AQMoT: Implementation of Special Queue Management Approach for Internet of Things

Kerem AYTAÇ[1 2], Harun ÖZDEMİR[2], Malik TÜRKOĞLU[2], Muhammed Murat DİLMAÇ[2], Ömer KORÇAK[2]
[1]Koç Digital R&D Center, [2]Marmara University
Istanbul, Turkey
keremaytac@gmail.com, hrn.ozdemir52@gmail.com, maliktr34@gmail.com, muhammedmurat11@gmail.com,
omer.korcak@marmara.edu.tr

Abstract—**In the age of digitalization, the impact of Internet of Things (IoT) networks can be seen in many different areas, and there are significant issues with networking and capacity due to the low smartness and cost issues of IoT devices. Congestion and queue management, especially in an IoT network buffer, is one of the most important issues to consider. In this paper, we explained how to develop, implement and test the algorithm built to manage the queue and to avoid potential congestion by adding some intelligence to dumb nodes with a lightweight method called AQM-of-Things (AQMoT) which we have proposed recently. Extensive-form game model is used for defining decision making criteria of both IoT nodes (when to send) and the gateway node (when to drop). We implement a game model according to the queue level by including other network conditions as well. We propose a content-aware priority assignment method for the packets. We develop a realistic test environment and evaluate effectiveness of the AQMoT algorithm. Test results suggest that AQMoT effectively avoids packet drops for high priority data and reduces the burden on the gateway queue caused by redundant data sent by dumb IoT devices.**

I. Introduction

Over the past few years, Internet of Things (IoT) has become one of the most important technologies of 21st century. We can connect lots of objects such as Televisions, watches, fridges, coffee machines and various types of sensors to the Internet. In this way, these objects become smart devices. These devices are typically connected to a gateway device which receive and process data. Currently number of IoT devices are increasing exponentially and it is expected to hit to 500 billion of devices as of 2030 according to a study [1].

The increase in the number of IoT devices inevitably brings some challenges. When there exists more IoT devices in a system, more problems are encountered due to congestion and load in the gateway devices. In addition, some IoT devices send too much unnecessary data, causing the system to slow down and sometimes the system to stop working. IoT devices connected to a gateway device are mostly heterogeneous and they generate and send data with different priorities. For example, data generated by a fire alarm system have high priority and should be received quickly without waiting in the queue for a long time. When the network is too congested, such critical data may also be delayed or dropped and this is a significant problem which prevents the system working properly. Sometimes, sensors send redundant data, such as same value continuously. This does not benefit anyone and only creates data density in gateway. So, these data should have low priority and it could be beneficial to prevent sending of these redundant data at the source node. To handle all these issues, employment of a queue management approach is crucial for IoT systems that generates dense amount of data.

In the literature, there are several queue management algorithms. RED (Random Early Detection) [2] is a well-known algorithm, where the queue average size is always evaluated, and the packets are dropped with determined probabilities between given minimum and maximum queue thresholds. Adaptive RED (ARED) [3] is an extension of RED, and it is an adaptive approach which makes itself more or less aggressive according to the continuously evaluated average queue length. XRED [4] is a content aware extension of RED. GREEN (Generalized Random Early Evasion Network) [5] is a congestion avoidance approach which contains some proactive decisions to ensure more fairness between network flows. It also adopts more intelligent way of drop possibility calculation by evaluating some network conditions such as Maximum Size Segment, Round-Trip Time or link capacities, rather than RED's simplicity and randomness. PIE (Proportional Integral controller Enhanced) [6] is tough and suitable for many network scenarios where per-packet extra processing is not a must, which leads to being a very lightweight algorithm and easy to deploy in terms of both device and software. Different than the others, it evaluates drop probability by sensing the delay behavior finding out where to shift as direction (i.e. increasing or decreasing delay). CHOKe and its variations can be given as examples for this type of algorithms [7], [8]. These types of algorithms are dedicated to provide some fairness by penalizing unresponsive (or aggressive) traffic in case a network congestion occurs. These algorithms are stateless and have ease of implementation which make the algorithms highly preferable in some cases.

Recently we proposed AQMoT algorithm [9] based on game theory [10]. This algorithm determines how often IoT sensors will send data, and how to manage queue when the gateway is busy. In AQMoT, gateway and sensors exchange some control information, and some decision rules are defined for the sensors (to decide whether to send or when to send) and for the gateway node (to decide whether to drop the received packet). The study in [9] includes basics of the algorithm, some theoretical details and provides a conceptual comparison with the above-mentioned algorithms, without including any implementation and performance evaluation details. In this study, we aim to fill this gap and present implementation details of AQMoT in a real test environment using microcontrollers (Raspberry Pi's [11]) and standard IoT communication protocols (such as MQTT [12], socket programming). We provide extensive set of real-time results which depict effectiveness of the algorithm.

