Experimental Evaluation of Improved IoT middleware for Flexible Performance and Efficient Connectivity

Abstract We previously proposed an Internet of Things (IoT) middleware called Middleware for Cooperative Interaction of Things (MinT). MinT supports thread pooling to quickly process requests from other IoT devices. However, using a thread pool with a fixed number of threads equal the number of CPU's cores can cause waste of memory, CPU's resource and performance of IoT device. In this paper, we propose an enhanced approach called Improved MinT (MinT-I) to improve the performace of MinT middleware efficiently by real-time adjustment of threads. In particular, we aim to improve the performance of connection part, which is responsible for analyzing, processing, and re-transmitting the received packets. The experimental result show that the MinT-I increases the average throughput by approximately 25~35% compared to the existing middlewares. Finally, proposed MinT-I not only can optimize the memory and resource usage, but also can reduce the latency and power consumption of IoT deivces.

Keywords: Internet of Things (IoT), MinT, Performance, Multi-Threading

1. Introduction

The Internet of Things (IoT) provides information services for everyday use and is built upon interconnections between sensor devices and network platforms. The IoT operates by receiving and integrating information from many interconnected devices to provide information-related services. Recently, various IoT middlewares have been researched and developed. Any IoT middleware needs to satisfy several key system requirements, including guaranteeing stable network connections and efficiently managing the system resources [1,2].

IoT devices collect peripheral information and provide services to users or other IoT devices. This role is similar to that of a sensor node in a wireless sensor network (WSN). The smallest systems include sensing devices that activate simple tasks at regular intervals, such as sensing, gathering, and transmitting to the sink node or central server [3,4].

From their early role as simple sensing and transmitting devices, IoT devices have evolved to perform sensing, gathering, processing, and service roles. Therefore, periodic requests between IoT devices arise, and such requests may be concentrated in only one device. However, because IoT devices process the received requests late, and message transmission is delayed sequentially, responses to requests cannot be sent on time. The transmission delays impact all connected IoT devices, and the entire chain of information exchange can be severely delayed. This can affect the reliability of information exchange in IoT networks.

To address this problem, we previously proposed a platform called Middleware for Interconnection of Things (MinT) [5]. It allocates many CPU resources to manage communication, rapidly processing requests received from the other IoT devices. Like most middleware, it supports the non-block socket for communicating with other IoT devices, preferring asynchronous communication, which uses thread pooling to prevent the creation of consecutive threads. The thread pool in MinT can operate multiple threads to handle requests quickly. To reduce the complexity and overhead of the middleware system, the number of threads equals the number of cores.

Thread pooling offers the advantage that the number of threads are fixed to be equal to the number of cores or are increased by the number of requests. The fixed number of threads has been demonstrated to increase the efficiency of the CPU; however, this method can cause unnecessary waste of resources. For example, when the IoT device receives a number of requests that is smaller than its throughput, thread pooling does not offer any advantage and also wastes them. In addition, creating the threads increases the IoT device’s memory usage because multi-threading generally allocates an independent stack for each task.

The virtual threading method has been proposed to address this problem, reducing memory usage by sharing stacks. However, this method can cause execution overhead because it requires extra work for memory sharing. Therefore, a new method is required that efficiently operates memory and execution resources by directly adjusting the number of threads to promptly process requests from the other devices.

In this paper, we propose Improved MinT (MinT-I) that extends new approach called variable thread adjustment (VTA) to the MinT. This optimizes thread pool management by controlling the number of threads in the handler thread pool (HTP) according to number of received packets. In particular, we aim to enhance the performance of the interaction thread pool group (ITPG) that is responsible for analyzing, processing, and retransmitting to quickly process the requests. Applying VTA to the ITPG adjusts the number of threads and throughput according to the number of packets, the capacity of the queue and retransmission time of each packet. By physically adjusting the number of threads, VTA can reduce the overhead and complexity of the structure caused by virtualization and optimize resource and memory management issues caused by the use of threads. In addition, VTA can process requests quickly and reduce the transmission delays of the network environment compared to existing approaches that use fixed thread pooling.

We demonstrate the effectiveness of the proposed method through experiments and analysis centered on the ITPG. For the performance evaluation, the proposed approach is compared with existing IoT middlewares in 3 types of server test beds, based on the following metrics: total throughput, throughput per packet block size, and throughput per hardware platforms. In the experiments, the proposed approach showed noticeably improved performance with regard to the processing of request messages and system throughput.

The rest of this paper is organized as follows. Section 2 describes all of the work related to thread management methods of currently existing IoT middlewares. Section 3 presents the problem of the existing MinT middleware. Section 4 describes the improved approach for optimizing thread pool management. In Section 5, the performance characteristics of the new method are measured and evaluated, and Section 6 concludes the paper.

2. Related Work

Many open source IoT middlewares have been studied in the pursuit of developing and connecting IoT devices. These middlewares have been integrated in the management of various sensing devices through collaborations with manufacturers. The middlewares have also ensured

[6] and message queue telemetry transport (MQTT) [7]. Both are IoT application protocols that can be used in both cloud and distributed service environments. They can enable efficient resource exchange between both high-performance IoT devices and low-power constrained IoT devices.

