EasyChair Preprint

№ 14596

Simple ARQ Protocol for Reliable Transport in LowPANs

Manel Chahed, Aref Meddeb and Amine Boufaied

August 29, 2024

Simple ARQ Protocol for Reliable Transport in LowPANs

Manel Chahed
Higher Institute of Computer Science and Communication Technologies of Sousse
University of Sousse
Sousse, Tunisia
chahedmanel07@gmail.com

Aref Meddeb
Faculty of Engineering
University of Sherbrooke
Sherbrooke, Canada
Aref.Meddeb@Usherbrooke.ca

Amine Boufaied
MARS Research Laboratory, Higher Institute of Computer Science and Communication Technologies of Sousse
University of Sousse
Sousse, Tunisia
Amine.boufaied@isitc.u-sousse.tn

***Abstract*—With the proliferation of IoT devices and applications, an overwhelming set of protocols have been proposed to respond to the requirements of such devices and applications. So far, UDP has been considered as the de facto transport protocol for IoT. Several reasons are behind this choice, primarily its simplicity, low latency, limited overhead, and small energy requirements. On the other hand, TCP requires significant resources, primarily energy and is too complex to implement on tiny devices, with limited memory, CPU power and battery. Therefore, often, reliability is provided by upper-layer protocols, mainly by the applications themselves. In fact, classical IoT applications such as sensing, identification and actuating generate multiple copies of data due to the hardware redundancy and periodic updates. Examples of such applications include, agricultural, environmental, traffic, and healthcare monitoring. However, with the increased requirement for reliable transport and real-time IoT applications, UDP and TCP may not be the best candidates for such applications. For instance, in battlefields, firefighting, natural disasters, sensors may not be capable of generating multiple copies of critical data periodically and might even be destroyed after sending one or two messages. Thus, the network must provide reliability in such scenarios. Moreover, smart textiles are more and more integrating sensory devices and require reliable transmissions. In fact, with the very stringent resource constraints on one hand, and the requirements to respect the Specific Absorption Rate (SAR) of human bodies, on the other hand, re-transmissions and power must be kept at their lowest levels. In this paper, we propose a simple Automatic Request (SARQ) transport protocol that uses acknowledgments and and a retransmission mechanism. Through a realistic simulation setup using Contiki motes in the Cooja simlator, we show that our protocol exhibits slightly higher energy consumption and resource requirements than UDP, but far less than TCP. Conversely, we show that our protocol exhibits 99% Packet Delivery Ratio (PDR), while UDP and TCP exhibit 74% and 99% PDR, respectively.**

***Index Terms*—TCP, UDP, IoT, Reliable Transport, ARQ.**

I. Introduction

Because early IoT applications were dominated by sensory and identification devices, re-transmission of lost and/or delayed packets have not been an issue since sensors and tag readers often perform redundant transmissions of the same information. As such, a simple transport protocol such as the User Datagram Protocol (UDP) was sufficient. Over the last decade, IoT has evolved to encompass loss-sensitive and mission-critical applications such as security surveillance, military operations, disaster monitoring, Industrial IoT clouds and Web of things. Thus, reliable data transfer becomes a requirement of IoT transport layer protocols. Naturally, the Transmission Control Protocol (TCP) (RFC 7414) emerges as the most popular candidate to deliver reliable transport, but it has been widely criticized and deemed inadequate for constrained devices. Therefore, several attempts have been made to reduce TCP complexity and overhead. These include μIP [1], lwIP [1], RIOT (GNRC TCP) [2], BLIP/TinyOS TCP [3], FreeRTOS TCP [4], μC/OS TCP [5] and TCPlp [6]. Besides UDP and TCP, commonly used transport layer protocols include Datagram Congestion Control Protocol (DCCP) (RFC 4340), Stream Control Transmission Protocol (SCTP) (RFC 9260), and Quick UDP Interconnections (QUIC) (RFC 9000), but these were not designed for IoT. In this paper, we analyze the performance of TCP in IoT networks. We propose a Simple

ARQ protocol, termed SARQ, that provides a low cost, energy efficient and reliable transport in IoT. The paper is structured as follows: In Section I, we present TCP implementations for constrained devices. In Section II, we explore low-power TCP for IoT solutions. In Section III, we describe our ARQ protocol and compare its performance to UDP and TCP using Contiki motes in the Cooja Simulator. Finally, we conclude the paper by summarizing our findings and some future work directions.