The rest of the paper is organized as follows. Next section describes the previously proposed AQMoT algorithm. Section III extends the idea behind AQMoT by giving further implementation details. Section IV describes the experimental setup including the environment details, hardware and network topologies. In Section V, test results are illustrated and discussed for several scenarios in the given setup. Finally, Section VI concludes the paper.

II. Overview of AQMoT algorithm

AQMoT is a queue management algorithm developed specifically for IoT environments with a novel decision-making mechanism for both the sender (whether to send a packet or not) and the receiver (whether to drop a packet or not) based on an extensive-form game [13] model as in the Fig. 1. The first player (sender) is typically a sensor node and the second player (receiver) is typically a gateway node. The sensor node has two actions (Send and Not Send) and the gateway node has two actions (Drop and Send) as well. Utilities of these nodes change according to action profiles and are defined by several variables defined in Table I. The details on how to determine these variables will be described in the next section, which was missing and left as a future work in the previous study [9].

The main aim is to avoid futile efforts of dumb IoT nodes, by throttling them if they are exploiting too much resources by sending a lot of low-priority data. This will prevent overhead in the queue system, and this make it efficiently work.

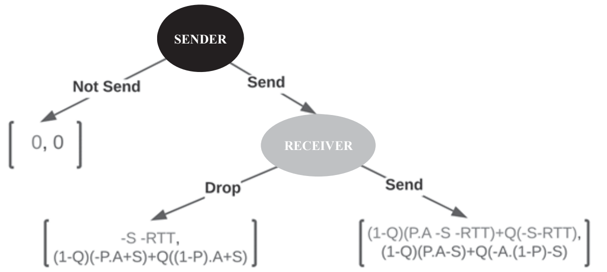

SENDER
Not Send
Send
0, 0
RECEIVER
Drop
Send
-S -RTT,
(1-Q)(-P.A+S)+Q((1-P).A+S)
(1-Q)(P.A -S -RTT)+Q(-S-RTT),
(1-Q)(P.A-S)+Q(-A.(1-P)-S)

Fig. 1 Extensive-form game formulation, where receiver the **destination** is Player 2 and sender the **source** is Player 1

TABLE I. Variables of the proposed game model

P	*Priority of packet from 0 to 1.*
S	*Size of packet from 0 to 1. It is calculated as the ratio of the packet size to the Maximum Segment Size (MSS).*
RTT	*Moving average of RTT occurred till then. 0 means low RTT, 1 means very high RTT.*
A	*An award. (-A stands for a penalty.) It should be greater than sum of maximum possible value of S and RTT.*
Q	*The percentage of busyness of queue at destination according to defined function.*

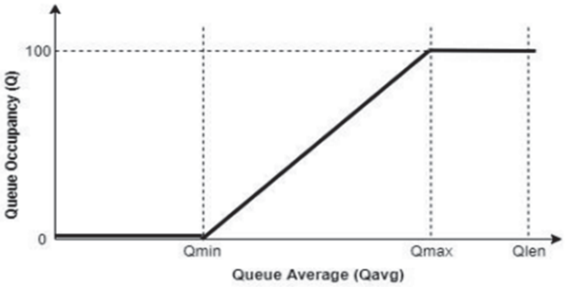


Fig. 2 Queue Occupancy Function by Queue Average

In the defined game, players are assumed to be rational and they try to maximize their payoffs. The sender communicates with the receiver to get the latest status. Q is the percent occupancy of the queue at the target illustrated in Fig. 2. This is similar to the drop function for other AQM (Active Queue Management) algorithms like RED [2]. $Qmin$ is the minimum threshold for the target to identify itself as null. The target doesn't tend to drop any packets up to this threshold but accepts them all. $Qmax$ is another threshold close to the target's maximum possible queue length. Beyond this threshold it tends to drop them all to avoid causing some network or buffering issues. Since the effect of these thresholds is not significant in the algorithm, we have determined them as linear and constant. The current occupancy percent (Q) is shared by the receiver at certain time intervals after the connection to all senders. If the sender sees no potential overhead on its part and finds possible action by the receiver, which is "accepted" for a low priority, small-size packet on the network under normal conditions, both sender and receiver will receive a good reward. The higher priority the packet, the more payoff will be received by the sender and receiver. We will try to boost both sides to complete a transfer with high priority packets. But also, the sender will pay the cost of the size of the package and the average RTT value. If the size of the packet is large and the RTT value is high, the sender should pay more from earnt payoff. On the other hand, if the receiver queue occupancy is high and the sender continues to send persistently, these messages are dropped and seen as wasted effort. Based on that, there will not be an award here, but the effort will be lost for S and RTT.