IoT devices share their information through flexible connectivity with other IoT devices, and therefore, the IoT middlewares need to be able to connect and operate efficiently with low-end to high-end hardware platforms, including those that support a multi-core processor. In particular, if the IoT device fails to process requests within the specified time, message transmission is delayed, which can affect the reliability of information exchange in IoT networks.

Existing constrained devices have included event-based platforms to expedite the processing of requests in WSN. Typical event-based platfoms are TinyOS [8], Contiki [9], and OpenIoT [10]. These support the parallel execution model to operate sensing devices and connections in parallel. However, these execute tasks following the cyclic executive pattern and are therefore not fit for environments in which variable tasks and requests are operated simultaneously.

TOSThread extends the multi-threading middleware model to the TinyOS. Because TOSThread allocates new stacks in the memory to create new threads, the platform causes memory management problems in the constrained devices [11]. UnstackedC extends the TinyOS multi-threading model so that new threads do not require additional stacks, reducing memory usage [12]. However, if a thread that is running in the process preempts the shared memory, another thread does not have the opportunity to preempt the same shared memory. In addition, if the thread must wait, excess overhead can occur during process execution because each of the thread's local variables must be managed by a separate data structure.

ProtoThread is a multi-threading model that can be extended to Contiki. This platform allows tasks driven by events to be configured as multiple virtual threads, which can reduce memory usage because the stacks do not have to be allocated by thread. However, this approach operates the virtual threads in a busy-waiting model and therefore cannot temporarily control the operation of the CPU. In addition, this platform cannot be applied to the multi-threading environment because it is basically a virtualization of the single-threading model [13].

Recently, because of advances in hardware technology, high-performance and small hardware platforms have been studied and developed. These support the multi-threading model for use on high-performance hardware platforms. In particular, in the IoT network area, these devices can cause a phenomenon in which requests are concentrated on just one device. Therefore, IoT devices should be able to respond quickly to this concurrent connectivity.

Toward this objective, IoTivity is an open source IoT middleware that has been adopted as the standard by the Open Interconnect Consortium organized by Samsung Electronics, Intel, Cisco, and GE among others. IoTivity consists of the Discovery module, which is responsible for finding local and/or remote devices; the Data Transmission module for exchanging and controlling the information; the Device Management module for managing device configurations and privileges; and the Data Management module, which implements data collection and analysis. In particular, IoTivity supports thread pooling to process the requests simultaneously. Because IoTivity works on the CoAP, the middleware can easily support devices with low specifications and low power [14].

Californium is an open source IoT middleware based on the CoAP developed using Java. It is composed of the Network State module, which supports connections through various network protocols; the Protocol Stage module that processes the request messages; and the Business Logic state that provides services to users. In addition, because Californium simplifies the Protocol Stage and introduces multi-threading, IoT systems operating on this can provide quick responses to multiple requests within a short timeframe [15].

An open source IoT communication middleware based on the CoAP is nCoAP, which uses a netty framework to support non-block asynchronous communication. Within this framework, nCoAP uses the multi-threading model and thread pooling to process requests simultaneously and quickly [16]. Android Things is an IoT middleware developed by Google, based on the kernel and upper subsystems of the Android operating system. Android Things is aimed at cross-platform applications for IoT devices with low power and minimal hardware specifications; it supports the cloud system based on machine-to-machine communication methods such as Bluetooth Low Energy and Wi-Fi [17].

Tizen is an open and flexible operating system built from scratch to address the needs of all stakeholders of the mobile and connected device ecosystem, including device manufacturers, mobile operators, and application developers. Tizen receives requests from other devices through the sensor manager; because the sensor manager creates a worker-thread for each requested event, the number of threads continuously increases with the number of requested events, which not only incurs CPU overhead and memory, but also stops the process [18].

Because many IoT devices are interconnected with each other in the IoT environment, they can share available resources to satisfy their service-related needs. The above studies provide an efficient system architecture for delivering scalable IoT services. In particular, the communication component, which is connected to other devices and processes requests in most of the platforms (excluding Tizen), uses thread pooling based on multi-threading.

thread pool to be equal to the number of CPU cores to maintain the performance appropriate for the power of the hardware platform. However, approaches that fix the thread pool size equal to the number of cores can reduce the energy efficiency of the IoT device because these methods can use unnecessary resources and memory, and can also cause processing delays because they cannot flexibly respond to incoming requests.

In this paper, we propose new approach for optimizing thread pool management to control the number of threads in the HTP according to the number of received packets. By physically adjusting the thread count, the proposed method can reduce the overhead and complexity of the structure caused by virtualization. In addition, the proposed method can improve the memory efficiency and performance of the IoT device compared to existing approaches that use fixed thread pooling.

3. MinT Overview and Challenge

In this paper, we propose a new approach to improve the performance of thread pooling in the MinT. Therefore, in this section, we discuss overview of existing MinT middleware and point out the drawbacks of thread pooling which are fixed to be equal to the number of cores.

