the minimum chunk size allowed by *Octodiff*, that is 128 bytes. In this case A-*Octodiff* provides essentially the same performance of A-*Octodiff* using a starting chunk size equal to 512 bytes. This fact can be explained by looking at Fig. 5 describing the adaption of the chunk size as the simulations proceed (the values on the vertical axis are reported on a logarithmic scale for convenience). Specifically, it is possible to observe that when the initial chunk size is 512 bytes the value tends to decrease (the same happens for the chunk size initially equal to 2048 bytes), while if the starting chunk size is 128 byte, the adaptation mechanism returns increasing values. Therefore, as the chunk sizes tend to converge, it is reasonable to measure the same performance. Furthermore, we observe that, since the optimal chunk size value equal to 150 bytes is quite close to the minimum one allowed, S-*Octodiff* and A-*Octodiff* return the same average traffic percentage when a 128 bytes initial chunk size is considered.

$R_{T_{\mathrm{p}}}$	A-*Octodiff*$_{128}$	A-*Octodiff*$_{512}$	A-*Octodiff*$_{2048}$
S-*Octodiff*$_{128}$	1.15	0.91	0.98
S-*Octodiff*$_{512}$	1.32	1.05	1.12
S-*Octodiff*$_{2048}$	0.67	0.53	0.56

Table 1: Time processing ratio measures (50 kB files, randomly distributed updates).

Finally, it is worth noting that when the differences between two files are computed returning the list of matching blocks, the *Octodiff* tool is able to compress the corresponding tokens in a single interval if they are listed in ascending order, reducing the information to be sent. So, using a small chunk size may seem always convenient since by doing so, even if the number of generated tokens is higher than the case when a larger chunk size is used, they can be however compressed . So this is the reason why *Octodiff* and A-*Octodiff* return the same generated traffic percentage. However, in general the time for file processing and difference grows increases as the chunk size decreases.

The benefits coming from the use of an adaptive approach with respect to the static one have been also evaluated by comparing the time required for processing. Of course the performance depend on the specific hardware employed to run the synchronization procedure, so we have introduced the following metric:

$$R_{T_{\mathrm{p}}} = \frac{T_{\mathrm{p,adaptive}}}{T_{\mathrm{p,static}}} \tag{1}$$

that is the ratio between the average processing time referring to the adaptive mechanism and the average processing time for the static one. By doing so, the performance comparison between adaptive and static mechanism can be expressed independently from the hardware used for tests. In this direction, Tab. 1 reports the results of $R_{T_{\mathrm{p}}}$ measured combining the A-*Octodiff* and S-*Octodiff* implementations. Specifically, having values less than 1 means that the adaptive mechanism is less time consuming than the static one. On the other hand, when values exceed 1, A-*Octodiff* requests more time than S-*Octodiff*. However, this latter results is not always negative. In fact, for example, even though A-*Octodiff*$_{512}$ is slower than S-*Octodiff*$_{512}$, the use of the adaptive approach provide a significant traffic saving with respect to the static mechanism (Fig. 4). Moreover it is also shown that, in general, the use of a small chunk size leads to the recognition of a larger number of matching blocks, but it makes the difference computing processing longer.

In order to highlight the performance of the considered synchronization mechanisms and the importance of using a proper chunk size, we have performed other simulations considering smaller files, with size equal to 5 kB. By referring to Fig. 6 we observe how using

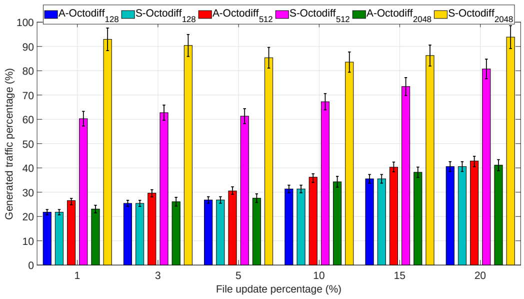

Figure 6: Performance of different remote synchronization mechanisms on 5 kB files (randomly distributed updates).

