InNetTC: In-Network Traffic Control for Full IP Sensor-Actuator Networks

Hideya Ochiai
The University of Tokyo/NICT
jo2lxq@hongo.wide.ad.jp

Yuuichi Teranishi
NICT/Osaka University
teranisi@cmc.osaka-u.ac.jp

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

ABSTRACT

The Internet protocol (IP) is now embedded into sensor-and-actuator nodes these days, rising the new age of full IP sensor-actuator networks. Though we can make them available on the Internet space with such embedded TCP/IP stacks, the appropriate link-layer media for networking them in facilities are not Ethernet. In stead, it is considered to be IEEE802.15.4 and RS485, which are known as lossy, unstable and narrow link media. This sometimes cause fatal problems especially traffic congestion at the router of connecting such narrow link in some application senarios. This paper proposes the architecture and design of In-Network Traffic Control (InNetTC) scheme that solves this issue. We carried out experiments with the prototype of InNetTC-enabled router. The result shows that InNetTC achieved 100% success of task execution for such application scenarios, indicating that InNetTC plays an important role for implementing applications on such full IP sensor-actuator networks.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network Communications; C.2.2 [**Network Protocols**]: Protocol Architecture

General Terms

Algorithms, Design, Performance

Keywords

Sensor-Actuator Networks, TCP/IP, Narrow Links, Traffic Congestion Control

1. INTRODUCTION

With the rise of TCP/IP embedded technologies, more and more sensors and acutators are getting available on the Internet space. This has enabled the world of Internet of Things[4, 1, 8, 18] and machine-to-machine (M2M) communications[6, 13, 17, 5]. They are now implemented into SmartGrid[7], building automation[2, 10], management of public infrastructure and environmental monitoring. This is widely known and accepted not only to the academic researchers but also to industries and governments. But, the nature of M2M communication is very different from the traditional TCP/IP network. We have to provide TCP/IP reachability not to our laptops or desktops but to the facilities (i.e., sensors and actuators deployed on ceilings or behind walls).

As Ethernet is not an appropriate media to physically network them, more proper media such as IEEE802.15.4 and RS485 is mainly used in the practical deployment. However, those media provide narrow-bandwidth in communication (e.g., 9600bps, 115.2kbps) and this sometimes causes traffic congestion at the router of the narrow-link media, which leads to critical failures in carrying out monitoring and controlling applications.

This paper proposes an In-Network Traffic Control (InNetTC) for full IP sensor-actuator networks, which can avoid this potential traffic congestion at the router. By avoiding such traffic congestion, this control scheme can also avoid application-level critical failures.

To deploy IP sensors and actuators in the real world (for example into a building), as we discussed, Ethernet is not an appropriate communication media. For example, in the case where we deploy 10 environmental sensors, light ON/OFF controllers, switches, motion detectors and electricity power meters in a room, we have to deploy Ethernet cables from an Ethernet hub to each sensor or controller one-by-one. This is unrealistic and will not be widely-accepted even in the future.

It is widely-considered among the practitioners of this area that only lossy, unstable, narrow-band communication media are available today for such environment. IEEE802.15.4 is considered to be a promising wireless media, and to deliver IP packets over it, 6LoWPAN[12] and ZigBeeIP[19] is proposed. Traditionally, system integrators have been using single twisted-pair physical communication media such as RS485 and Lonworks[10] because they allow to connect devices in cascade manner. Cascade cabling is the optimal form of deployment and accepted by them. Here, IP packets could be also delivered over such networks[3]. And, because of the lightweightness of UDP, UDP-based approaches are attempted on such networks [16].

In this paper, we focus on traffic congestion caused by the nature of narrow-links. When a remote requester accesses to multiple (e.g., hundreds of) actuators as a task of an application, it generates a lot of IP packets in a very short time causing traffic congestion and huge delay at the gateway of the narrow link. This delay sometimes leads to request retry at the requester, which causes further traffic congestion. We analyze this problem in section 3 of this paper.

InNetTC allows traffic control by pause request for a specific network address ranges. When a narrow link is congested, the router notifies this status to the requester so that it waits to send IP packets to such network segment. As this traffic congestion control should be made for a network address range, end-to-end traffic control scheme is not applicable. To manage this, InNetTC generates the pause request by the router: i.e., in network. Thus, we call this in-network traffic control.