MinT aimed at supporting the cooperative ineteraction of IoT devices. It can operate IoT devices in a fully distributed IoT network, and adapts to IoT environments by recognizing the requirements of IoT middleware. To satisfy the requirements of IoT middleware, the goals of MinT have been as follows [5]:

- Support the hardware abstraction layer to allow development of various sensors and network devices
- Provide a network environment to support active interaction between IoT devices
- Improve processing performance and network reliability
- Allow energy-efficient sensor device and data management
- Enable easy installation, and provide a simple high-level API

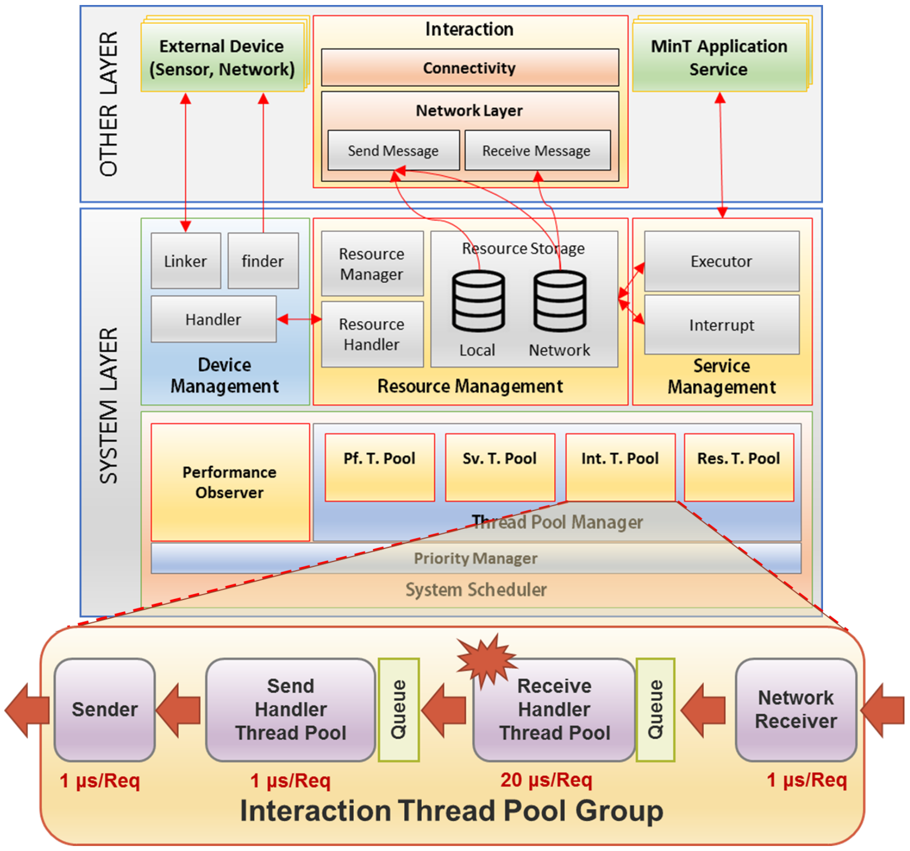
Especially, MinT provides a system process model to improve processing performance and network reliability. IoT networks can be configured with numerous devices, ranging in size from local area to global area networks. In distributed networks, rapid request processing is one of the most important factors in IoT environments. Because processing delays can cause service and network latency across the entire network, IoT devices require efficient and rapid processing within the middleware. To improve processing performance, MinT prioritizes system resource utilization towards the Interaction Layer; this layer manages and processes requests received from other IoT devices. MinT also supports multi-threading with thread pooling in the System Layer, to allow efficient processing of requests according to the hardware platform performance.

3.1 Structure of System Layer

Figure 1 shows the structure of the System layer in MinT. For the purpose of implementing efficient system operations, the System Layer's functionality is implemented as multiple modules – Device Management, Resource Management, Service Management, and System Scheduler.

The Device Management module can find/link to the sensing and networking related device drivers connected to MinT by employing finder and linker sub-modules. The Handler sub-module manages the data and operations for all the linked devices. The data handling is accomplished by working with the Resource handler sub-module of the Resource Management module.

The Service Management module is primarily utilized during the development of application services. The APIs built into Mint enable the rapid building of IoT applications with functionality such as control of deployed sensors, resource searching, resource release, and information requests.



The Resource Management module is responsible for managing all the local and network resources in the MinT. The resources can be saved into local storage or network storage. Local storage saves the resources of MinT devices while network storage saves the resources received from external IoT devices via the Interaction Layer. In addition, in order to effectively handle and manage repeated requests, the Resource Management module provides network cache storage. The module directly connects to the Handler sub-module of the Device Management module. The Resource management module plays a key role in both saving system resources and in providing a sophisticated interface to the sensing devices.

The System Scheduler is an integrated thread management tool, which handles threads created from the System Layer, Interaction Layer, and Application layer. As shown in figure 1, the System Scheduler is composed of the Thread Pool Manager, Priority Manager, and Performance Observer. The Thread Pool Manager manages four distinct types of thread pools:

- Application Service: This type of thread pool is used to support the complete group of services used in MinT and in the Service Management module of the System Layer. Given the fact that the number of threads created in each application is different and that all services cannot work simultaneously, the maximum size of each thread pool is adjusted according to the number of services currently running.

- Resource Management: This type supports resource handling for the resources collected from the external sensing devices. The maximum size of the thread pool is assigned to be equal to the number of CPU cores. The size of the associated waiting queue is set to the value 100 because the volume of I/O is low.

- Interaction: It is used in the Interaction Layer and is the thread pool management group for the Protocol Adapter and Send/Receive Handler. The maximum size of the thread pool is setup to be equal to the number of CPU cores. This is the most important group among all the groups in MinT. The size of the waiting queue is set to be between 1 KB and 10 MB according to the memory size of the hardware platform.