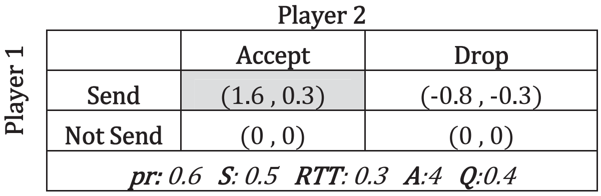
To describe the rationale behind the formulated game, if the receiver has a low occupancy rate (about 0), sending a packet will give a payoff associated with the priority of the packet. The sender always pays for the packet size and packet delay RTT from its award. An award A will be decisive for the behavior of the senders and receiver. Increasing A will make both sides greedy and both sides will try to complete a packet transfer even in a full target. Reducing A will make them to stay on the safe side and take less risk. Accepting the package for the receiver will award the same amount except for RTT as it is not important on the receiver side. The round trip delay is experienced by the sender.

Now we consider the payoffs when the sender sends the packet, and the receiver's queue is not occupied which puts the receiver into *"packet welcomer"* state. Sender will take same reward as it already executed its main duty (i.e., packet sending), however, the receiver will receive a penalty proportional to the priority of the packet if it happens to refuse accepting that packet, although it can do that. This would be an undesirable result. We want to avoid scenarios where the receiver does not accept packets even though the system is available. That's why we give some penalty to this action which forces the receiver to avoid this action and accept the packet to be granted a payoff as well.

Considering another scenario, where the receiver queue is occupied, and the sender sends the packet. In this case, the receiver tends to drop the packet and the sender pays for this unnecessary effort. In the payoff formulas, the value multiplied by Q (in the right-hand side) dominates when Q value converges to one, and the value multiplied by 1-Q (in the left-hand side) dominates when Q value converges to zero. If the packet is very important, the penalty of dropping will be high accordingly. Also, in the sender side, the penalty of sending high priority packet is not too much even if the receiver queue is highly occupied.

Table II. illustrates an example scenario with the given parameters. Here, the Nash equilibrium is (Send, Accept), because both the sender and the receiver will get highest payoffs and not tend to change their strategies. For more scenarios, [9] can be referred.

TABLE II. A SIMPLE GAME WITH PAYOFF TABLE FOR HIGH PRIORITIZED PACKET



		Player 2	
		Accept	Drop
Player 1	Send	(1.6 , 0.3)	(-0.8 , -0.3)
	Not Send	(0 , 0)	(0 , 0)
	pr: 0.6 S: 0.5 RTT: 0.3 A:4 Q:0.4		

Further details of AQMoT, such as how priority values are assigned and what to do with the packets that are decided to be not send will be described in the next section. Afterwards, we will describe details of a real experimental setup, in which we implement both the scenario where no queue management algorithm is employed, and the scenario where AQMoT is utilized.

III. AQMoT PROTOCOL DETAILS

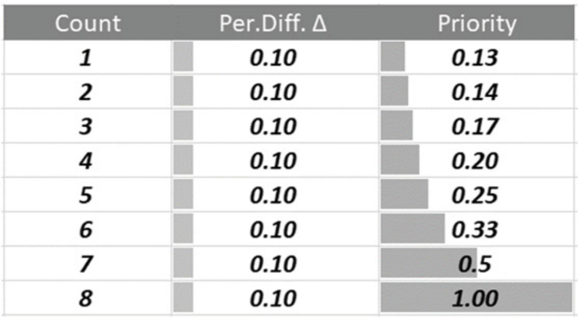
In this section, we provide some details of AQMoT that were not included in [9] and left as future work. AQMoT is a content aware algorithm, which assigns priorities to the packets according to their contents. For example, if a sensor value is not changed or slightly changed compared to the previously sent value, its priority should be not high. But time is also important here, such that packets of a sensor should get more priority if the time passed after sending the last packet of that sensor increases. To handle this, we propose an algorithm for setting the appropriate priorities of packets according to the changes in the content and time.

Priority values for each sensor packet are calculated according to the formulas given in (1) and (2). We calculate the percentage difference Δ between the current value generated by the sensor and the value in the last sent packet. We also define a parameter δ, such that when Δ exceeds δ value, packet priority is set to 1. δ value may differ according to the sensitivity to change for different types of sensors. For some sensors 0.1% is a significant change, whilst for some other 1% change is not significant. Decreasing the δ value would increase the sensitivity to change (i.e., very slightly changes will increase the priority for the next packets). Another variable is Count, which is set to zero after sending a packet, and incremented by one for each "Not Send" action.

$$\hat{P} = \frac{1}{10\,(\delta - \Delta) - Count} \qquad (1)$$

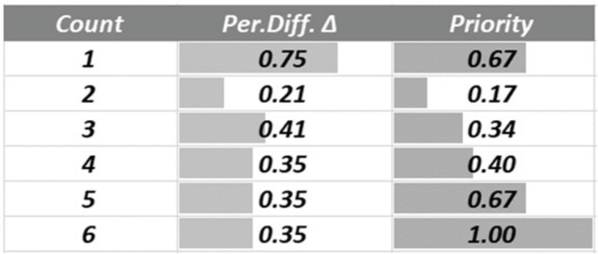
$$P = \begin{cases} \hat{P} & \text{if } 0 \le \hat{P} \le 1 \\ 1 & otherwise \end{cases} \qquad (2)$$