II. TCP IMPLEMENTATIONS FOR CONSTRAINED DEVICES

TCP was designed in the early eighties, when most computers were running with 8- and 16-bit, 128-512 Kbytes of memory, and 2-4 Mhz CPU computers. Today, those computers would be considered as constrained devices. Nonetheless, the amount of data that must be processed and the performance requirements are not as they used to be. While one would accept to wait a couple of minutes to collect environmental forecasts back then, today we need such information to be gathered, received, and treated within a few seconds. Very recently, the IETF launched the Light-Weight Implementation Guidance (LWIG) working group (WG) to collect feedback from implementers of IP stacks in constrained devices. Other important WGs include the Constrained RESTful Environments (CoRE) and the TCP Maintenance and Minor Extensions (TCPM) WGs. In this Section, we provide the main proposals made so far for constrained devices running on 8- to 16- or 32-bit processors. We focus on those primarily based on TCP (RFC 9006). Table I gives a concise comparison between the various TCP implementations in IoT systems.

TABLE I
COMPARING THE TCP IOT IMPLEMENTATIONS

Comparing various implementations of TCP in IoT						
Solution Name	Platform	OS	RTT/ RTO	Congestion Avoidance	Flow Control	RAM
μIP/ μIPv6	8-16 bit	None/ Contiki	Yes (Karn's algorithm)	No	Stop & Wait	1/1.8 KB
lwIP	8-32 bit	None/ Multiple	Yes	Yes	Slow Start	40 KB
GNRC TCP	8-16 bit	RIOT	Yes	No	Single MSS Window	1.5 KB
BLIP	8-16 bit	TinyOS	Constant RTO	Yes	Single MSS Window	1 KB
FreeRTOS TCP	8-16-32 bit	FreeRTOS	Yes/ Delayed acknowledgement	No	multiple-MSS	20 KB
μC/OS TCP	8-16-32 bit	μC/OS	Yes	No	multiple MSS	22 KB
TCPlp	32 bit	FreeBSD	Yes	Yes	Slow Start	32-64 KB

A. μIP

μIP (github.com/adamdunkels/uip) is a twenty years old lightweight TCP/IP stack designed for constrained embedded systems using 8 and 16-bit micro-controllers that is written by Adam Dunkels. Due to its simplicity, μIP has been widely implemented by manufacturers including Cisco, Atmel and Texas Instruments. However, μIP implements only limited TCP functions. Further, the stop-and-wait flow control has very poor performance especially with poor quality links where a single segment may require several re-transmissions.

B. lwIP

Lightweight IP (lwIP) (www.nongnu.org/lwip) is another TCP/IP stack, also developed by Adam Dunkels, designed for constrained embedded systems using 8- and 32-bit microcontrollers. lwIP have gained significant acceptance and is being implemented by well-known manufacturers as well such as Texas Instruments, Atmel, Analog Devices, ST Microelectronics, and others. However, as far as we know, implementations made so far run UDP-Lite or basic TCP functions and have not been tested with congestion scenarios.

C. RIOT : GNRC TCP

The RIOT (api.riot-os.org) TCP implementation, also known as Generic Network Stack (GNRC) TCP or GNRC TCP, was designed for Class 1 devices (RFC 7228). It is primarily aimed for 8- and 16-bit micro-controllers. As most IoT hardware is based on the streamlined IEEE std. 802.15.4 MAC and PHY layers, this std. defines Full Function Devices (FFD) and Reduced Function Devices (RFD). FFDs have resources allowing them to act as network coordinators and perform routing and various network functions. However, RFDs have very limited resources and may not be able to support GNRC TCP.

D. BLIP/TinyOS TCP

TinyOS (www.tinyos.net) is an old OS designed for tiny devices. Native TinyOS provides a subset of the socket interface primitives via a basic library, allowing the implementation of an experimental TCP stack. BLIP TCP, developed to enhance TCP performance on TinyOS, differs from standard implementations in three key ways. Firstly, it incorporates basic congestion control mechanisms such as slow-start, congestion avoidance, and fast re-transmission. However, it only utilizes them for fast re-transmission, as it transmits all buffered data once the transmission timer expires. Secondly, due to memory constraints, BLIP TCP sets the receive window to one Maximum Segment Size (MSS) window size. Thirdly, it employs a constant TCP Retransmission Timeout (RTO) value (3 seconds by default), resulting in a consistent re-transmission rate.