This paper is organized as follows. In the next section, we address related works. Section 3 provides the model of a full IP sensor-actuator network. We propose in-network traffic control scheme in section 4. Section 5 shows our performance evalution. Section 6 provides the discussion on the approaches we have made, and section 7 summarizes this paper.

2. RELATED WORK

Many protocols for Internet-of-Things and M2M communications have been proposed and developed at the application layer. FIAP[15] introduced the concept of Gateway(GW) for networking sensors and actuators over the Internet. The GW typically has Ethernet port and works as a TCP/IP communication edge. It provides access with XML messages over HTTP at the Internet side. The GW is usually equipped with non-IP field-buses; e.g., it allows access to sensors and actuators with Modbus[11] serial communication protocol over RS485. Web-based systems such as oBIX[14] and BACnetWS[2] also takes this approach for networking them over the Internet.

A full IP sensor-actuator network takes different approaches for connecting them. It (1) assigns IP addresses to any sensors and actuators even if they are attached on RS485 network or IEEE802.15.4 network, and (2) allows to exchange IP packets with them. CoAP[16, 9] and ZigBee SEP2.0[19] take this style for connecting sensors and actuators to the Internet space with using 6LoWPAN[12] and ZigBeeIP[19] for the IP layer. Because of the cost and delay of radio communications of IEEE802.15.4, they strongly recommend to use UDP/IP protocol for accessing the devices.

In-network traffic control has not yet been proposed for those UDP/IP-based full IP sensor-actuator networks. We encounter traffic congestion problem at the router when we use such narrow link. In this paper, we propose the scheme of traffic control, and provide basic analysis of it.

3. A FULL IP SENSOR ACUATOR NETWORK

3.1 Definition

This subsection defines a full IP sensor-actuator network we assume in this paper. The definition described here is a simplified model – in the typical implementation, we must specify it in mode detail.

Let d be a device (i.e., a sensor or an actuator). It has an IP address, to which we can access from the Internet. A device could have multiple IP addresses in some implementations, but in this paper, we assume only one address for one device for simplification.

A sensor (and actuator) has *get* and *set* method with which a requester can get and set data in binary format. We denote them by $d.get()$ and $d.set(value)$ in this paper(here, $value$ is data in binary form going to be set into the actuator d). Sensors actually don't implement (i.e., empty codes in) *set* method because it is a read-only device.

$d.get()$ sends a request UDP datagram to the device, and it returns a response UDP datagram with data in binary form. It waits until it gets the response or timeout occurs. More formally, we define it as follows.

Let $p_{(r,d)}$ be a packet from requester r to sensor d. By switching r and d (i.e., $p_{(d,r)}$) we can denote a packet from sensor d to requester r. And, in a full IP sensor-actuator network, we define four types of packet: (1) get request (2) get response (3) set request and (4) set response. We denote them by $getReq$, $getRes$, $setReq$ and $setRes$ respectively.

When the $d.get()$ is invoked, it generates a packet which type is $getReq$: i.e., $p_{(r,d)}.type = getReq$. When the sensor d receives the packet, it generates a response packet, which type is $getRes$: i.e., $p_{(d,r)}.type = getRes$. $d.get()$ provides the value after it has received the packet or TIMEOUT notification if it has not received it in the certain time. This behaviour can be defined as the following pesudo-code.

pesudo-code of function $d.get()$

```
function d.get(){
  p=new packet(r,d)
  p.type=getReq
  send(p)

  q=recv(p)
  if(q==TIMEOUT){
    return TIMEOUT
  }
  if(q.type==getRes){
    return q.content
  }
  return ERROR
}
```

In a typical full IP sensor-actuator network, we assume, sensors and actuators are usually not directly connected to the Ethernet. Instead, it uses other communication media such as RS485, RS232C and IEEE802.15.4. This feature is not made explicit in this model yet. It is discussed in the following sub sections.

3.2 Typical Access Algorithm

Let's consider a power monitoring task of an application. This task takes a current snapshot of hundreds of power sensors. A requester sends requests to those hundreds of sensors and gets values from them. We analyze the algorithm that performs this task.

Let $S(r)$ be the set of sensors, which the requester r wants to communicate with. To read the current status of all sensors, typically, the requester implements the algorithm below. Here, we denote V a set of key-value pairs, which stores the result.

```
V = Φ
Forall d ∈ S(r) in sequentially or in parallel,
    max_retry=3
    do{
       value=d.get()
    }while(value==TIMEOUT && max_retry-->0){
    V = V.put(d, value)
EndForall
```

This algorithm is awared of packet-loss management: i.e., retries three times at maximum if it does not receive a reponse packet in a certain time duration.