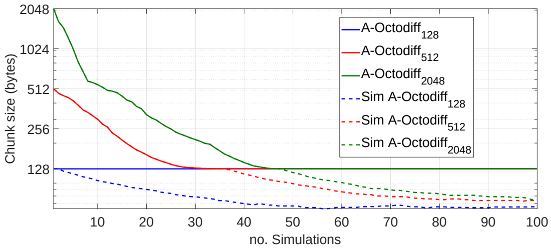

Figure 7: Chunk size adaptation during 5 kB files update (randomly distributed updates).

a 2048 bytes chunk size results to be unsuitable, with very low performance provided by S-*Octodiff*. Significant improvements are instead obtained when considering A-*Octodiff*. Actually, the chunk size equal to 512 bytes is recognized to be too large as well, in fact traffic percentage generated by S-*Octodiff* is quite high, around the 70%. In fact, the curves in Fig. 7 show how the optimal chunk size is essentially the minimum one allowed by *Octodiff*. So, when dealing with very small size data it is convenient to reduce the chunk size, as demonstrated by the results referring to S-*Octodiff* and A-*Octodiff* implemented with an initial chunk size equal to 128 bytes. Moreover, the advantage given by an adaptive approach can be once more appreciated by observing that the performance of A-*Octodiff* implemented with a 512 bytes initial chunk size approaches the one provided using a smaller chunk size. Interestingly, by referring Fig. 7, the adaptive chunking seems not have effects when A-*Octodiff* is implemented according to a 128 bytes initial chunk size, since no changes are observed. At first glance, it would be inferred that the optimal chunk size is exactly 128 bytes. However, taking into account the constraint about the minimum chunk size characterizing *Octodiff*, the optimal chunk size could be actually less than 128 bytes as well. So, in order to investigate this issue, we simulated the behavior of *Octodiff* in Matlab removing the limit about the allowed chunk size, and we found out that the optimal value would be around 70 bytes, as demonstrated by the dashed curves reported in Fig. 7. Finally, Tab. 2 reports results referring to the time processing ratio between adaptive and static mechanisms, and it is possible to appreciate that the processing time of A-*Octodiff* is always less or at most equal to the processing time characterizing A-*Octodiff*. So, in this context, we can conclude that when the dimension of files is in the order of few kilobytes, the use of *Octodiff* tool may not be convenient for data synchronization since the constraint about the minimum value of the chunk size could not allow the achievement of best performance in terms of traffic saving.

R_{T_p}	A-*Octodiff*$_{128}$	A-*Octodiff*$_{512}$	A-*Octodiff*$_{2048}$
S-*Octodiff*$_{128}$	1	1	1
S-*Octodiff*$_{512}$	0.62	0.65	0.63
S-*Octodiff*$_{2048}$	0.43	0.45	0.44

Table 2: Time processing ratio measures (5 kB files, randomly distributed updates).

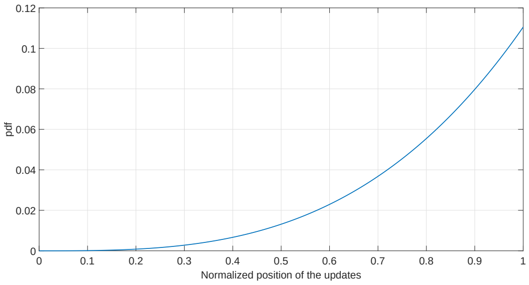

Figure 8: File updates with exponential probability density function.

The second scenario under investigation has considered the file synchronization performed on input files where the updates distribution was modeled according to the following exponential probability density function:

$$f(x;\lambda) = \lambda x^3 e^{\lambda x} \qquad 0 < x \leq 1$$

with $\lambda = 0.1$ and the random variable x, representing the index (position) of new bytes in the original file, normalized to 1 (Fig. 8). By doing so, we refer to the possibility of having updates occurring at the end of the file more frequently than at the beginning, because the more x is close to 1 the more the updates are located in the final part of the file. Figs. 9-10 reports the results referring to the remote synchronization of 50 kB and 5 kB files, respectively. In general, we observe the better results with respect to Figs. 4-6 since now the updates are no more randomly distributed, but concentrated at the end of the file. So it is reasonable to find a higher number of matching blocks, resulting in a lower generated traffic amount. However, the

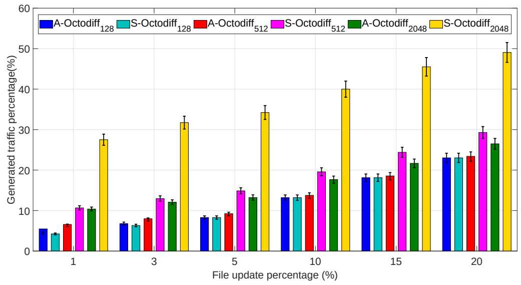

Figure 9: Performance of different remote synchronization mechanisms on 50 kB files (exponentially distributed updates).