E. FreeRTOS TCP

FreeRTOS (www.freertos.org/) is a real-time operating system for embedded devices running on 16- and 32-bit microprocessors. However, FreeRTOS faces a tradeoff between memory allocation and performance. In its basic configuration, memory isn't allocated to TCP connections, and instead, the stack dynamically allocates and deallocates memory from the FreeRTOS heap as needed. While this approach can slow down data transmission, it ensures flexibility in memory usage. Conversely, a faster implementation allocates buffers to TCP connections, resulting in quicker and more predictable memory allocation. Yet, this approach may consume a significant amount of memory.

F. µC/OS TCP

µC/OS (www.micrium.com/rtos/kernels/) is a Real-Time Operating System (RTOS) for embedded devices. It is intended for 8-, 16- and 32-bit microprocessors and supports a preemptive multitasking real-time kernel, with an optional round robin scheduling. While FreeRTOS is licence free, µC/OS requires the purchase of support documentation and a licence for commercial applications. Further, as opposed to FreeRTOS, µC/OS uses a bitmap scheduler that has the advantage of allowing a more deterministic and faster processing. This however reduces the application design flexibility as it only allows a one-to-one matching between tasks and priority levels. On the other hand, FreeRTOS is relatively flexible as it allows any number of tasks to share the same priority.

G. TCPlp

Based on Berkeley's FreeBSD OS, TCPlp is designed to operate within the resource constraints of IoT hardware. While some IoT hardware may be capable of supporting full TCP functionality, much of the hardware available on the market may have as little as 1 KB of RAM. As the trends moves towards smaller and more discrete sensors, wearable devices, and body networks, resources are expected to become even scarcer, with device sizes potentially not exceeding a couple of millimeters. Further, the memory is not expected to be fully allocated for the networking functions. Furthermore, the TCPlp solution does not address the header compression issues. Thus, it may not be implementable for RFDs.

III. Low Power TCP for IoT

Several papers have studied and critiqued various TCP implementations in IoT devices [7] [8] [9]. However, despite this extensive research, none of the above-mentioned TCP solutions for IoT provides comprehensive documentation on how all TCP features are actually implemented. Further, connection setup, teardown and header compression are often not provided. Developers and designers must go through sparse source code and documentation to figure out the specific implementation details. In this Section, we present a basic proposal to support TCP in IoT systems. This mainly includes connection setup and tear down, flow and congestion control, and header compression.

A. Header Compression

Draft [10] specifies TCP header compression. However, the draft expired in April 2011 and, as far as we know, no subsequent updates have been made so far. In this paper, we propose a new compression scheme inspired by that draft but modified to support segment numbering rather than bytes. We propose a jumping window flow control scheme, as specified in figure 1. We also explain how to map between native TCP, which might be typically running on an unconstrained device, and our proposed scheme, running on a constrained device.

Source Port	Destination Port	DATA	LASTF	URG	ACK	PSH	RST	SYN	FIN	SEQ	ACK	Checksum
4	4	1	1	6						32	32	16 (bits)

Fig. 1. TCP Header Compression

Below we provide detailed explanations how our protocol works.

B. Port Numbers

TCP and UDP headers start with the 16-bit source and destination port numbers, respectively, used to uniquely identify multiplexed applications. In a way similar to that used by 6LowPAN where UDP port numbers may be reduced to 4 bits, TCP port numbers may follow the same rules. However, a specific range must be defined for TCP, similar to the 61616-61631 range used for UDP compression (RFC 8138).

C. Sequence Numbers

The sequence (SEQ) and acknowledgment (ACK) number fields used by TCP are 32 bits long each. This means that 64 bits would be required. Moreover, the window size field (WNDW) is 16 bits long. This makes it a total of 80 bits i.e., 10 bytes, which is obviously too much for small frames. With limited RAM space for the entire OS, typically a couple of hundred kilobytes, there is no need for such huge space just for numbering segments. Thus, we need to adequately compress the SEQ, ACK, and WNDW fields. These are replaced by 4-bit long N(S) and N(R) fields and the WNDW field is elided, as it is no longer needed as explained in figure 2.