There are two types of implementations of this algorithm: i.e., sequential implementation and parallel implementation. A sequential implementation sends request and receives response one-by-one for each sensor incrementally. A parallel implementation sends requests and receives repsponses at the same time independenly. There are problems in either sequential-case or parallel-case.

3.2.1 Problem in Sequential-Case

Let $RTT(r, s)$ be a round trip time between the requester r and the sensor d.

We denote l the number of packet loss the requester encounters, and T the duration of the timeout. Then, the total delay D_{total} becomes

$$D_{total} = \sum_{d \in S(r)} RTT(r, d) + l \times T \quad (1)$$

Even if there was no packet loss (i.e., $l = 0$), $RTT(r, d)$ takes tens of milliseconds on the narrow link. If the number of $S(r)$ is large (e.g., hundreds or thousands), it takes seconds or ten seconds to complete the task.

Sensing itself could be tolerant for this delay in many applications, but this is critial for some controlling applications which use *set* method. If this delay happens when controlling ON/OFF lights, people in the room become unconfortable.

3.2.2 Problem in Parallel-Case

If it is in parallel, it generates a burst traffic, which might cause traffic congestion especially at the router for narrow-link media. To consider the worst situtation, let's assume the case that:

$$\forall d \in S(r), ip(d) \in network(m) \quad (2)$$

Here, the function $ip(d)$ gives the IP address of the device d. m is a link media, which corresponding IP address range is given by $network(m)$. Thus in this case, to perform the task, the requester has to communicate with the sensors on the same communication link. Here, the link is narrow such as 9600bps.

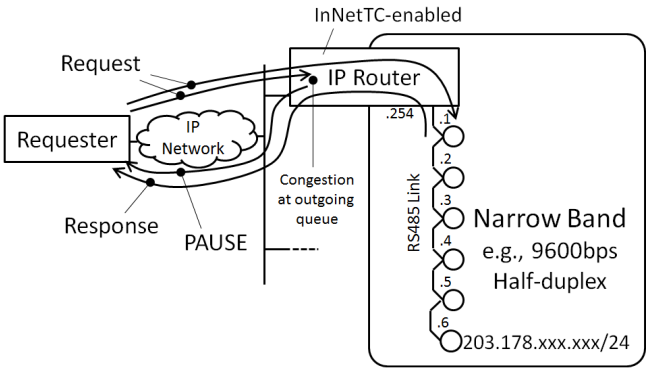

Figure 1: Architecture for In-Network Traffic Control. The InNetTC-enabled router normally just forwards IP packets between a narrow link and a fast link. If the router detects traffic congestion for the narrow link, the router generates PAUSE packet to the requester to stop further requests temporarily.

In this situation, burst traffic generated by the requester, causes traffic congestion at the router. Then,

1. If the buffer size for outgoing interface is not enough for holding all of the received request packets, the router destroys them.
2. The RTT increases linerly with the increase of the buffer queue length. If the queue becomes too long, timeout occurs at the requester side, sending retry request again, which causes further traffic congestion. And in the worst case, the requester cannot receive any responses from some sensors and abundan them after finishing the last retry.

4. IN-NETWORK TRAFFIC CONTROL

To manage these problems, this paper proposes a in-network traffic control for such a full IP sensor actuator network.

Figure 1 shows the target architecture. The requester sends UDP datagrams (IP packets) to sensors and actuators, and they reply to the requester. The router bridges IP packets between the Internet and the narrow-band link: e.g., RS485 or ZigBeeIP network.

Here, in-network traffic control works as follows:

active sending: The requester actively sends UDP requests to sensors and actuators in parallel. Usually, it targets at different IP addresses and carries out in a very short time.

pause request: The router sends pause request to the requester when it detects traffic congestion. This makes the requester temporarily stop packet sending (for sensors and actuators on the same narrow link).