- Performance Profile: It manages and measures the performance of each thread pool group. The Performance Observer in the System Layer operates in conjunction with this thread pool. A separate thread is created for each thread pool in order to measure its performance.

3.2 Performance Challenge Due to Specific Thread Pools

The MinT's Interaction Layer receives a message requested from other IoT devices. The Network Layer in the Interaction Layer activates the Matcher & Serialization, Transportation, and System Handler, which are included in thread pools to process the received message. As shown in Figure 1, MinT has two modules and two important thread pools for receiving, analyzing and responding to the message. Each module and thread pool are managed in the ITPG. First, a message received from the Network Receiver (NR) is added to the HTP Queue. Second, the HTP analyzes the requirements of messages that are retrieved one by one from the queue. Based on the analyzed message, the MinT starts new jobs to operate sensing devices and sends the resources stored in the Resource Manager according to the requirements analysis. Third, after executing the requirements, the message is added to the Send Thread Pool Queue to prepare to send the response message. Finally, the Send Thread Pool makes a packet with the message retrieved from the queue to adjust the CoAP and sends the packet to the final destination using the Platform Adapter Layer.

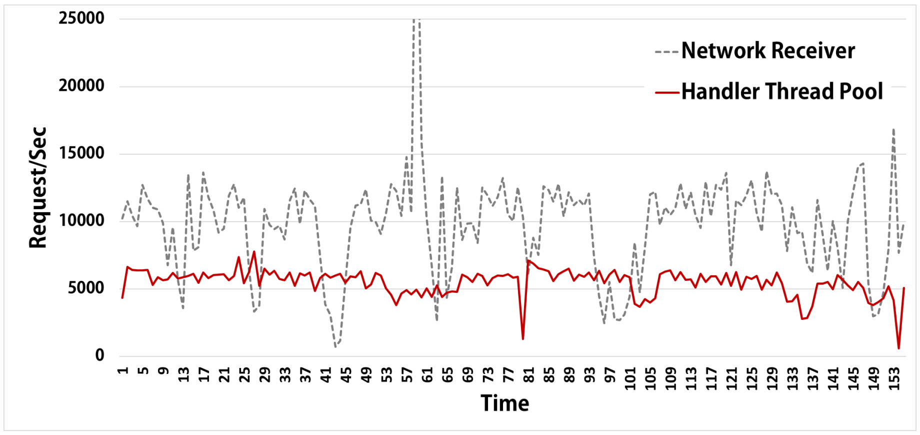


Figure 2. Throughput of NR and HTP on Single Thread.

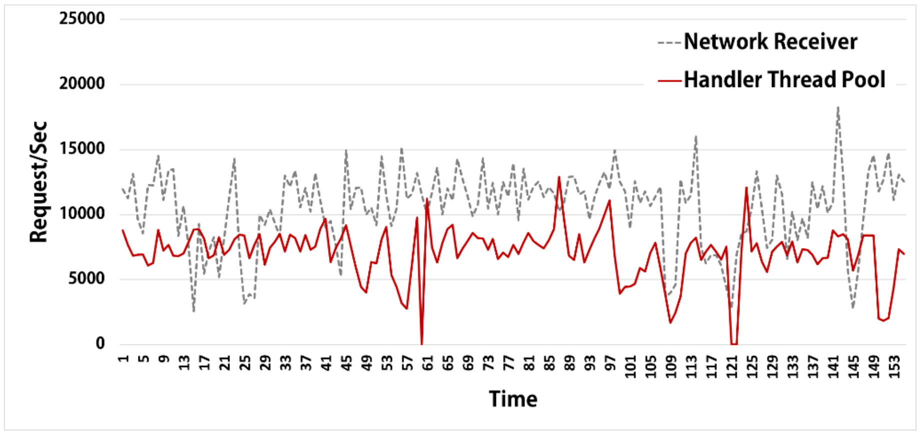


Figure 3. Throughput of NR and HTP on 2 Threads.

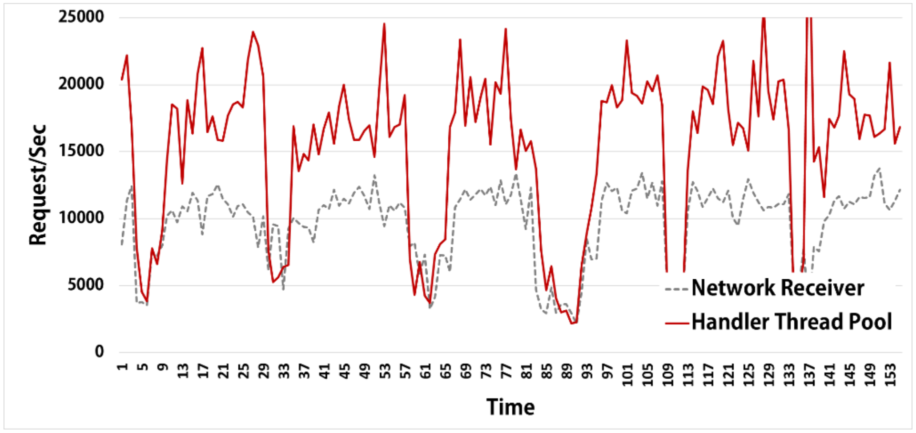


Figure 4. Throughput of NR and HTP on 4 Threads.

