Analysis and Evaluation of HTTP/2 Flow Control Algorithm for IoT

1st Diego Londoño[1 , 2]
[1]*Department of Electrical Engineering*
Universidad de Chile
[2]*Everis an NTT Company*
Bogotá, Colombia
dlondond@everis.com

2nd Sandra Céspedes[1 , 2]
[1]*Department of Electrical Engineering*
[2]*NIC Chile Research Labs*
Universidad de Chile
Santiago, Chile
scespedes@ing.uchile.cl

3rd Javier Bustos
NIC Chile Research Labs
Universidad de Chile
Santiago, Chile
jbustos@niclabs.cl

***Abstract*—Traditional Internet is focused in communicating people. In contrast, in the Internet of Things (IoT), devices or things communicate among them. However, these devices are usually constrained in terms of processing, storage, batteries, and transmission power. Such limitations have led to reconsider the use of widely used protocols on Internet and to create new protocols designed to constrained scenarios. Although, HTTP, one of the well-known and most used on the Internet application layer protocol, was not designed for constrained scenarios; currently, it is highly used on the IoT. Considering this, it is necessary to adapt the protocol to the restricted scenarios to achieve its properly work with an acceptable performance. In this work, we took HTTP/2, the most recent version of the protocol, as a basis to propose a flow control algorithm oriented to IoT. Through mathematical modeling and simulations, we compare our proposal with another generic and not IoT-oriented flow control algorithm for HTTP/2, NGHTTP/2. The results showed that the proposed algorithm decreased overhead between 25% and 69% in all scenarios. In terms of throughput the results were mixed, in two of three scenarios the reference algorithm was between 3% and 13% faster. In the remaining scenario the throughput was very similar.**

***Index Terms*—HTTP/2, IoT, Application protocols, Flow control, constrained devices.**

I. Introduction

The *Internet of Things* (IoT) is a paradigm whose basic idea raises the connection to the Internet of a new variety of ”things” or ”objects”, such as sensors, motors, RFID *tags*, household appliances, machines, vehicles, etc. The objects can send and receive information, interact, and cooperate with each other. This new field opens up many possibilities for applications in sectors such as: housing and buildings, industry, mining, the agricultural and forestry sector, urban infrastructure, and transport.

IoT implies that the number of Internet connections generated by things is much greater than the connections originated by humans; thus, communications and traffic *Machine-to-Machine* take greater prominence, but at the same time, challenges are generated in terms of *big data*, communication protocols, and security, among others.

IoT devices are typically constrained, which means that, they have low memory, low processing capacity, low transmission power, and are powered by batteries, that are expected do not to need to be changed in months or even years. The aforementioned restrictions have led to rethink the use of the most widespread protocols on the Internet, such as HTTP/1.1 and TCP, as they are not designed in principle for these constrained devices, and can be expensive in terms of computing requirements. This is why different entities and consortia began efforts to create standards for IoT more than a decade ago; some examples are: IEEE 802.15.4 in the physical and access layer, 6LoWPAN as an adapting layer between the link layer and the network layer, and the *Constrained Application Protocol* (CoAP) and the *Message Queue Telemetry Transport* (MQTT) protocol at the application layer, among other technologies and protocols.

Some functionalities of the protocols recently created for IoT already existed in older protocols, with the advantage of the latter being widely known and tested. Due to the above, some authors have pointed out the feasibility of adapting HTTP/2 to IoT [1]. Some of the reasons for doing such an adaptation are interoperability, security, knowledge, and wide use of the protocol. The present work addresses this discussion and seeks to generate a proposal for adapting HTTP/2 to IoT, starting from a tour of the theoretical framework of what concerns IoT, the different protocol stacks that are handled in this field, and related works.