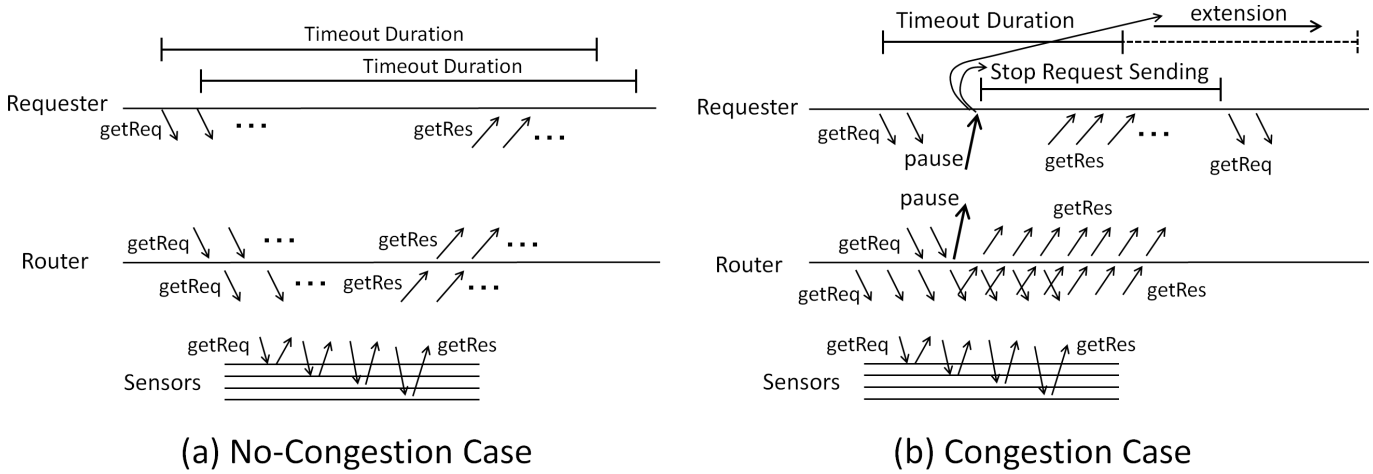

Figure 2: Behavior of InNetTC-enabled router and requester. (a) No-Congestion Case. (b) Congestion Case.

pause: The requester stops for a while according to the pause request received.

4.1 Pause Request

In section 3.1, we described that a full IP sensor-actuator network has four types of packets: i.e., *getReq*, *getRes*, *setReq*, *setRes*. Here, we define a new type of packet called pause request packet, which we denote by *pause*. This pause packet can contain IP address range information and pause duration. This packet should be generated at the router (not at the end device) as the response of *getReq* or *setReq*.

This is a pesudo-code of generating pause packet at the router.

Pause Request Generation

```
queue.push(p(r,s))
if( queue.length > PAUSE_THRESHOLD ){
  p = new packet(s,r)
  p.type=pause
  p.address_range = 203.178.135.0/25
  p.duration = queue.length × WEIGHT
  forward(p)
}
```

If the length of the router's outgoing queue exceeds the threshold, it triggers pause request packet generation. It contains the target IP address range and the estimated duration of delay. Here, terminologies used in the algorithm are:

p(r,s): packets received at the router (destination is a sensor or an actuator on the narrow-link).

queue: outgoing queue for the narrow link.

PAUSE_THRESHOLD: a parameter to start sending pause request.

WEIGHT: weight of time to estimate pause duration from the length of the queue.

forward(p): forwards packet p in the router (pushes into an appropriate outgoing queue).

203.178.135.0/25: a sample IP address range

Though *p.duration* is defined as a linear of *queue.length* here, it should be appropriately estimated depending on the features of the link layer.

The source address of the pause request is the IP address of the sensor though this packet is generated at the router. The requester can receive the pause request which source IP address is the sensor, even if it is actually sent from the router.

We take this approach because the requester cannot receive the pause request if the router sends it from the IP address of the router itself. For the requester, the IP address of the router is unknown in priori or even not authorized.

4.2 Requester with Pause Algorithm

In InNetTC, the requester controls traffic with the following algorithm:

pesudo-code of pause algorithm at Requester

```
function d.get() {
  p=new packet(r,d)
  p.type=getReq
  send(p)

  do{
    q=recv(p)
    if(q==TIMEOUT){
      return TIMEOUT
    }
    if(q.type==getRes){
      return q.content
    }else if(q.type==pause){
      global_wait(q.address_range, q.duration)
      continue
```