The Thread Pool Manager in the System Scheduler controls all thread pools operated in the Interaction Layer. The Thread Pool Manager allocates the highest priority and the largest number of threads to the ITPG that has the most tasks. In other existing methods, the maximum size of the thread pool is set to be equal to the number of CPU cores of the hardware platform. In MinT, we fixed the size of the thread pool to be equal to the number of CPU cores to maintain the performance that is appropriate to the power of the hardware platform.

In this paper, we use the server-client model, processing request messages on the Raspberry Pi 3 with MinT, to analyze the comparative message throughput of various thread environments. Figures 2, 3, and 4 show the throughput of the HTP and NR by the size of the thread pool. In this paper, we decided that fixing the size of the thread pool according to the number of CPU cores could improve the performance and energy consumption of the middleware. However, as shown in Figure 2, in a single-core hardware platform that operates a single thread represents, the throughput of the HTP, which processes the received message, is lower than the throughput of the NR, which receives all messages from the other IoT devices. The HTP handles most tasks requested by received messages. Therefore, as shown in Figure 2, the HTP with a single thread not only causes processing delays, but also adversely affects the reliability of the network environment. Figure 4 shows that the throughput of the HTP in the multi-core environment, with a thread pool of four, is clearly higher than that of the NR, in contrast to the single thread hardware platform shown in Figure 2. The comparison of these figures shows that multi-threading can accommodate many requests but can also consume unnecessary CPU resources, even when the number of received messages is small.

We expected that the existing thread pool management strategy would be an efficient method to operate the IoT device. The experimental result showed that the performance of the proposed method is more efficient than existing middlewares. However, from the above result, degradation, waste of resources, and energy consumption. Therefore, in this paper, we propose an improved approach for optimizing thread pool management to resolve the associated issues.

4. Improved Approach for Optimizing Thread Pool Management

In Section 3.2 we discussed multi-threading in the Thread Pool Manager for quickly processing many messages requested from other IoT devices. However, existing threading methods in which the thread pools are based on the number of CPU cores incur performance degradation, waste of resources, and energy consumption. In this paper, we propose an improved approach called MinT-I for optimizing thread pool management to solve the above problems.

The MinT analyzes the performances of modules that are operated by the Network Layer or the System Layer using the Performance Observer in the System Layer. This feature measures the performances every few seconds and stores the performances within 10 seconds. In this paper, we aim to improve the performance of the ITPG that processes the messages requested from other IoT devices, as shown in Figure 1. Toward this objective, we propose the variable thread-pool adjustment method called VTA.

The VTA is organized as follows. First, the management method analyzes the performance of the hardware platform and predicts (1) preliminary number of threads based on the throughput of requests received from the other IoT devices. Second, (2) additional preliminary number of threads is calculated by the queue size and the Retransmission Timeout (RTO) waiting time to ensure efficient memory management; the maximum value of both two calculated preliminary values is applied as the next number of threads. Third, the maximum number of threads is continually calculated while measuring the performance of the thread pool. Finally, the final number of the thread pool is calculated as the maximum value of both (1) and (2) preliminary values within the maximum number of threads at the next time t.

Table 1 represent all formula notations used in the sections below.

Table 1. Formula Notations.

Symbol	Meaning
t	Current time
H_t	Throughput of the Handler
R_t	Throughput of the Network Receiver
$C_{N,t}$	Current number of threads
S_t	Single thread throughput of Handler
n	Time period for data average
QS_t	Current number of packets in the queue
e_t	Amount of packets expected be received at the next time
$W_{min,RTO}$	Minimum RTO waiting time

4.1 Preliminary Number of Threads from Request Throughput

In this section, we calculate the preliminary number of threads that is required to process the requests by comparing the throughput of the NR and the HTP as shown in Figure 2-4. We aim to calculate the remaining number of threads which is required to satisfy Request (R_t, Request/Sec), which is the throughput of the NR. First, we calculate the single thread throughput (S_t, Request/Sec) of Handler (H_t, Request/Sec), which is the throughput of the HTP as shown in Equation (1).

Second, we calculate Sa_t, which is the average throughput of S_t from the current time to n seconds earlier, as shown in Equation (2). The n value takes the average value of the data from the current time to 5 seconds earlier. The reason for using 5 seconds (1-s intervals × 5 intervals) is that the n value is then close to the size of the HTP, and the data fluctuation range of data can thus be minimized.

Third, the equation for $PR_{N,t}$, which is the preliminary number of HTP threads, is easily determined using the difference ($R_t - H_t$) and average throughput of single thread (Sa_t) as shown in Equation (3).

$$S_t = \frac{H_t}{C_{N,t}} \tag{1}$$

$$Sa_t = \frac{1}{n}\sum_{i=0}^{n} S_{t-i} \tag{2}$$

$$PR_{N,t} = \frac{R_t - H_t}{Sa_t} \tag{3}$$

4.2 Preliminary Number of Threads from Queue Size and RTO

In this section, we establish a second preliminary number of threads that is calculated by the queue size of the HTP and the RTO to ensure quick processing of requests encountered from other IoT devices. The queue size of the HTP is set at approximately 1–2% of the memory size of the hardware platform. In the case of the Raspberry Pi 3, the queue size is up to 10 MB, which is dynamically designated in the heap memory. Packets received by the CoAP, which predominates in the IoT protocols, range from 4 bytes to 20 bytes. Therefore, the maximum queue size of the HTP can be up to 500,000 packets in the Raspberry Pi 3. High-capacity IoT devices such as Raspberry Pi 3 can reduce the packet loss rate because these devices can handle many packets received from many other IoT devices.