This work focuses on designing an HTTP/2 flow control algorithm for IoT. Therefore, it is based on the architecture principles for this purpose established in [2], where they indicate to take into account the operability, interoperability, security, performance, etc. when making an extension to a protocol. To test the proposed algorithm, we developed a simulation tool that considers different IoT scenarios; also, the proposed algorithm is compared against a generic HTTP/2 flow control algorithm not designed for IoT. Some key concepts are discussed below, followed by the proposed algorithm, mathematical modeling, simulation scenario, results, and future work.

In the following sections, we discuss some of the key concepts for IoT and constrained devices, then we reviewed several related works, followed by a presentation of the proposed algorithm, the mathematical modeling, the simulation

978-1-7281-8715-0/20/$31.00 © 2020 IEEE

scenario, discussions of results, and conclusions.

A. Constrained Devices

According to RFC 7228 [3], constrained devices are those that have limitations in terms of computing capacity, battery, communication ability, and available memory. They are often used as sensors or actuators, smart objects, or smart devices. Constrained devices are classified into:

- Class 0: Severely restricted devices in memory and computing capabilities. They cannot connect to the Internet safely and for this they need other equipment such as *gateways, proxies* or servers.
- Class 1: Devices that cannot easily work with traditional Internet protocols such as HTTP, TLS, and TCP. However, these devices can run protocols that have been designed for constrained devices, such as CoAP over UDP.
- Class 2: Devices with fewer restrictions that can run the traditional Internet protocol stack, however, it is desirable for these devices to be efficient in terms of power consumption and bandwidth.

Constrained devices have different energy use strategies according to their needs. The first strategy is when the device is always on, hence, the device is always connected to the network. The second strategy is when the device is usually off, which is the normal power off strategy. The device is asleep for long periods of time and reconnects only when needed. The last strategy is when the device is energy efficient, hence, the device sleeps for short periods between transmissions but somehow preserves the connection to the network. This low power consumption strategy is for devices that require to operate with a small amount of energy, but that have high connectivity requirements.

B. HTTP/2

This new version of HTTP is described in RFC7540 [4] and introduces adjustments and modifications in order to have a robust and agile protocol, which is better adapted to today's applications. One of the characteristics of the previous versions of HTTP, which can be seen as a disadvantage, is the size of the *headers* and that the information in many cases is repetitive, which generates unnecessary traffic and congestion causing the TCP window to fill up quickly.

In HTTP/2 these issues are mitigated, optimizing the semantics of *headers*, which enable the interleaving of requests and responses on the same connection. The prioritization of requests is also allowed, besides adding new forms of interaction, such as the *server push*, that allows the server to send advance information to the client, assuming that it will need it [4].

As a result of the above, it is gotten a lighter protocol without losing robustness, since it requires fewer TCP connections and smaller headers. Additionally, the user will see greater agility in the applications.

The most relevant characteristics of HTTP/2 are identified below [5]:

- Binary frames: In HTTP/1.x the frames are in plain text, this makes inspecting them relatively easy and their size increases. In HTTP/2, frames are binary, thus reducing their size.
- Compression of the *header*: In HTTP/1.x the header field contains plain text metadata, this can sometimes add more than a kilobyte to the frame. In HTTP/2 the metadata in the *header* is compressed using the Huffman algorithm to reduce the load and improve performance.
- Multiplexing: One of the disadvantages of HTTP/1.x is that if the client requires multiple parallel requests to the same server, multiple TCP connections must be created. This increases traffic and response times significantly, due to the *handshake* that each connection must have. In HTTP/2, frames of different *streams* can be sent over the same TCP connection.
- *Stream* prioritization: If a client has several *streams* with the same server, the client can assign a priority for a new *stream*; in this way, the client can express how he will prefer that the server manages the resources.
- Flow control: In addition to TCP flow control, HTTP/2 defines a window for flow control at the application layer. This is done in order to protect end devices that operate under restricted resources, ensuring that *streams* on the same connection do not destructively interfere with each other. The flow control is used for each stream individually or also for the connection as a whole.
- *Server push*: The server sends responses to a client in advance in association with a previous request initiated by the client. This is useful when the server knows that the client will need these responses because they are associated with the original request. With this functionality you save time and traffic.