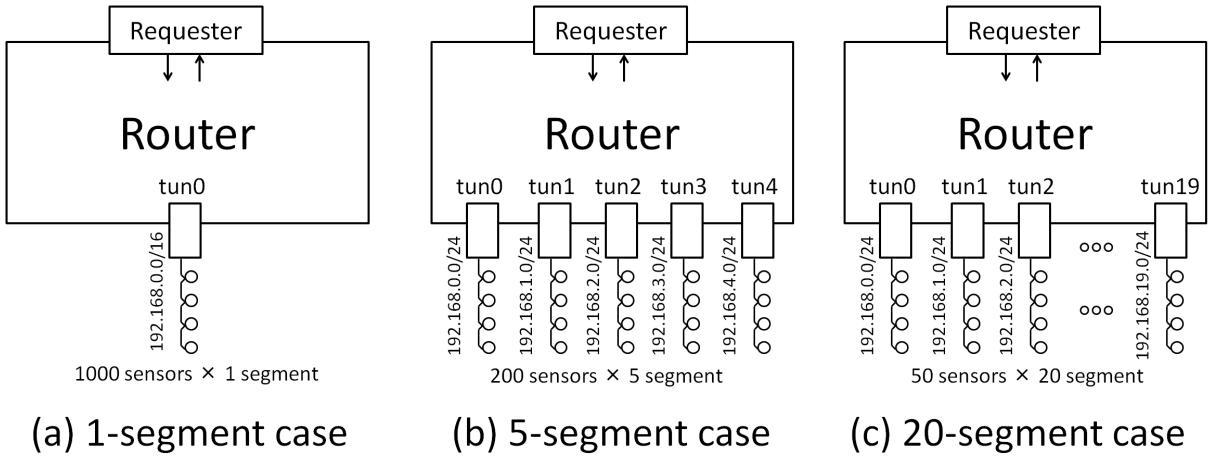

Figure 3: Network configurations of the experiment. Three types of network configurations has been applied to study the differences of successes, packets exchanges, queue length and duration of task execution.

```
    }
    return ERROR
  }while(true)
}
```

$global\_wait(address\_range, duration)$ stops carrying out all the *send* methods related to $address\_range$ for $duration$. It also extends timeout duration for receival of the response packet.

Figure 2 shows this action with the time chart.

(a) when the traffic is not congested at the router, the router just forwards IP packets between the devices and the requester.

(b) when the traffic is congested at the outgoing queue (for the narrow-link) of the router, it sends pause request back to the requester. Then, the requester stop sending further requests temporarily with extending the timeout duration for the response of the request. For example, if the timeout is configured as 1 second and it receives 5 seconds for pause, it waits at least 6 seconds (1+5 seconds) from the sensor.

5. EVALUATION

We evaluated InNetTC with regard to success rate, number of packets exchanged, queue length and duration of task execution. To evaluate it in these aspects, we implemented an InNetTC enabled router and carried out experiments with emulated sensors. The results show that InNetTC achieves reliable task execution with appropriate execution time by controlling the traffic with the pause algorithm.

5.1 Experiment Setting

Figure 3 shows the configuration of the experiment we setup for evaluation. We operated 1000 sensors in the field-networks by emulation, and the requester got values from those sensors with the InNetTC's pause algorithm. To emulate the sensors and the field-networks, we implemented them into TUN/TAP device(s), and it slowly processed the requested packets (i.e., *getReq* packets) and replied to the requester. The TUN/TAP devices also implemented InNetTC's pause algorithm, which generated pause request and sent it back to the requester based on the length of the outgoing queue and the PAUSE_THRESHOLD parameter setting.

To carry out experiments with different network environment, we have tested three types of field-network configurations: 1-segment case, 5-segment case and 20-segment case.

1-segment case: We attached one field-network (i.e., one TUN/TAP device) to the router, and assigned 192.168.0.0/16 to the field network. We deployed 1000 sensors to the single segment as 192.168.[0-4].[10-209] and the requester created and sent *getReq* requests to those sensors.

5-segment case: We attached five field-networks (i.e., five TUN/TAP devices) to the router, and assigned 192.168.x.0/24 (x=0,...,4) to each field-bus network. We deployed 1000 sensors to the five segments (200 sensors to each segment) as 192.168.[0-4].[10-209] and the requester created and sent *getReq* requests to those sensors.

20-segment case: We attached twenty field-networks (i.e., twenty TUN/TAP devices) to the router, and assigned 192.168.x.0/24 (x=0,...,19) to each field-bus network. We deployed 1000 sensors to the twenty segments as 192.168.[0-19].[10-59] (50 sensors to each segment) and the requester created and sent *getReq* requests to those sensors.