The formula given in (2) ensures that the priority increases when new data differs more compared to the last sent one, and it also increases after each deferred transmission. It also guarantees that the priority of a packet is always 1 after Count value reaches to 10. This prevents the scenarios where a sensor does not send any data for a long time. Fig. 3, 4 and 5 illustrates some priority values for various transmission scenarios. By default, δ is assumed to be 1. Fig. 3 shows a scenario where the sensor value slightly changes (0,1%) after sending a packet and it remains unchanged for next periods. In this case, priority increases to 1 after the eighth "Not Send" decision. Fig. 4 shows a more random scenario. The effect of the Count value (number of deferrals) increases exponentially. The first packet has the highest Δ value, its priority is relatively high, but not 1. Although the sixth packet has a lower Δ value, its priority is 1. Fig. 5 shows another scenario where the priority reaches to 1 in a shorter time.



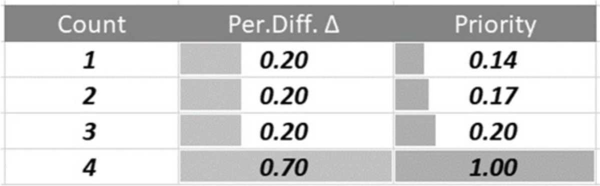
Count	Per.Diff. Δ	Priority
1	0.10	0.13
2	0.10	0.14
3	0.10	0.17
4	0.10	0.20
5	0.10	0.25
6	0.10	0.33
7	0.10	0.5
8	0.10	1.00

Fig. 3 Change in the priority values when all the packets have the same percentage difference of 0.1% compared to the last sent value



Count	Per.Diff. Δ	Priority
1	0.75	0.67
2	0.21	0.17
3	0.41	0.34
4	0.35	0.40
5	0.35	0.67
6	0.35	1.00

Fig. 4 Change in the priority value for various Count and Δ values



Count	Per.Diff. Δ	Priority
1	0.20	0.14
2	0.20	0.17
3	0.20	0.20
4	0.70	1.00

Fig. 5 Change in the priority value for various Count and Δ values

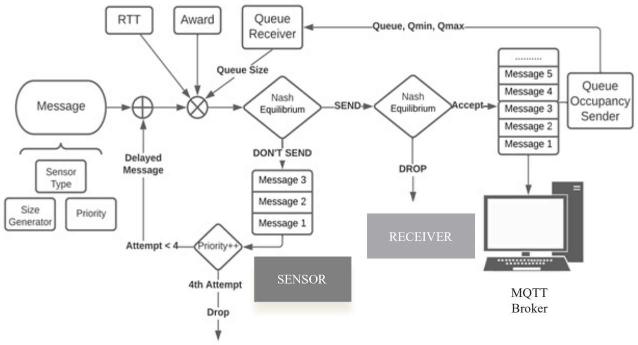


Fig. 6 Flowchart of our System

Fig. 6 illustrates the flowchart of the system. After creating a message object, sender makes a decision to whether send it immediately to the receiver or not. Here it checks the Nash Equilibrium of the game using the feedback information obtained from the receiver. It sends the packet if its action in the Nash Equilibrium is "Send". Otherwise, if it is "Don't Send", this packet is pushed to the Delayed Queue. Purpose of this queue is to resend this message after a delay. Duration of this delay is determined proportional to the absolute value of greatest negative payoff of (Send, Accept) action profile in the payoff table of the game. We multiply this value with 100 ms and delay is determined as the resulting value. For example, if the payoffs in (Send, Accept) action profile are (-2.4, -1.2), the packet is delayed for (2.4*100) ms. The choice of 100 ms is obtained empirically in our realistic test scenarios, but the best coefficient may be determined for different systems. The packets in the delayed queue are tried to be retransmitted after waiting for the determined delay value. Before each retransmission attempt, the priority of these delayed packets are incremented by 0.05. If the same message cannot be sent four times in a row, the message is permanently dropped.

After sending a message to the receiver successfully, this message is inserted to the queue and forwarded to the MQTT broker. At the receiver side, queue size is broadcasted to all the clients in constant intervals. These intervals may depend on the system characteristics, but it should be short enough to ensure that the client's knowledge is up to date.

For each packet, following fields are included: (i) *topic* (string), which indicates function of the sensor, and its identity; (ii) *message* (float/boolean), which indicates the sensor value to be sent; (iii) *operation* (int), which is an attribute for deciding how to process the message; (iv) *size* (float), the size of the packet; (v) *delayed* (boolean), which is an attribute that indicates whether the packet was "Not Sent" and delayed due to the game theoretic decision making; (vi) *priority* (double), the priority of the packet; (vii) *initialPriorityifDelayed* (double), which is required to recalculate priorities for delayed packets; (viii) *counter* (int), which indicates how many subsequent messages of a sensor is not sent; and (ix) *rtt* (double), the current round trip time.