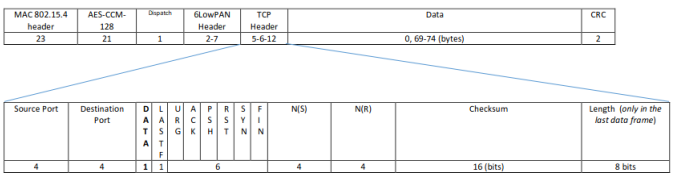

Fig. 2. TCP Header Field Compression: SEQ, ACK, and WNDW

D. Connection Setup

As pet IETF draft [11] a TCP connection in IoT is typically initiated by the constrained device, allowing it to enter sleep periods. The draft does not address the three-way handshake process or sequence numbers. TCP typically uses randomly generated Initial Sequence Numbers (ISNs), but this requires

additional processing for constrained devices. To address this, the draft aims to simplify the algorithm for generating ISNs, without affecting their length, ensuring randomness and unicity.

E. Flow, Congestion, and Error control

We propose a very basic solution to perform flow, congestion, and error control. The idea is to use a Selective Repeat (SRP) jprocedure with HDLC-like numbering for flow, congestion, and error control. Due to IoT device radio chip limitations, Jumping Windows (JW) is used, where the sender sends up to W frames and waits for cumulative acknowledgment before moving on to the next window. In our proposal, the JW flow control is implemented at the transport layer. It resumes the TCP control mechanisms inside the IoT network. As native TCP uses bytes as sequence numbers, we perform a mapping between these numbers and those used for the JW. Following the opening of the TCP connection, the ISN is memorized. The sequence numbers of the first fragment of the JW begin at 0. Each fragment of a TCP segment has a configurable Fragment Size (FS) of 64, 69, 72 or 74 bytes.

```
Dispatch
11111PWW
P=0:   SRP
P=1:   GBN
WW:    00  FS=64
       01  FS=69
       02  FS=72
       03  FS=74
```

We limit the maximum JW size to 8 fragments of 64—74 bytes each. This allows at least 512 bytes of data per TCP segment with using the Selective Repeat Scheme. Thus, the MSS must be correctly set, according to the implemented window Size. In the IoT network side, we support two main types of frames carrying specific TCP headers: *data frames*, with sequence numbers and *connection setup frames*. Data frames, marked with the DATA flag, carry information up to the maximum configured size, with only the last frame being potentially smaller and including the ”length” field. Once all data frames have been sent, the receiver acknowledges the total data received, using the ISN to deduce SEQ and ACK values for the end-to-end user connection. The total data received is calculated as ACK = SEQ + $N \times S + L$, where N is the number of full fragments, S is the maximum fragment size, and L is the last fragment's size.

IV. Simple ARQ Protocol Operation

To achieve a balance between the reliability characteristics of TCP and the speed benefits of UDP, our SARQ protocol combines the best of both worlds, offering singnificantly improved reliability compared to UDP, but faster data transfer and less power usage than TCP. Furthermore, we maintain the three-way handshake mechanism of TCP as it assures reliability of connection setup, since we are using an IP-based connectionless network layer. The SARQ algorithm is outlined. See the algorithm 1 for details.

Algorithm 1 Simple ARQ Algorithm

Initialize the process and UDP connection
Enter an infinite loop to handle incoming and outgoing packets:
if the device is the Sender **then**
  Create a packet with type 'D' and sequence number 1.
  Send the packet to the receiver.
  Handle re-transmissions for unacknowledged packets after a timeout.
  Wait for an ACK from the receiver.
  if the ACK is received **then**
    send another packet.
  end if
  if the ACK is not received after a timeout **then**
    Re-transmit the packet.
    if maximum attempts are reached and there is no ACK received for this packet from the receiver **then**
      The packet will be considered lost.
    end if
  end if
end if
if the device is the Receiver **then**
  Listen for incoming packets.
  Receive a packet, send an ACKs to the sender.
  Log information about the received packets.
end if
The process remains active to handle incoming and outgoing packets indefinitely.