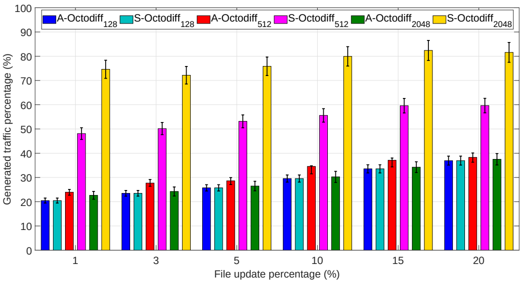

Figure 10: Performance of different remote synchronization mechanisms on 5 kB files (exponentially distributed updates).

comparison between S-*Octodiff* and A-*Octodiff* implemented with different chunk sizes highlights the same trend, thus showing the effectiveness of the proposed adaptive approach even in this scenario. For the sake of completeness, we also report in Figs. 11-12 the performance of A-*Octodiff* in terms of chunk size adaptation for both file size cases. The chunk size converges towards quite the same value reached in Figs. 5-7, approximately around 70 bytes. Tabs. 3-4 report the results in terms of time processing ratio referring to the 50 kB and 5 kB files synchronization analysis, respectively. In that case we observe that the processing time referring to A-*Octodiff* is in general higher than the processing time characterizing S-*Octodiff*.

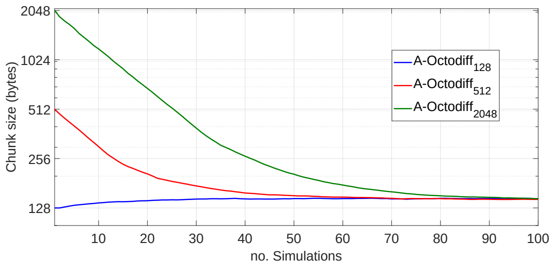

Figure 11: Chunk size adaptation during 50 kB files update (exponentially distributed updates).

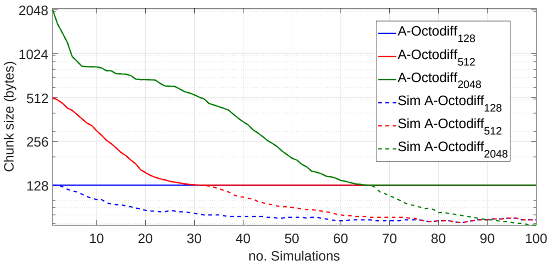

Figure 12: Chunk size adaptation during 5 kB files update (exponentially distributed updates).

R_{T_p}	A-*Octodiff*$_{128}$	A-*Octodiff*$_{512}$	A-*Octodiff*$_{2048}$
S-*Octodiff*$_{128}$	1.34	1.28	1.11
S-*Octodiff*$_{512}$	1.63	1.56	1.35
S-*Octodiff*$_{2048}$	0.71	0.68	0.69

Table 3: Time processing ratio measures (50 kB files, exponentially distributed updates).

R_{T_p}	A-*Octodiff*$_{128}$	A-*Octodiff*$_{512}$	A-*Octodiff*$_{2048}$
S-*Octodiff*$_{128}$	1.1	1	1
S-*Octodiff*$_{512}$	0.85	0.78	0.77
S-*Octodiff*$_{2048}$	0.46	0.42	0.41

Table 4: Time processing ratio measures (5 kB files, exponentially distributed updates).

However better performance are provided in terms of generated traffic percentage, as shown in Figs. 9-11.