IV. Experimental setup

We simulated the system with two Raspberry Pi's and three laptops which creates our desired IoT environment. Two Raspberry Pi's and two laptops emulates the sensors (sender-side) that send data to the server (receiver-side) pretending to be a gateway with different intervals. The sensors have different attitudes, and they are classified as dumb, normal, burst and random sensors. Dumb sensors send data rapidly and redundantly in small intervals, and these sensors may fill the queue at destination side because of these redundant efforts. Normal sensors send data two times slower than dumb sensors. Burst sensors send data with inconsistent time intervals and they send many messages on a sudden. Finally, random sensors send messages randomly at irregular intervals, which means that they may be sometimes slower and sometimes faster than normal sensors.

Both Raspberry Pi's and all three laptops publish values read from a predefined dataset simultaneously to the server via up to 18 sensors. These clustered sensors and server are located in different places and communication is provided with Hamachi [14] which builds a private LAN amongst computer scattered around the world physically while communicating via Internet. This approach served to provide unstable network conditions. The experimental IoT system is shown in Fig. 7. We note that Raspberry Pi's behave like many clustered sensors that are running on different connections. Each sensor is emulated by a different thread running on the Raspberry Pi, reading values from dataset and publishing these to the server.

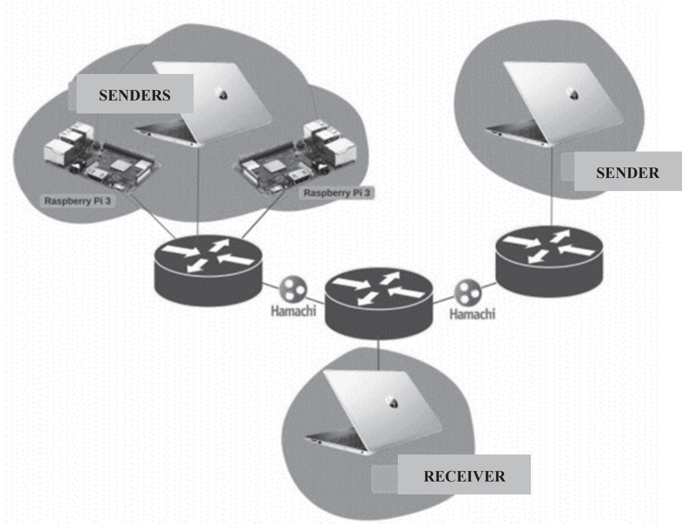


Fig. 7 Experimental IoT System

In the server-side, the server which implements MQTT broker, listens a specific port and waits sensor values. The server is multi-threaded so that for every coming message object, server creates a new thread and receives the message. Received messages will be placed in the queue. Maximum queue size is set to 100. Message sizes are assigned randomly between 0 and 1. In order to simulate edge processing, we add some sleep time for the threads for each message that is proportional to its size.

It is important to use real sensor values to better reflect content awareness of the proposed algorithm. For this purpose, we decided to use IoT dataset provided by Zamora-Martinez et al. [15] which includes many real sensor values that are used for indoor temperature forecasting. In our experiments, the sensor values for 18 different sensors are all obtained from this dataset. Some of the topic names of the sensors are weather temperature, CO_2 level, weather moisture level, etc. Temporal changes of the sensor values in the dataset are directly used to simulate variation in the sensor values, hence to define the priority values the packets.

To create a more realistic experimental setup, we create an IoT environment that consists of three subsystems in different locations, instead of restricting to a single local network. These three subsystems behave like clustered sensors or server, with the following hardware components:

- The first subsystem consists of a single computer. This computer serves as the server. It receives incoming messages.
- The second subsystem consists of a computer and two Raspberry Pi's. This system acts as a multi-client including clustered sensors. Majority of data flows through this system since it includes high number of sensors.
- The third system consists of a single computer. This computer acts as a client also.

The first setup, the server, is in a place located in Tokat, Turkey which is 700km away from the clients. The second setup, the multi-client, is located in Bahçelievler district of Istanbul and establishes a connection with the server. The third setup, the client 2, is located in Sariyer district of Istanbul, and establishes a connection with the server and a virtual, private LAN has been established between server and clients with Hamachi as mentioned previously.

Now we give further detail on different types of sensors used in the experiments.

Dumb Sensor: This type of sensors continuously sends messages to the server at 400 ms intervals.

Regular Sensor: This type of sensors continuously sends a message to the server at 800 ms intervals.

Burst Sensor: This type of sensors goes into sleep mode for a duration that is uniformly random between 10 seconds and 20 seconds, after sending nearly 70 messages at 200 ms intervals. These sensors significantly increase queue occupancy in a short period of time.