We changed the PAUSE_THREASHOLD parameter from 1 to 3000 and studied (1) success rate, (2) number of packets sent by the requester, (3) number of packets received by the requester, (4) max length of outgoing queue of the router and (5) duration of whole task execution. If the

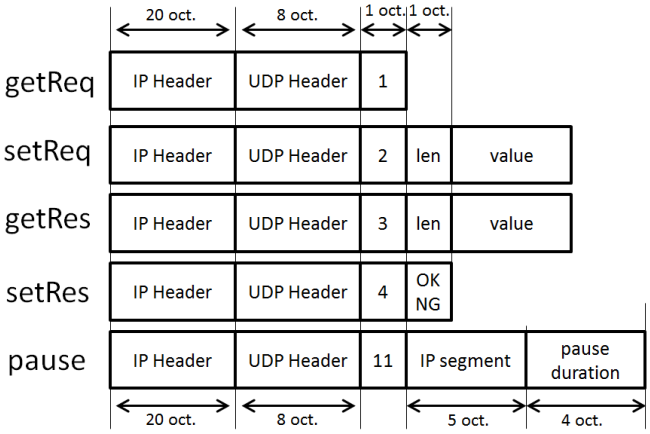

Figure 4: Protocol implementation of the full IP sensor-actuator network and InNetTC's pause mechanism.

Table 1: Configuration of the experiment

Parameter	Value
Process one packet in the field-network	30[msec]
Duration estimation	Number of packets in queue × 30 [msec]
Timeout	1000 [msec]
Maximum Retry	3
CPU	Intel Core 2 Duo U9400 1.4GHz
Memory	4 GByte
OS	Linux 2.6.17-2-686

PAUSE_THRESHOLD is large enough, it never triggers InNetTC's pause algorithm, so we could evaluate the differences between not-enabled case (i.e., PAUSE_THRESHOLD is 3000) and strongly-enabled case (i.e., PAUSE_THRESHOLD is 1).

For the experiment, we have implemented the application layer protocol on the UDP transport protocol as Figure 4. We assigned numbers for the message type (e.g., *getReq* for 1, *getRes* for 3), and for each type, we designed the structure as the figure. In this experiment, we did not assume packet loss on the Internet: i.e., between the requester and the router. So, we executed the requester on the same platform of the router. Table 1 shows the other parameters used in the experiment.

5.2 Success Rate, Packets and Queue Length

Figure 5, 6, 7 show the success rate, number of packets sent and received, and the maximum length of the queue.

In **1-segment case**, it has achieved 100% success rate (i.e., 1000 successes for 1000 sensors) when PAUSE_THRESHOLD was below 400. Especially, when PAUSE_THRESHOLD was less than 200, the requester has sent 1000 (or 1001 in some cases) *getReq* packets for 1000 sensors, and received *getRes* packets from all of them. It also received *pause* packets,

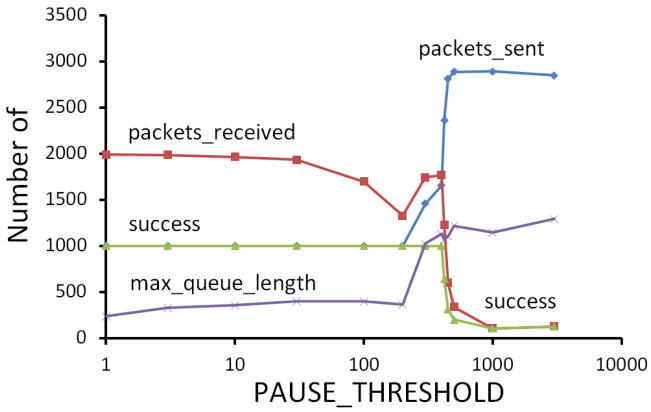

Figure 5: 1-segment case: the number of success, packets exchanged and queue length

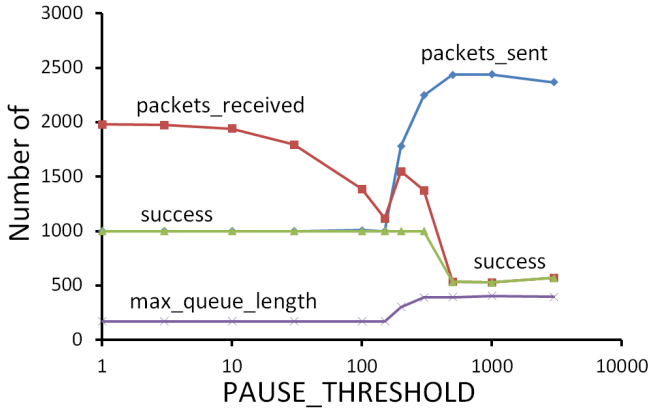

Figure 6: 5-segment case: the number of success, packets exchanged and queue length