II. Related Works

In this section, we review some of the works that have evaluated application protocols in IoT environments. In [6], the performance of CoAP is evaluated considering the number of hops between nodes; the authors simulated a scenario using Contiki, where humidity and temperature sensors are periodically required by a CoAP client. As a result, it is evident how light CoAP is and the advantage of using UDP, it also highlights the energy saving and improves the response times of constrained devices.

One of the reasons for trying to use HTTP/2 in IoT is the security risks suggested by the creation of new protocol stacks, as in the case of CoAP. Regarding this, in [7], the security aspects of the whole protocol stack used by CoAP in a constrained environment are analyzed: IEEE 802.15.4, 6LoWPAN, RPL, and CoAP. Security discussions in CoAP have mainly focused on the use of DTLS, the heavy cost of computing, and the *handshake* that causes fragmentation of messages. Compressing DTLS has been proposed as a solution. Furthermore, DTLS has the disadvantage of not supporting *multicast* messages in communication groups. The

document concludes that there are areas where DTLS falls short, becoming a security threat.

In [8], the authors design, implement, and evaluate an HTTP-CoAP *proxy*, to go from HTTP over IPv4 to CoAP over a 6LoWPAN network. The software of this *proxy* includes three modules. The first one is for an Apache web server. The second one for mapping HTTP to CoAP. And the last one is the CoAP module. The proxy has a caching mechanism that behaves as if it were directly the CoAP server, in order to improve system performance. The Apache JMeter tool was used to measure performance, which allows simulating multiple concurrent clients. As a result, it was found that latency increases as clients increase; the throughput indicated that the proxy is capable of supporting 25 GET requests per second and the effectiveness of the cache mechanism was verified, since it managed to reduce latency.

Other works have focused on simulated and experimental comparisons of application protocols. In [9], the authors seek to evaluate the performance of different application protocols for restricted scenarios: MQTT, CoAP, *Data Distribution System*, DDS, and a proprietary protocol based on UDP. The tests were done with a series of medical sensors that send information to a server. As relevant results regarding the performance of the protocols, the protocols based on TCP do not experience packet losses in difficult network conditions, with lost packet rates of up to 25% and latency of 400 ms. However, DDS significantly outperforms MQTT when it comes to telemetry latency in poor network conditions; even if DDS has a higher bandwidth consumption than MQTT, this is a better candidate for networks that require less loss.

More recently, in [10], different HTTP/2 parameters are evaluated using IoT-Lab [11] and Rapsberry Pi with NGHTTP/2. In addition to the window size, parameters such as: the maximum frame size, table size, and HEADERS list, server push and number of concurrent streams are evaluated, measuring the occupation of CPU, memory, and used power. Despite the large number of tests, the conclusions do not suggest precise changes in the parameters or in the operation of the protocol.

III. Proposed flow control algorithm

IoT scenarios are restricted since devices that interact on the network have limited capabilities. For this reason, protocols designed for IoT tend to be simple and light, unlike traditional protocols, such as HTTP or TCP, which are usually robust and heavy for a constrained device. Based on the above, a flow control algorithm for HTTP/2 is proposed to simplify control traffic and reduce the load on the devices when using the HTTP/2 protocol at the application layer.

The operation of the proposed algorithm is illustrated in Fig. 1. Packet exchange begins when the receiver sends a packet of the type SETTING_INITIAL_WINDOW_SIZE. With this packet, the receiver communicates to the sender the size of the *buffer* in bytes, which has been reserved for receiving packets. The free space of the *buffer* is known as a window and its size changes during transmission due to the input and output