Low throughput can lead to an increase in latency with out packet capacity. For example, in the CoAP protocol that uses CONs messages, if the client does not receive an ACK for its CON within an initial back-off time, the CoAP retransmits the same CON again. By default, the initial back-off is set to a random time between 2 s and 3 s. If a reply to a CON packet is not received within the initial back-off time, the CoAP will double the initial back-off time and retransmit the same packet up to four times. This process can take up to approximately 93 seconds, referred to as MAX_RETRANSMIT [6]. Increasing the waiting time affects the overall response time of the network environment. To achieve fast response times, we must reduce the retransmission rate (RTO rate) of the packet on the client side by increasing the reliability. One approach to increasing reliability is to increase the processing ability of the IoT device on the server-side.

This paper proposes Equation (4) using the queue size and RTO in order to ensure expedited request processing. To calculate Equation (4), we predict the number of packets in the queue at the next time (1 second later). First, the current number of packets (QS_t, number of requests) in the queue is measured. Second, expected demand, which is the amount of packets expected be received at the next time (e_t, number of requests), is calculated using historical data. The e_t uses the average value of the data from the current time to 5 seconds earlier. As with Sa_t in the Equation (2), the reason for using 5 seconds (1-s intervals × 5 intervals) is that the e_t is then close to the queue size and the fluctuation range of data can be minimized. Third, the preliminary number of threads from queue size and RTO ($QR_{N,t}$) is calculated from the ratio of the predicted number of packets in the queue to H_t times the minimum RTO waiting time ($W_{min.RTO}$). The RTO is calculated by the application protocol that is used in the middleware's network protocol. For example, CoAP sets the minimum RTO to 2 s and uses exponential back-off for message retransmission.

Finally, the preliminary number of threads of HTP for the next time ($P_{N,t+1}$) is obtained from Equation (5).

$$QR_{N,t} = \frac{QS_t(req) + e_t(req)}{H_t(req/s) * W_{min,RTO}(s)} \tag{4}$$

$$P_{N,t+1} = MAX[PR_{N,t}, QR_{N,t}] + C_{N,t} \tag{5}$$

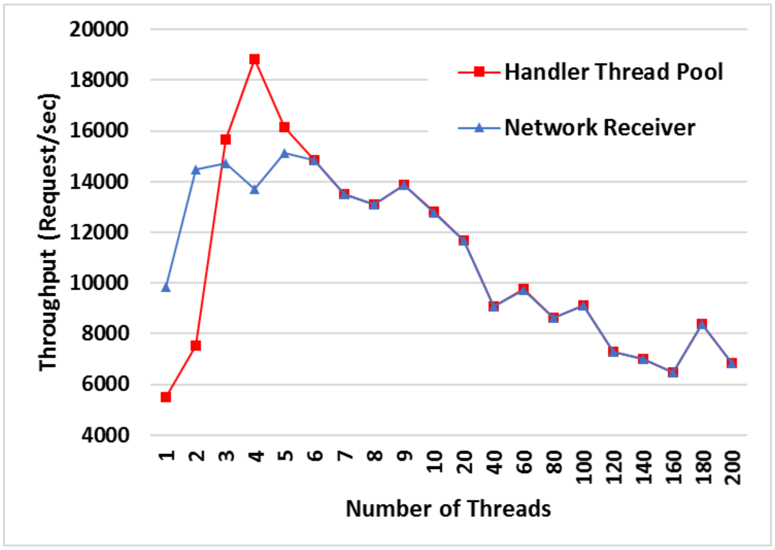


Figure 5. Throughput of NR and HTP by Number of Threads.

4.3 Determining the Number of Threads in the HTP

In Section 4.2, we consider the HTP's throughput and queue information to improve the IoT device's performance. Equation (5) adjusts the number of threads in the HTP to promote high performance in the server platform. This section addresses the issue of multi-thread processing, which is one approach to handling tasks efficiently, but also suffers from inefficient management, incurring performance degradation, excess memory consumption, deadlocks, and so on. In particular, having too many threads can seriously degrade performance in two ways. First, partitioning a fixed amount of work among too many threads gives each thread too little work, and the overhead associated with starting and terminating threads far exceeds the useful work. Second, having too many concurrent threads incurs additional overhead from the sharing of fixed hardware resources. To confirm the performance of the HTP, Figure 5 shows the throughput of the NR and HTP by number of threads. This experiment was conducted in the same experimental environment as that shown in Figures 2, 3, and 4, with the number of the highest performance on the Raspberry Pi 3, and then the throughput decreases with further increased numbers of threads.

In this section, we calculate the maximum value of the number of threads at the next t ($NT_{max,t+1}$) to maintain the highest performance of the HTP by continuously assessing it. Equation (6) measures the variation of H_t and calculates the maximum number of threads. If H_t of the HTP increases, the maximum number of threads in the pool increases and is accepted at the next time t. However, if H_t does not increase further, the maximum number of threads is fixed. In Equation (6), $H_{d,t}$ denotes the difference between H_t and H_{t-1}, and $H_{max,t}$ is the maximum H_t.