Finally, we considered data synchronization involving file of variable size. Specifically, we run the synchronization mechanism on 100 files with sizes ranging from 40 kB to 80 kB. Each file is modified with respect to its previous version according to an update percentage randomly varying between 5% and 20%. The updates are represented by bytes addition, removal and substitution. The generated traffic percentage and the processing time ratio, averaged on 100 simulations, are reported in Tab. 5 as a function of the synchronization mechanism employed, that is static or adaptive, and of the chunk size. It is worth highlighting that the values referring to A-*Octodiff* are quite uniform, thus meaning that the use of the adaptive approach makes the performance less sensitive to the choice of the initial chunk size. On the other hand, when using a static mechanism, the set up of fixed, unsuitable chunk size may lead the generated traffic percentage to be higher, as in the case of S-*Octodiff*$_{2048}$. The effectiveness of A-*Octodiff* can be further appreciated by observing Fig. 13 reporting the chunk size adaptation along the 100 simulations. In fact, despite A-*Octodiff* is initialized with different values, the chunk size tends to adapt toward the same optimal value around 200 bytes.

	Av. Traffic Percentage	R_{T_p}
A-*Octodiff*$_{128}$	4.01%	0.68
S-*Octodiff*$_{128}$	3.52%	
A-*Octodiff*$_{512}$	4.03%	1.06
S-*Octodiff*$_{512}$	6.01%	
A-*Octodiff*$_{2048}$	6.11%	0.62
S-*Octodiff*$_{2048}$	15.32%	

Table 5: Average generated traffic percentage and processing time ratio when variable size data are considered.

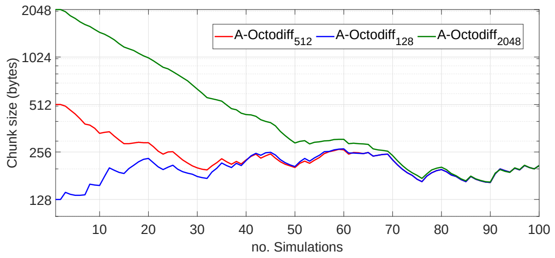

Figure 13: Chunk size adaptation when variable size data are synchronized.

Summarizing, the implementation of the adaptive chunking in the *Octodiff* framework has allowed the realization of a remote synchronization mechanism characterized by low sensitivity to the characteristics of the files under processing. This aspect turns out to be significant in the IoT context where heterogeneous data are typically handled.

5 Conclusions

This paper proposed an analysis of remote data update solutions to be used in IoT scenario. Specifically, one of the most known file synchronization tools, namely *Octodiff*, has been combined with a recently proposed algorithm for adaptive chunking. The performance of the resulting mechanism has been evaluated, highlighting the provided traffic saving with respect to the classic *Octodiff* version working with a static approach. The proposed solution aims at catching the benefits of an edge computing for IoT that helps to reduce the network load on the one side and the IoT power consumption on the other side.

Acknowledgments

This work was developed in partnership with Aenduo s.r.l. and partially funded by the POR FESR italian project “Life 2020 - Biomedical IoT” CUP F87H18000390007.

References

[AGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015.

[Gitff] Github. *Octopus Deploy - Octodiff*, https://github.com/OctopusDeploy/Octodiff.

[MTPC19] E. G. Maria Verzegnassi, K. Tountas, D. A. Pados, and F. Cuomo. Data conformity evaluation: A novel approach for iot security. In *IEEE 5th World Forum on Internet of Things (WF-IoT)*, pages 842–846, April 2019.

[PCS+18] A. Petroni, F. Cuomo, L. Schepis, M. Biagi, M. Listanti, and G. Scarano. Adaptive data synchronization algorithm for iot-oriented low-power wide-area networks. In *Sensors*, volume 18(11), Nov 2018.

[RGXZ17] J. Ren, H. Guo, C. Xu, and Y. Zhang. Serving at the edge: A scalable iot architecture based on transparent computing. *IEEE Network*, 31(5):96–105, 2017.

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[SNT04] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proceedings. 20th International Conference on Data Engineering*, pages 153–164, April 2004.

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. 1996. Technical Report TR-CS-96-05.

[WZL+19] T. Wang, J. Zhou, A. Liu, M. Z. A. Bhuiyan, G. Wang, and W. Jia. Fog-based computing and storage offloading for data synchronization in iot. *IEEE Internet of Things Journal*, 6(3):4272–4282, June 2019.

[YIS08] H. Yan, U. Irmak, and T. Suel. Algorithms for low-latency remote file synchronization. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, pages 156–160, April 2008.

[ZL16] Zhijie Lin and Lei Zhang. Data synchronization algorithm for iot gateway and platform. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pages 114–119, Oct 2016.