which is shown as that the number of packet received is more than that of sent packets. When PAUSE_THRESHOLD was between 200 and 400, the requester sometimes resent *getReq* packets, indicating that the requester sometimes encounter TIME for their requests because it did not receive pause request. The maximum length of the queue also increased suddenly when PAUSE_THRESHOLD exceeded 200. When PAUSE_THRESHOLD becomes over 400, it suddenly increased the packets sent, and decreased success rate. This indicates TIMEOUT happened many times and the requester made retry, but finally it failed to recover and abundon to complete the task.

In **5-segment case**, the basic feature was the same as 1-segment case. It has achieved 100% success when the threshold was below 300. Around it increased over 150, it changed the behavior from pause request-based traffic control to traffic congestion at the router. Over 300, it suddenly increased the packets sent, and decreased success rate.

In **20-segment case**, it has achieved 100% success rate for any PAUSE_THRESHOLD settings. However, the behaviour of sending and receiving packets were different de-

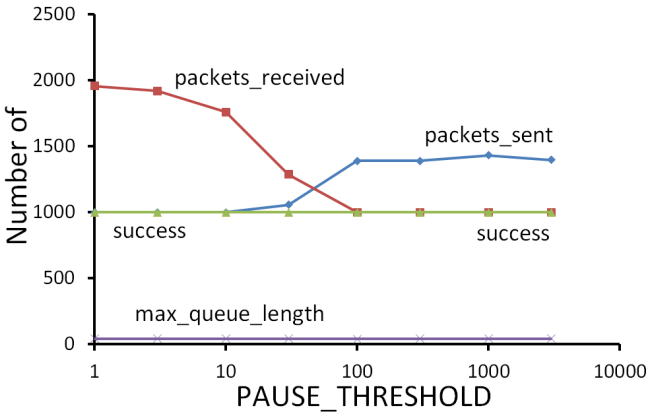

Figure 7: 20-segment case: the number of success, packets exchanged and queue length

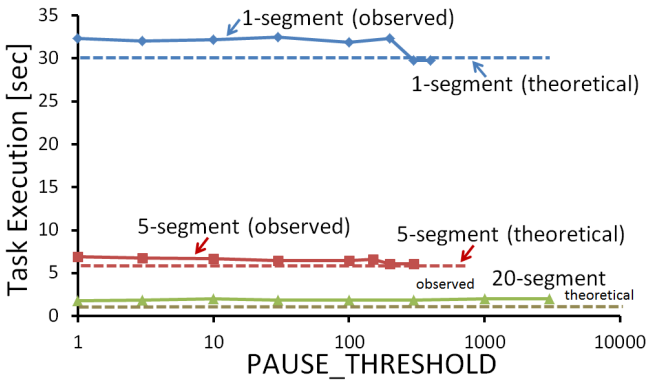

Figure 8: Total task execution time for the range of PAUSE_THRESHOLD: comparison with the theoretical estimation.

pending on the setting. When PAUSE_THRESHOLD was below 10, the requester has sent 1000 *getReq* packets for 1000 sensors, and received *getRes* packets from all of them. Where as, when PAUSE_THRESHOLD was over 100, the requester almost did not receive pause request, and it resent *getReq* packets. It got the *getRes* packets from the sensors for the first *getReq* packets and it did succeeded for all of the sensors. It could complete the task even without InNetTC's pause algorithm because each segment had only 50 sensors and the delay in the queue was not so large (as the graph indicates) to result in fatal error.

From these results, we can conclude that in InNetTC-enabled case, it can safely achieve 100% success at least for such network configurations. And, to surely enable InNetTC's pause algorithm, PAUSE_THRESHOLD should be set as 10.

5.3 Duration of Task Execution

Figure 8 shows the duration for executing the task (only successful cases are plotted). This graph also shows the theoretical duration time calculated from the configuration parameters. The theoretical duration for each case are calculated as follows. From the experiment setting, it takes 30 [msec] for one *getReq* packet processing. Thus, for 1-segment case, it takes 30 [sec] (30 [msec] $\times$ 1000 sensors). For 5-segment case, it takes 6.0 [sec] (30[msec] $\times$ 200 sensors for one segment), and for 20-segment case, it takes 1.5 [sec] (30 [msec] $\times$ 50 sensors for one segment).