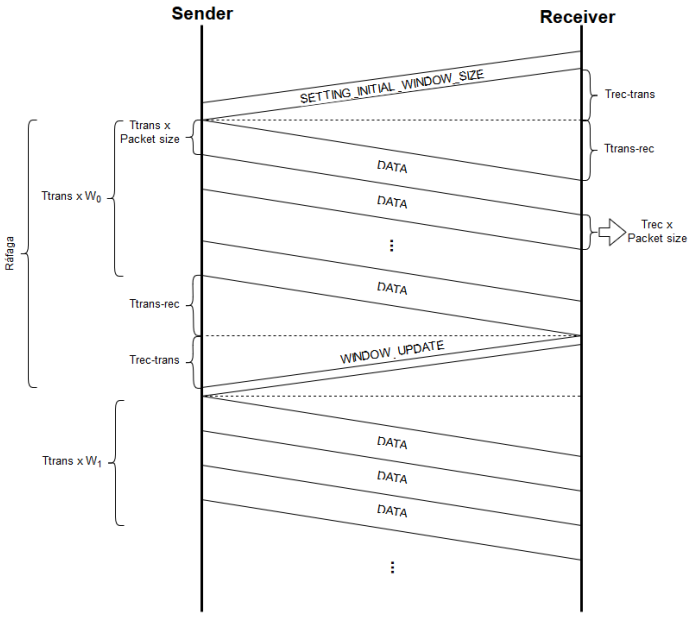

Fig. 1. Packet flow in the proposed algorithm

$T_{rec-trans}$	Propagation time in the network from receiver to sender. [s]
T_{trans}	Processing time for the transmission of a Byte. [s/Byte]
$T_{trans-rec}$	Network propagation time from sender to receiver. [s]
T_{rec}	Reading time of a Byte in the receiver. [s/Byte]
W_i	Window size. [Bytes]

TABLE I
DEFINITION OF TIMES INVOLVED IN THE TRANSMISSION.

of packets to the *buffer*. Then, the initial window corresponds to the size of the *buffer* and it is the maximum value that it can take.

After the SETTING_INITIAL_WINDOW_SIZE packet is received, the sender begins transmitting the DATA packets. The sender transmits as many bytes as the window allows, and once that number is transmitted, the transmission stops. At the same time, when the receiver gets the number of bytes announced in the window, it sends a WINDOW_UPDATE packet, where it tells the sender the window size is available at the moment, that means, how much free space it has in the *buffer*. After the WINDOW_UPDATE packet has been received by the sender, transmission resumes and the process repeats.

The variables shown in Fig. 1 are defined in Table I.

A. Theoretical performance analysis

The flow control algorithm can be mathematically modeled through a metric such as *throughput*. Based on [12], the maximum *throughput* in a burst can be modeled as shown in the equation 1 as follows:

$$\lambda_{sup} = min\left(\frac{8}{T_{trans}}, \frac{8}{T_{rec}}, \frac{8W}{W \times T_{trans} + T_{trans-rec} + T_{rec-trans}}\right). \quad (1)$$

When the transmission time is very long in relation to the other times involved in a complete transmission, the sender

behaves like a bottleneck. The equation 2 models this situation as follows:

$$\lambda_{sup}\left[\frac{bits}{s}\right] = \frac{8\left[\frac{bits}{Byte}\right]}{T_{trans}\left[\frac{s}{Byte}\right]}. \quad (2)$$

In the case where the receiver acts as a bottleneck, the modeling would be as shown in the following equation 3:

$$\lambda_{sup}\left[\frac{bits}{s}\right] = \frac{8\left[\frac{bits}{Byte}\right]}{T_{rec}\left[\frac{s}{Byte}\right]}. \quad (3)$$

The previous cases may occur when there are blockages or collapses in the devices or simply when one of these is very slow in relation to the other. In the case where the receiver and sender speeds are similar, the *throughput* will be determined by all the times that interact in the transmission, as shown in the equation 4 as follows:

$$\lambda_{sup}\left[\frac{bits}{s}\right] = \frac{8\left[\frac{bits}{Byte}\right] W\,[Bytes]}{W\,[Bytes] \times T_{trans}\left[\frac{s}{Byte}\right] + T_{trans-rec}\,[s] + T_{rec-trans}\,[s]}. \quad (4)$$