Random Sensor: In this sensor type, the sensor sends a message to the server at random intervals between 15 ms and 900 ms.

For each client we created, we define a sensor type. The client connects and sends data to server with the features of associated sensor type. On the developed test environment, we applied various test scenarios and evaluate performance of AQMoT algorithm. We compare performance of AQMoT with baseline algorithm which do not employ AQMoT, where sensors send all the generated data, and the receiver drops the packets if the queue is full. Test results are described in the next section.

V. TEST RESULTS AND DISCUSSION

In the described experimental setup, three different test scenarios are implemented, such as low, medium and high intensity systems. These scenarios are obtained by changing the sensor types, while number of sensors are constant (18 sensors are used in all tests). For each scenario, we ran the tests for both AQMoT algorithm and the baseline algorithm. During all tests, we tried to keep the environmental conditions constant for both cases.

1) *Low Intensity System*

In low intensity system, distribution of 18 sensors is decided as follows:

- 2 Dumb sensors
- 10 Random sensors
- 3 Regular sensors
- 3 Burst sensors

Fig. 8 illustrates the queue occupancy of the server in a low intensity system. It is observed that AQMoT significantly reduces the queue occupancy in the server side. Fig. 9 shows the histogram for the number of dropped packets versus priority of packets. Here both sender-side and receiver-side packet drops are counted. In AQMoT, almost all of the packets are dropped in the sender side, i.e., they are not sent. It is observed that only low priority packets are dropped, and all the packets with priority 1 are successfully sent. On the other hand, when AQMoT is not employed, 1.8% percent of the highest priority packets are dropped, although the system is not intensive. AQMoT algorithm prevents dropping the high priority packets by not sending a portion of low priority packets and avoiding overload in the buffers due to unnecessary effort made by dumb sensors.

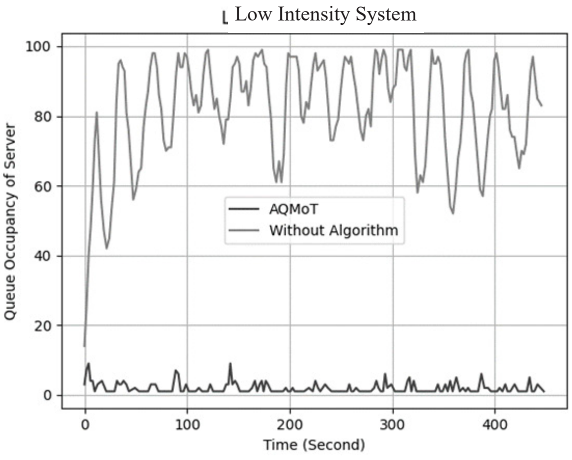


Fig. 8 Queue occupancy in a low intensity system when AQMoT is employed and not employed

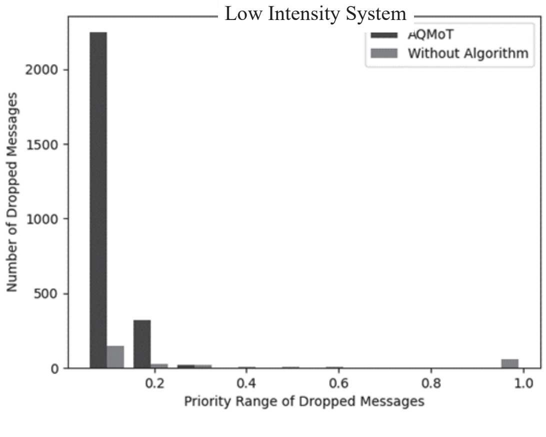


Fig. 9 Number of dropped packets versus packet priorities in low intensity system

2) *Medium Intensity System*

In medium intensity system, the distribution of 18 sensors is as follows:

- 2 Dumb sensors
- 8 Random sensors
- 3 Regular sensors
- 5 Burst sensors

Fig. 10 illustrates the queue occupancy of the server in a medium intensity system. Queue occupancy is higher for both cases compared to low intensity system, but still AQMoT provides queue occupancy level below 20%. Fig. 11 shows the histogram for the number of dropped packets versus priority of packets. It is observed that number of random drops increase, and larger number of high priority packets are dropped when AQMoT is not employed. When AQMoT is employed, again the number of packet drops increased to avoid high queue occupancy, but this increase is observed in low priority packets. All the packets with priority of 1 are successfully sent. On the other hand, when AQMoT is not employed, 22.17% of such packets are dropped. These results suggest that AQMoT perform well in a medium intensity system.