As is shown in the figure, the duration observed were close enough to the theoretically estimated time for all the successful PAUSE_THRESHOLD cases, which means that the execution time does not rely on the PAUSE_THRESHOLD.

6. DISCUSSION

The pause algorithm proposed as the main part of InNetTC scheme is unique compared to the traditional traffic control approaches. For the requester to sensors and actuators, the pause request looks like that it comes from sensors or actuators because they use the IP address of them. However, it is actually generated by the router, which is "in-network" between the requester and the devices. This type of traffic control is different from TCP's algorithm (because TCP is end-to-end), and the network operation of bandwidth limitation.

In InNetTC, an intermediate router sends a pause request to the requester. This pause request actually plays another important role beyond stopping further requests to the network segment. It notifies that the intermediate router has certainly received the request. Then, the requester should consider that the request is certainly approaching to the devices and it should not send retry packets again unless it has exceeded the notified estimated delay.

With these two features, it achieved application-level task execution (i.e., getting values from 1000 sensors) successfully. The requester waited appropriately without sending both further unsent packets and further retry packets when it received pause packets from the router. This resulted in the success of the task execution.

7. CONCLUSION

In Network Traffic Control (InNetTC) has achieved 100% success of task execution for three types of configurations in a full IP sensor-actuator network. When the InNetTC was off, traffic congestion occurred at the outgoing queue for the field-bus of the router in some cases, and it caused fatal error at the application-level task execution. This result indicates that InNetTC's pause algorithm plays an important role to implement applications on full IP sensor-actuator networks.

8. REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[2] Bacnet - a data communication protocol for building automation and control networks. http://www.bacnet.org/.

[3] N. W. Bergmann, M. Wallace, and E. Calia. Low cost prototyping system for sensor networks. In *IEEE ISSNIP*, 2010.

[4] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi. Architecture and protocols for the

internet of things: A case study. In *IEEE PERCOM workshop*, 2010.

[5] H. Chao, Y. Chen, and J. Wu. Power saving for machine to machine communications in cellular networks. In *IEEE GLOBECOM Workshop*, 2011.

[6] M. Chen, J. Wan, and F. Li. Machine-to-machine communications: Architectures, standards and applications. *KSII TRANSACTIONS ON INTERNET AND INFORMATION SYSTEMS*, 6(2):480–497, feb 2012.

[7] Z. M. Fadlullah, M. M. Fouda, N. Kato, A. Takeuchi, N. Iwasaki, and Y. Nozaki. Towards intelligent machine-to-machine communications in smart grid. *IEEE Communications Magazine*, 49(4):60–65, apr 2011.

[8] S. Hong, D. Kim, M. Ha, S. Bae, S. J. Park, W. Jung, and J.-E. Kim. Snail: an ip-based wireless sensor network approach to the internet of things. *IEEE Wireless Communications Magazine*, 17(6):34–42, dec 2010.

[9] M. Kovatsch, S. Duquennoy, and A. Dunkels. A low-power coap for contiki. In *IEEE MASS*, oct 2011.

[10] Lonmark international. http://www.lonmark.org/.

[11] The modbus organization. http://www.modbus.org/.

[12] G. Mulligan. The 6lowpan architecture. In *ACM EmNets workshop*, 2007.

[13] D. Niyato, L. Xiao, and P. Wang. Machine-to-machine communication for home energy management system in smart grid. *IEEE Communications Magazine*, 49(4):53–59, apr 2011.

[14] Open building information exchange. http://www.obix.org/.

[15] H. Ochiai, M. Ishiyama, T. Momose, N. Fujiwara, K. Ito, H. Inagaki, A. Nakagawa, and H. Esaki. Fiap: Facility information access protocol for data-centric building automation systems. In *IEEE INFOCOM M2MCN workshop*, 2011.

[16] Z. Shelby, K. Hartke, and C. Bormann. Constrained application protocol (coap) draft-ietf-core-coap-18, jun 2013.

[17] J. Wan, D. Li, C. Zou, and K. Zhou. M2m communications for smart city: An event-based architecture. In *IEEE CIT*, 2012.

[18] E. Wilde. Putting things to rest. Technical Report 2007-015, UC Berkeley, nov 2007.

[19] Zigbee alliance. http://www.zigbee.org/.