The scenarios grouped in the equation 1 apply for each W, that is, for each burst. A transmission usually consists of multiple bursts; therefore, the average *throughput* of the whole transmission must consider its entire duration, which is determined in the equation 5 as follows:

$$T_{total} = \sum_{i=0}^{n} T_{parcial_i} = \sum_{i=0}^{n} W_i \times T_{trans_i} + T_{trans-rec_i} + T_{rec-trans_i}. \quad (5)$$

Finally, the total *throughput* is established in the equation 6, where n is the number of bursts, which in turn is determined by the number of WINDOW_UPDATE packets sent by the receiver. Note that $i = 0$ refers to the first burst, which is determined by the initial window announced in the SETTINGS_INITIAL_WINDOW_SIZE packet.

$$\lambda_{total}\left[\frac{bits}{s}\right] = \sum_{i=0}^{n} \frac{8\left[\frac{bits}{Byte}\right] W_i\,[Bytes]}{T_{total}\,[s]}. \quad (6)$$

B. Reference Algorithm

In order to assess the contribution of the proposed flow control algorithm, we employ a reference algorithm already deployed with the NGHTTP/2 implementation [13] as a baseline. This is an implementation that is not designed specifically for IoT, so following the protocol's objectives, it seeks agility for current web applications. In this sense, the flow control algorithm that it uses is oriented to a fast and fluent data transfer in web scenarios.

The fundamental difference between the proposed algorithm and the reference algorithm, is that the latter does not wait to receive all the data in the window, instead, anytime the *buffer* is half full or more, it sends a WINDOW_UPDATE packet. In other words, the receiver announces, its window size each time it receives a data packet and its buffer is half-full or more than half-full. The operation of this algorithm is shown in Fig. 2.

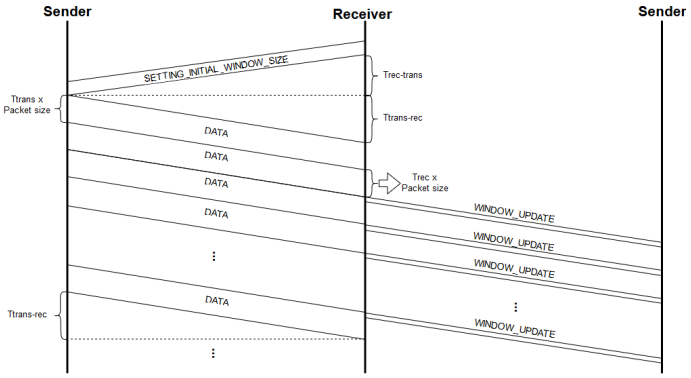

Fig. 2. Packet flow in the reference algorithm.

Parameter	Value
Propagation times in both directions	100 ms y 200 ms
Initial windows	2048, 4096, 8192, 16384, 32768, 65536
Pack sizes	512 B
Bytes to transmit	500 kB
Transmission speeds	10 kbps, 100 kbps, 500 kbps
Read speeds at receiver	10 kbps, 100 kbps, 500 kbps

TABLE II
SUMMARY OF PARAMETERS EVALUATED IN THE SIMULATIONS.

Graphically, when comparing the behaviors in Fig. 1 and Fig. 2, it is identified that the reference algorithm achieves a more fluent and fast communication by not stopping, but also, it uses more WINDOW_UPDATE packets.

IV. Evaluation and Results

In order to simulate the data transfer between sender and receiver, a customized simulator was built in Java. Its operation is based on a queue that simulates the window and a two-thread execution that simulates the sender and receiver. This simulator allows modifying variables such as: packet size, reading speed at the receiver, sending speed at the sender, propagation times, initial window size, and amount of data to transmit. Table II summarizes the parameters evaluated in the simulations of both the proposed algorithm (PA) and the NGHTTP/2 algorithm.