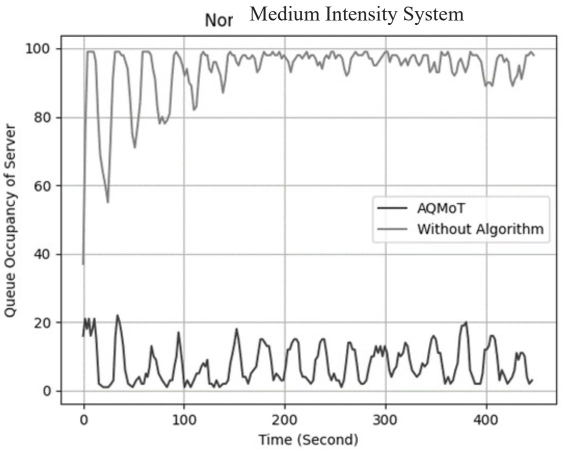


Fig. 10 Queue occupancy in a medium intensity system when AQMoT is employed and not employed

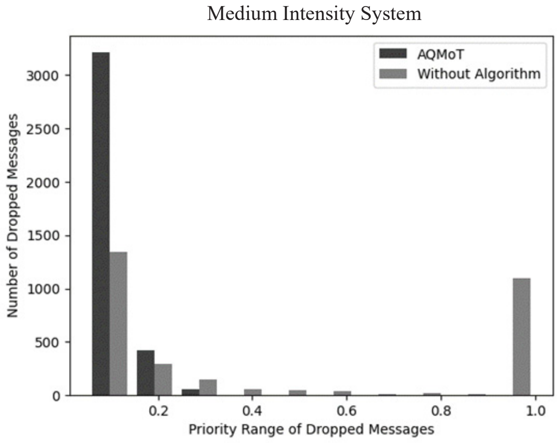


Fig. 11 Number of dropped packets versus packet priorities in medium intensity system

3) *High Intensity System*

In high intensity system, the distribution of 18 sensors is as follows:

- 12 Dumb sensors
- 0 Random sensor
- 0 Regular sensor
- 6 Burst sensors

Fig. 12 shows the queue occupancy of the server in a high intensity system. It is observed that when AQMoT is not used, the server queue is almost full all the time. When AQMoT is employed, queue length increases but still it is in tolerable levels. It fluctuates between 20% and 60%. Fig. 13 shows the histogram for the number of dropped packets versus priority of packets. When AQMoT is not employed, almost half of the packets (43.33%) with priority 1 are dropped. When AQMoT is used, most drops are encountered by redundant low priority packets, and only 1.06% of packets with priority 1 are dropped. These results suggest that AQMoT protects significant packets even in a highly congested network.

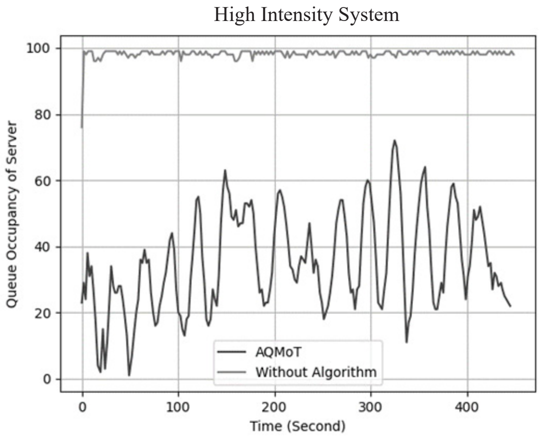


Fig. 12 Queue occupancy in a high intensity system when AQMoT is employed and not employed

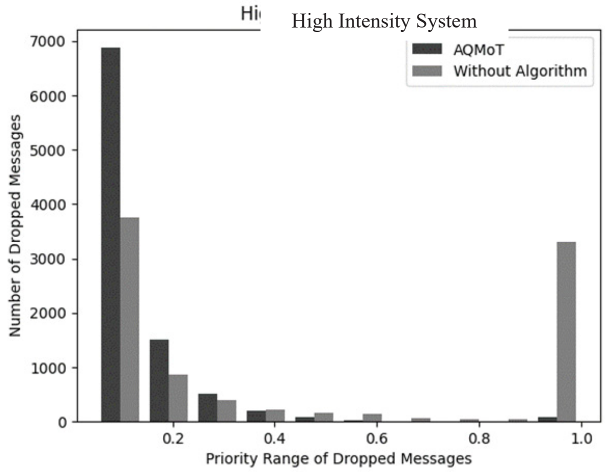


Fig. 13 Number of dropped packets versus packet priorities in high intensity system

4) *Discussion*

Table III. illustrates the drop rate of highest priority messages (i.e. with priority of 1) in low, medium and high intensity networks.