V. Experimental Results

In this section, we compare the performances of UDP, TCP, and the SARQ protocols in the context of Internet of Things (IoT) applications. This comparison aims to evaluate UDP and TCP, and illustrate how SARQ is more suitable for sensitive and mission-critical IoT applications. The simulation platform used is Contiki-NG (github.com/contiki-ng/contiki-ng/tree/release/v4.2), an open-source operating system designed for resource-constrained IoT devices, with version 4.2 used for this work. It features a low-power IPv6 communication stack, enabling Internet connectivity, and supports various platforms such as the Texas Instruments MSP430. Contiki-NG accommodates several types of motes, including Cooja, Sky, and Z1 motes. However, the documentation for Contiki-NG is limited, but it can be supplemented with the OpenTestbed simulator, which provides a realistic environment for simulating and testing IoT devices. In this setup, We deployed four Z1 client motes and one Z1 server mote, with each client mote configured to transmit 100 packets to the server mote. Figure 3 illustrates the network topology utilized in our simulations. Packet size varies from 10 to 80 bytes due to the RAM size constraint of the Z1 mote in Contiki-NG. This limits the maximum packet size to 80 bytes. The receiver actively monitors the number of connected motes, rejecting connections if the limit (MAX_CONNECTED_MOTES) is

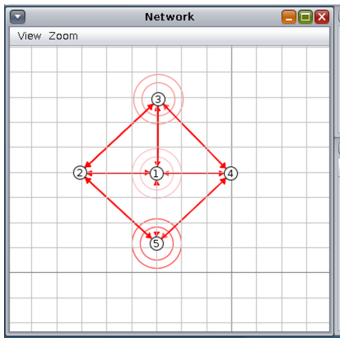

Fig. 3. Network topology

reached.

A. Performance Evaluations

We used three metrics to assess the performance of the three transport schemes: Packet Delivery Ratio (PDR), energy consumption and the end-to-end delay. We evaluate the impact of the packet size and the SARQ window size on PDR, energy consumption and the end-to-end delay. These metrics offer valuable insights on the suitability of the transport schemes for IoT applications. The impact of the Packet Size in IoT networks has been investigated in [12], which emphasized the role of packet size in determining the overall performance and reliability of data transmission within IoT networks. In this study, the packet size used ranged from 10 to 100 bytes. Further, energy consumption is critical in IoT networks and it has been deeply investigated [13]–[15]. Ref. [16] analyzes the critical effect of packet payload size, hops count, and interface speed on the end-to-end packet delay in IoT networks. The study highlights that larger packet sizes significantly increase the end-to-end packet delay. This is because larger packets require more time to transmit and are more likely to be delayed due to network congestion and packet loss.

B. Simulation and Results

In the implementation of TCP, the RTO is set to 3 seconds, and the Maximum Segment Size is calculated as the difference between UIP_BUFSIZE (the size of the uIP buffer) and the total combined length of the IPv6 and TCP headers, denoted by UIP_IPTCPH_LEN. However, the networking stack configuration in Contiki-NG establishes the IPv6 header length as 40 bytes and the TCP header compression length as 20 bytes. In Figure 4, we give the PDR vs the packet size. We observe that the PDR is nearly constant for all the packet sizes for the three protocols. While TCP reaches a 100% packet delivery, UDP delivers only 74.25% of the packets, while our SARQ scheme also achieves a 100% PDR. As opposed to what has been sated above, it seems that the packet size does not to have a major influence on PDR. At a first glance, this finding might somehow contradict the claims regarding the impact of packet size on PDR discussed qbove, but when we dig deeper into that statement, we limited ourselves here to packet sizes not exceeding 80 bytes because of the limitations of IoT devices. If we add the MAC and PHY layer headers, we are respecting the Maximum Transfer Unit (MTU) of IoT networks, i.e., the 127 bytes specified by the IEEE 802.15.5 [17] as a reference.