The metrics defined for the evaluation are the *throughput* and the amount of control traffic or *overhead* required to complete the transfer. The overhead is calculated using the number of WINDOW_UPDATE packets that the receiver sends to the sender. The code for the simulations is available in [14].

The results can be divided into three scenarios according to the transmission and reading speeds of the buffer: i) Receiver slower than the sender; ii) sender and receiver with similar speeds; and iii) receiver faster than the sender.

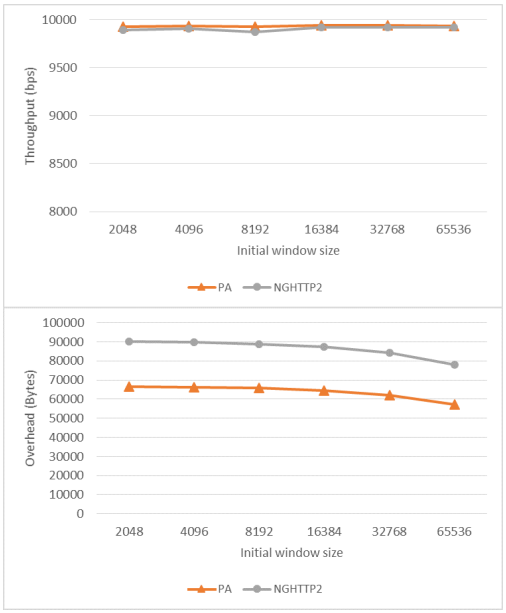

Fig. 3. *Throughput* and *overhead* when the sending speed is 100 kbps, the reading speed is 10 kbps, and the *delay* is 100 ms.

A. Scenario 1: Receiver slower than sender

Figure 3 shows the scenario when the sending speed is 100 kbps, the reading speed 10 kbps, and the delay 100 ms. In this scenario, the results show that in terms of *throughput* there are no significant differences between the two flow control algorithms, the performance of the proposed algorithm is the same or even slightly higher in most cases. Instead, the *overhead* is reduced by 26% on average by the proposed algorithm. This means that the use of the proposed algorithm does not sacrifice performance and reduces control traffic in the scenario where the receiver is very slow in relation to the sender. The reason for this result is that the NGHTTP/2 protocol suffers from packet losses at the start of a transmission, due to both the relative slowness of the receiver to read the *buffer* and the propagation time. The sequence where packet loss occurs is:

1) The sender receives a SETTING_INITIAL_WINDOW_SIZE packet, where the receiver announces the size of its initial flow control window.
2) The sender starts sending as many data packets as the advertised window allows.
3) Meanwhile the receiver gets and stores the received packets in the *buffer*.
4) When the receiver *buffer* is half full, it sends a WINDOW_UPDATE packet updating its available size.
5) Due to the propagation time between receiver and sender, the sender does not immediately receive that WINDOW_UPDATE packet. While the WINDOW_UPDATE packet travels to the sender, it has already sent as many bytes as the initial window allowed.
6) The sender receives the WINDOW_UPDATE packet that authorizes it to send a number of bytes that is around half of the initial window size, but at that moment the receiver *buffer* does not have the space available, since it has already received all (or almost all) the bytes of the first burst.
7) The sender sends the second burst with the amount of data pointed out by the last WINDOW_UPDATE packet. Since the receiving *buffer* does not have yet enough space to receive, some of the data is discarded.
8) In the NGHTTP/2 algorithm implemented in the simulation, the receiver checks the status of the *buffer* after receiving each packet, so it sends several WINDOW_UPDATE packets that causes packet drops.

The problem previously described is temporary. After that, the system stabilizes and the advertised windows begin to have values that do not lead to packet discarding, as they are controlled by the propagation and *buffer* reading times.