TABLE III. DROP RATE OF PACKETS WITH PRIORITY OF 1 COMPARED WITH AQMOT AND WITHOUT AQMOT

	Low Intensity	Medium Intensity	High Intensity
W/o algorithm	1.83%	22.17%	43.33%
AQMoT	0%	0%	1.06%

Experimental study shows that the following performance results are obtained with AQMoT algorithm, that are aligned with the initial design goals.

i. The queue congestion level in server side is decreased about 44%.

ii. Rate of the successful delivery of high priority data is significantly increased. In the low intensity and medium intensity systems, all the high priority data are sent successfully. In the high intensity system, 98.94% of the high priority data are sent successfully.

iii. Transmission rate of redundant data (with repeating values) are decreased, thus avoiding needless efforts, reducing energy consumption and relaxing the traffic in the receiver queue.

To sum up, it is always an important issue to have positive discrimination over prioritized data rather than unimportant ones in IoT world. Here, we put effort on flowing prioritized data in the system by also reserving the most of system resources for them, while getting rid of many of unimportant ones when necessary to create some relief decreasing the intensity on the system.

Without AQMoT in IoT setup, all the drop counts shared by any kind of message in terms of priority, and unimportant messages can overload the system by also overriding the prioritized ones. AQMoT also prevents exploiting by the sensors which send lots of redundant data.

VI. CONCLUSION

In this study, our primary objective is to evaluate previously proposed queue management algorithm, AQMoT, which is specifically designed for IoT environment, and try to avoid futile efforts made by dumb IoT sensors by restraining their eagerness of sending too much redundant data. We propose a method for prioritizing IoT data and designed a detailed workflow of the system. We implement AQMoT in a realistic test environment and perform extensive set of simulations/experiments.

In the simulated low intensity and medium intensity systems, all important data, with priority equals to 1, was delivered in full. This shows that AQMoT performs well to avoid loss of high priority data. In the highly occupied system, 98.96% of important data was delivered. This percentage depends on the environment, and it can vary considerably in different occupation scenarios. However, it is observed that as the traffic intensity level increases, the contribution of AQMoT to avoid loss of high-priority data also increases. AQMoT prevents queue occupancy of unimportant/redundant data effectively.

ACKNOWLEDGMENT

This work is supported by Koç Digital R&D Center.

REFERENCES

[1] Ataç, C., and Akleylek, S. "A survey on security threats and solutions in the age of IoT", Avrupa Bilim ve Teknoloji Dergisi, No. 15, 2019 (pp. 36–42).

[2] Floyd, S., and Jacobson, V. "Random early detection gateways for congestion avoidance." IEEE/ACM Transactions on networking, Vol. 1, No. 4, 1993, (pp. 397–413).

[3] Feng, W.-C., Kandlur, D. D., Saha, D., and Shin, K. G. "A self-configuring RED gateway", IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320), Vol. 3, IEEE, 1999 (pp. 1320–1328).

[4] Hassan, M., and Jain, R. "High performance TCP/IP networking", Vol. 29, Prentice Hall Upper Saddle River, NJ, 2003

[5] Feng, W.-c., Kapadia, A., and Thulasidasan, S. "GREEN: proactive queue management over a best-effort network." Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE, Vol. 2, IEEE, 2002, (pp. 1774–1778)

[6] Pan, R., Natarajan, P., Piglione, C., Prabhu, M. S., Subramanian, V., Baker, F., and VerSteeg, B. "PIE: A lightweight control scheme to address the bufferbloat problem." IEEE 14th International Conference on High Performance

[7] Pan, R., Prabhakar, B., & Psounis, K. "CHOKe-a stateless active queue management scheme for approximating fair bandwidth allocation." In Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, 2000 (Cat. No. 00CH37064) (Vol. 2, pp.942-951). IEEE.

[8] Abbas, G., Manzoor, S., & Hussain, M. "A stateless fairness-driven active queue management scheme for efficient and fair bandwidth allocation in congested Internet routers." Telecommunication Systems, 67(1), 3-20, 2018

[9] K.Aytaç, Ö.Korçak, "AQM-of-Things: Special Queue Management Approach for Internet of Things", European Journal of Science and Technology (Avrupa Bilim ve Teknoloji Dergisi), 2020.

[10] Aumann, R. J. "What is game theory trying to accomplish?", Frontiers of Economics, edited by K. Arrow and S. Honkapohja, 1985

[11] Raspberry Pi official website, Raspberry Pi, Web: https://www.raspberrypi.org [Online][Accessed 30- Aug- 2021]

[12] Naik, N. "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP." IEEE international systems engineering symposium (ISSE), IEEE, 2017. (pp. 1–7).

[13] Kuhn, H. "Extensive games and the problem op information" In H. Kuhn and A. Tucker, editors, Contributions to the Theory of Games, 2016 (pp. 193–216).

[14] Hamachi official website, LogMeIn Hamachi, Web: https://www.vpn.net [Online][Accessed 30- Aug- 2021]

[15] F. Zamora-Martínez, P. Romeu, P. Botella-Rocamora, J. Pardo, "On-line learning of indoor temperature forecasting models towards energy efficiency", Energy and Buildings, Volume 83, Pages 162-172, ISSN 0378-7788, 2014