Finally, the number of threads at the next time t (N_{t+1}) is obtained from Equation (7).

$$NT_{max,t+1} = \begin{cases} NT_{max,t} + 1, & H_{d,t} < 0 \text{ and } H_t > H_{max,t} \\ NT_{max,t}, & otherwise \end{cases} \tag{6}$$

$$N_{t+1} = \begin{cases} 1, & P_{N,t+1} < 1 \\ NT_{max,t+1}, & P_{N,t+1} > NT_{max,t+1} \\ P_{N,t+1}, & otherwise \end{cases} \tag{7}$$

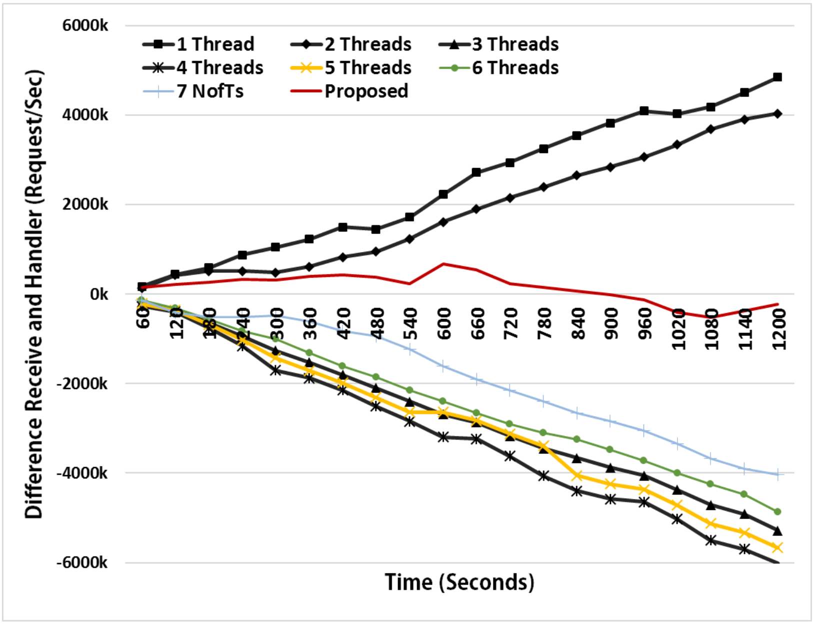


Figure 6. Cumulative Differences in Throughput between NR and HTP.

4.4 Experimental Verification of Optimized Thread Pool Management

In Section 3.2, we discussed fixing the size of the thread pool to be equal to the number of platform. However, this approach can cause other problems in managing the System Scheduler resource, and these problems affect the performance of the platform. In this paper, we propose a MinT-I to optimizing the thread pool management to control the number of threads in the HTP according to the number of received packets. This improved method is intended to be able to control the number of threads and process packets to be similar to the number of received messages. Therefore, we tested the improved method's capability in this respect by conducting basic experiments.

In the experiment, we sent 10,000–15,000 packets/s to the MinT-Server on the Raspberry Pi 3 for 20 minutes, imposing a significant stress on the IoT device to ascertain the behavior of the platform in an extreme situation. Figure 6 shows the cumulative difference of throughput between the NR and HTP. Positive values indicate that the throughput of the NR, which is the number of requests from other devices, is higher than that of the HTP, and negative values indicate that the throughput of the HTP is higher. Therefore, when the HTP has 1 or 2 threads, the results show that the value increases in the positive direction because the throughput of the HTP is less than that of the NR. However, when the HTP has 3–7 threads, the difference value moves further in the negative direction. Over time, the differences in either direction between the number of received packets and the throughput of handled received packets can cause transmission delays, excess energy consumption, and resource management problems. In turn, these problems delay the response time and reduce reliability in the network environment.

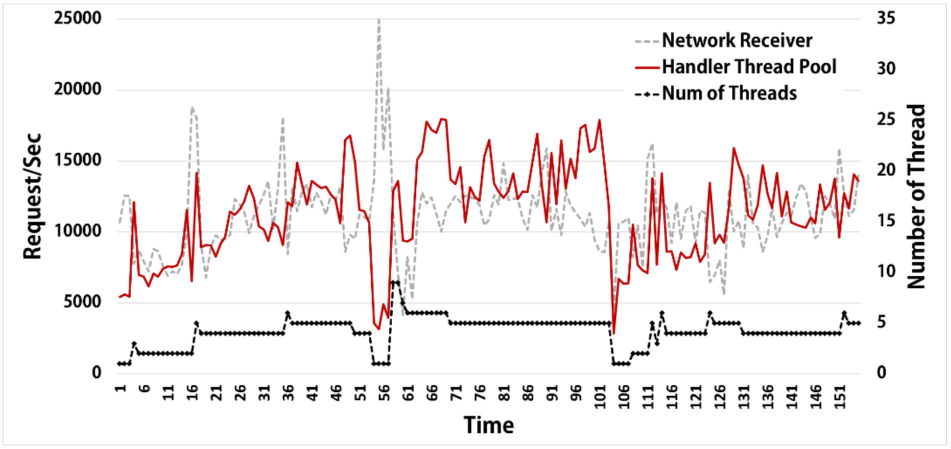


Figure 7. Throughputs and number of threads using the proposed method to improve thread pool performance.