B. Scenario 2: Sender and receiver with similar speeds

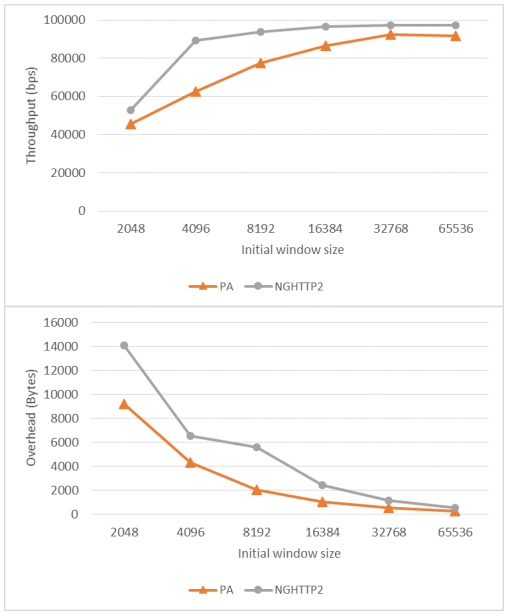

Fig. 4. *Throughput* and *overhead* when the sending and reading speeds are 100 kbps and the *delay* is 100 ms.

Figure 4 shows that the control traffic decreases significantly when the proposed algorithm is employed. In the case of same speed at sender and receiver, i.e., 100 kbps, the decrease is 49% in overhead for the proposed algorithm. Traffic volumes, using the NGHTTP/2 algorithm, tend to be higher when using low speeds, because the advertised windows are smaller. Instead, the traffic volumes with the proposed algorithm are similar for all cases, except when the initial window is 2048 Bytes.

As for the *throughput*, the NGHTTP/2 algorithm performs better with most of the simulated window sizes. For the case of sender and receiver with similar speeds, the reference algorithm is 13% higher. However, it should be noted that when the windows are enlarged, the differences in throughput are reduced. This is due to the low number of WINDOW_UPDATE

packets used by the proposed algorithm, since the advertised window sizes are close to their maximum value, allowing longer data traffic bursts.

C. Scenario 3: The receiver is faster than the sender

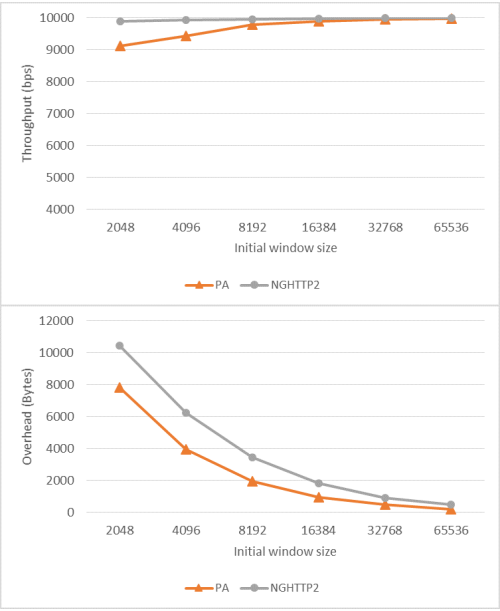

Fig. 5. *Throughput* and *overhead* when the sending speed is 10 kbps, the reading speed is 100 kbps, and the *delay* is 100 ms.

In the scenario shown in Fig. 5, the sender acts as a bottleneck. The high reading speed of the receiver compared to the speed with which data arrives, allows the *buffer* to remain almost idle, so the announced windows are values close to the maximum. In this scenario, NGHTTP/2's throughput is 3% higher on average in the scenario 3, although the differences are reduced for large window sizes.

In terms of overhead, the proposed algorithm decreases the control traffic by an average of 46% in scenario 3.

V. Conclusions

In this work, we have presented a proposed flow control algorithm for HTTP/2 when employed in Internet of Things scenarios. The algorithm seeks to reduce the overhead as a way to minimize the need for extra transmissions in constrained devices. We compared the proposed algorithm with the implementation of flow control of the NGHTTP2.