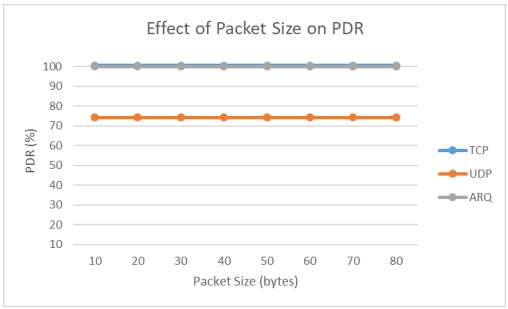

Fig. 4. Effect of Packet Size on PDR

Figure 5 gives the difference in energy consumption among TCP, UDP and SARQ protocols per transmitted packet. As we can notice, the consumed energy slightly increases with the packet size. Most importantly, SARQ consumes about 40 mJ more than UDP per transmitted packet, but about 40 mJ less energy than TCP.

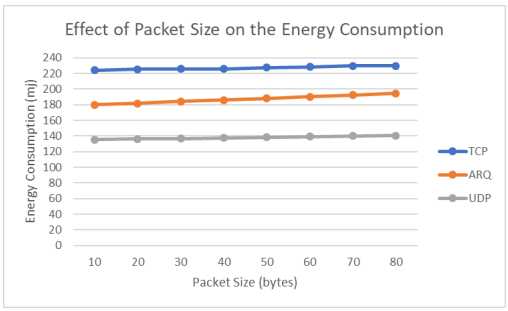

Fig. 5. Effect of Packet Size on Energy Consumption

On the other hand, figure 6 illustrates the effect of packet size on the average end-to-end delay for three different protocols: TCP, UDP, and SARQ. The results indicate that TCP exhibits the highest average end-to-end delay across all packet sizes compared to UDP and ARQ. Specifically, TCP's delay begins at approximately 0.03 ms for 10 bytes packets and gradually rises to around 0.035 ms for 80 bytes packets. In contrast, UDP consistently maintains the lowest average end-to-end delay among the three protocols, starting slightly below 0.023 ms for 10 bytes packets and only slightly increasing to just above 0.026 ms for 80 bytes packets. SARQ shows a delay that lies between TCP and UDP, beginning at about 0.025 ms for 10 bytes packets and rising to approximately 0.028 ms for 80 bytes packets. This comparison underscores the efficiency of UDP protocol in minimizing delay, while TCP's higher delay reflects its additional overhead for reliability, and SARQ's delay represents a middle ground with its error correction capabilities.

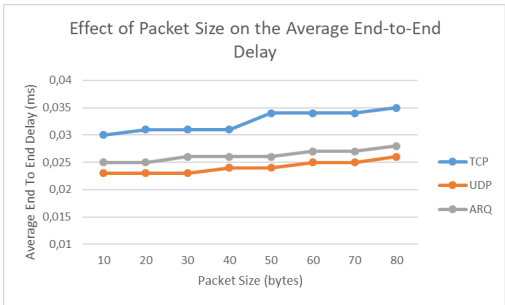

Fig. 6. Effect of Packet Size on the Average End-To-End Delay

C. Discussion

We can state summarize that SARQ is much more reliable than UDP due to its higher PDR (100%) compared to UDP (74.25%). Furthermore, when comparing it to TCP, SARQ exhibits better performance in terms of energy consumption, using about 40 mJ less than TCP. Additionally, SARQ achieves better average end-to-end delay compared to TCP, with a delay that is about 5 milliseconds shorter than TCP, while also maintaining a 100% PDR. Therefore, it can be concluded that SARQ provides higher reliability than UDP and outperforms TCP in terms of energy consumption, rendering it a suitable candidate for resource-constrained mission critical IoT applications.

VI. Conclusion

In this paper, we introduced a novel protocol for IoT networks termed Simple Automatic Request (SARQ), aimed at enhancing reliability and efficiency in data transmission within IoT networks. The SARQ protocol is designed to tackle the challenges of packet re-transmission, encompassing aspects such as Packet Delivery Ratio, energy consumption and and the end-to-end delay. Using simulations, we demonstrated the effectiveness of SARQ protocol in achieving reliable and efficient communication, offering a promising alternative to traditional protocols such as TCP and UDP. Our findings suggest that Simple ARQ protocol holds significant promise for enhancing the performance and reliability of IoT networks, making it a valuable contribution to the field. Given the diverse and increasingly specialized requirements expected by IoT applications, where reliability is paramount, the Simple ARQ protocol emerges as a more viable alternative to TCP. Unlike UDP, which is known for its speed in transmission time but lacks reliability, the Simple ARQ protocol offers a balance between speed and reliability, making it an attractive option for IoT devices. As a continuation of this work, we will envision the implementation of SAQR on real hardware platforms.