The results from simulations show that the proposed algorithm decreased the overhead between 25% and 69% compared with NGHTTP/2, which means a significant improvement for protocol simplification. In terms of *throughput*, the results were mixed, because in most cases the NGHTTP/2 algorithm was slightly faster, despite the fact that, in the scenario in which the receiver is very slow in relation to the sender, the proposed algorithm performed better. This behavior is relevant to IoT, in which restricted receivers may be common.

When the sending and reading speeds are similar, the network conditions become more relevant, especially for the proposed algorithm that is more sensitive to an increase in *delay*. Increasing the propagation times usually results in using less WINDOW_UPDATE packets, but decreasing the *throughput*.

The proposed algorithm showed to be a good option for the use of HTTP/2 in IoT scenarios, since it reduces the control traffic exchanged, without sacrificing too much *throughput*. The throughput reduction may not be a significant factor in many types of *Machine to Machine* communications, typical in IoT.

In the future work, we expect to test the proposed algorithm in a real implementation of HTTP/2 for constrained devices [15].

Acknowledgment

This work has been partially supported by ANID FONDECYT Regular No. 1201893, and the ANID-Basal Project FB0008.

References

[1] G. Montenegro, S. Cespedes, S. Loreto, and R. Simpson, "HTTP/2 Configuration Profile for the Internet of Things," Internet Engineering Task Force, Internet-Draft draft-montenegro-httpbis-h2ot-profile-00, Mar. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-montenegro-httpbis-h2ot-profile-00

[2] B. Carpenter, B. Aboba, and S. Cheshire, "Design considerations for protocol extensions," Internet Requests for Comments, RFC Editor, RFC 6709, September 2012.

[3] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," Internet Requests for Comments, RFC Editor, RFC 7228, May 2014, http://www.rfc-editor.org/rfc/rfc7228.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7228.txt

[4] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (http/2)," Internet Requests for Comments, RFC Editor, RFC 7540, May 2015, http://www.rfc-editor.org/rfc/rfc7540.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7540.txt

[5] "] HTTP/2, High Performance Browser Network," https://hpbn.co/http2/, [Accessed: 06-sept-2017].

[6] S. Thombre, R. U. Islam, K. Andersson, and M. S. Hossain, "Performance analysis of an ip based protocol stack for wsns," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016, pp. 360–365.

[7] R. A. Rahman and B. Shah, "Security analysis of iot protocols: A focus in coap," in *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, March 2016, pp. 1–7.

[8] A. B. Sulaeman, F. A. Ekadiyanto, and R. F. Sari, "Performance evaluation of http-coap proxy for wireless sensor and actuator networks," in *2016 IEEE Asia Pacific Conference on Wireless and Mobile (APWiMob)*, Sept 2016, pp. 68–73.

[9] Y. Chen and T. Kunz, "Performance evaluation of iot protocols under a constrained wireless access network," in *2016 International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT)*, April 2016, pp. 1–7.

[10] D. Ruz, "Evaluación del protocolo http/2 para internet de las cosas," *Memoria de Título, Universidad de Chile*, 2019.

[11] "Future internet of things (fit) iot-lab," https://www.iot-lab.info/, [Accessed: 17-Ago-2019].

[12] E. Arthurs, G. Chesson, and B. Stuck, "Theoretical performance analysis of sliding window flow control," *IEEE Journal on Selected Areas in Communications*, vol. 1, no. 5, pp. 947–959, November 1983.

[13] Nghttp2, "Nghttp2: HTTP/2 C Library and tools," https://github.com/nghttp2/nghttp2, [Accessed: 04-sept-2017].

[14] diegold91, "Simuladores de control de flujo - github," https://bit.ly/2kuuheB, Sep 2019, [Accessed: 10-sept-2019].

[15] F. Lalanne, "two: Http/2 for constrained iot devices," https://github.com/niclabs/two, Mar 2020, [Accessed: 15-jul-2020].