References

[1] Dunkels, A. ”Full TCP/IP for 8-Bit Architectures” MobiSys ’03, pp. 85-98 DOI 10.1145/1066116.106611, Internet draft draft-Dunkels, 2003.

[2] Baccelli, E. Gündoğa, C. Hahm, O. Kietzmann, P. Lenders, M. Petersen, H. Schleiser, K. Schmidt, and T. M. Wählisch ”RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT” Internet draft draft-RIOT, 2018.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, Internet draft draft-TinyOS, draft-TinyOS, 2005.

[4] F Guan, L Peng, L Perneel, and M Timmerman, ”Open source FreeRTOS as a case study in real-time operating system evolution”, Internet draft draft-FreeRTOS, 2016.

[5] Jean J. Labrosse and Freddy Torres, ”The Real-Time Kernel and the NXP LPC1700”, Internet draft draft-uC/OS , 2010.

[6] Behrouz A. Forouzan, and Sophia Chung Fegan, ”TCP/IP Protocol Suite”, Internet draft draft-TCPIp, 2002.

[7] Chansook Lim, ”Improving Congestion Control of TCP for Constrained IoT Networks” Internet draft draft-Journal-Sensors, 2020.

[8] Mohan Kumar and Ada Gavrilovska, ”TCP Ordo: The cost of ordered processing in TCP Servers”, Internet draft draft-of-The-35th-Annual-IEEE-International-Conference-on-Computer-Communications, 2016.

[9] Sam Kumar, Michael P Andersen, Hyung-Sin Kim and David E. Culler, ”TCPlp: System Design and Analysis of Full-Scale TCP in Low-Power Networks”, Internet draft draft-arXiv, 2018.

[10] A. Ayadi, D. Ros, and L. Toutain, “TCP header compression for 6LoWPAN”, Internet draft draft-aayadi-6lowpan-tcphc-nn, 2010.

[11] C. Gomez, J. Crowcroft and M. Scharf, “TCP Usage Guidance in the Internet of Things (IoT)”, Internet Draft draft-ietf-lwig-tcp- constrained-node-networks-nn, 2020.

[12] Smangaliso Mnguni, Pragasen Mudali, Adnan Abu-Mahfouz, and Mathew Adigun ”Impact of the Packet Delivery Ratio (PDR) and Network Throughput in Gateway Placement LoRaWAN Networks”, Internet draft draft-Southern-Africa-Telecommunication-Networks-and-Applications-Conference-(SATNAC), 2021.

[13] Zibuyisile Magubane, Paul Tarwireyi, and Mathew .O Adigun, ”Evaluating the Energy Efficiency of IoT Routing Protocols”, Internet draft draft-International-Conference-on-Multidisciplinary-Information-Technology-and-Engineering-Conference-(IMITEC), 2020.

[14] Valerio Freschi, and Emanuele Lattanzi, ”A Study on the Impact of Packet Length on Communication in Low Power Wireless Sensor Networks Under Interference”, Internet draft draft-IEEE-INTERNET-OF-THINGS-JOURNAL, 2019.

[15] Augustine Ikpehai, Bamidele Adebisi, and Kelvin Anoh, ”Effects of Traffic Characteristics on Energy Consumption of IoT End Devices in Smart City”, Internet draft draft-Global-Information-Infrastructure-and-Networking-Symposium-(GIIS), 2018.

[16] Imane Maslouhi, El Miloud Ar–reyouchi, Kamal Ghoumid, and Kaoutar Baibai, ”Analysis of End-to-End Packet Delay for Internet of Things in Wireless Communications”, Internet draft draft-International-Journal-of-Advanced-Computer-Science-and-Applications-(IJACSA), 2018.

[17] Marcin Piotr Pawlowski, Antonio J. Jara, and Maciej J. Ogorzalek, ”Maximizing the Extensible Authentication Protocol Maximum Transfer Unit to minimize the authenticating data transmission in the IEEE 802.15.4 networks”, draft-IEEE, 2015.