Figure 7 shows the throughput and number of threads using the proposed method to improve the performance of the thread pool. As shown in the figure, because proposed method adjusts the number of HTP threads according to the NR's throughput, the throughput of the HTP is similar to the throughput of the NR. However, there may be some dissimilarties in some areas. MinT-I determines number of threads at the next time t according to the amount of requests and queue usage at the current time t. The throughput of HTP at the next step is determined according to number of thread.

For example, Figure 7 shows that the throughput of the NR and HTP increase and decrease sharply as shown in 54~58 seconds of time axis. Since the throughput of NR decreased from 46 to 54 seconds, MinT-I reduces not only the number of threads but also the throughput of the HTP. Reduced throughput of HTP cannot process all requests received from NR. The remaining requests are stored in the queue. Since the part of 59 seconds, MinT-I increases the number of threads and the throughput of HTP because it must process the all requests stored in the queue. Therefore, it shows that the throughput of HTP is higher than NR as shown in 59~75 seconds. This can be seen as a natural result of the proposed approach. We also claim that overall trends of NR and HTP are similar expect for the above cases.

In addition, Figure 6 shows that the proposed method continues to maintain a difference near zero, with the throughputs of the NR and HTP remaining similar over time. Thus, we demonstrate that VTA can reduce unnecessary memory and resource consumption because this method can adjust the number of threads according to number of requests. In addition, VTA can respond to the requests faster than existing methods that fix the thread pool size in the ITPG. Therefore, the proposed method efficiently improves the platform's performance, achieving low transmission delays and enhanced battery life in an IoT environment.

5. Performance Evaluation and Results

While information and requests may be flowing smoothly through large sections of an IoT network, it is possible that at a given point in time a very small number of IoT devices are suffering a degradation in performance due to network bottlenecks. This can result in a domino effect with the performance of other connected IoT devices. To address this issue, the middleware should enable quick and efficient processing for those IoT devices saturated with requests. Performance enhancement introduced by middleware can make a significant difference in maintaining overall IoT system-level performance.

The primary function of an IoT middleware is to provide efficient message processing and systems management capabilities for the IoT environment. In this paper, we have proposed a VTA called MinT-I to improve information processing performance. In particular, to enable rapid responses to requests from other IoT devices, we aim to improve the performance of the ITPG.

which is responsible for analyzing, processing, and retransmitting requests. Section 3 described the problem of system-level performance degradation due to a fixed thread pool size. In addition, in Section 4, we described the VTA and demonstrated its efficiency through experiments. In this section, we experimentally evaluate the processing performance of MinT-I, using throughput and scalability as the performance metrics for comparison.

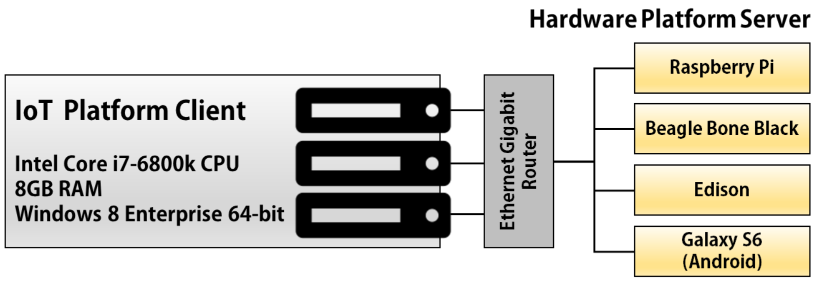


Figure 8. Test Environment of Three IoT Platform Clients and Four Hardware Platform Servers.

5.1 Experimental Environment

Basically, we uses CoAP protocol for experimentation and evaluation. Using the CoAP protocol, IoT devices, which are high-end or low-power constrained hardware platform, can share their resources efficiently each other. Therefore, most of IoT middlewares support the CoAP protocol and use the main application protocol of them. In the basic test scenario, all of the IoT middlewares can sent verifiable response data ("hello world") to simple GET requests (/request), and each of the request and response data records was assigned a CoAP header. In addition, we used the yardstick program structure, consisting of a load generator and virtual clients, in the Internet protocol to achieve accurate performance measurements [19].

Figure 8 shows the experimental setup, with three PCs for the virtual clients and four servers for the platform performance testing. The specifications of the client PCs were: Intel Core i-7-6800K CPU, 8 GB RAM, and Windows 8 Enterprise 64-bit. The hardware platform servers tested were Raspberry Pi 3 (wired), BeagleBone Black (wired), Intel Edison (wireless), and Galaxy S6 (wireless). These server platforms enable us to test the adaptability and scalability characteristics of the MinT compared to other IoT platforms. To connect between clients and servers, we used a gigabit Ethernet router with either wireless or wired connections to the server platforms.

With this test setup based on three client PCs, the load generator can vary the number of virtual clients from 1 to 10,000. Each client sends 10 packets/s to the tested server for 60 s. The performance measurement module executes three iterations for each client transmission. If the setting for the number of clients is changed, the platform is given 60 seconds to cool down before continuing with the next step. Table 2 shows more detailed specifications for the experimental environment.

Table 2. Specifications for Clients and Servers.

	Platform	CPU	RAM	Con.
Client	PC	Intel Core i7-6800k @ 3.6GHz quad-core	8GB	Wired
Testbed	PC	Intel Core i7-6800k @ 3.6GHz quad-core	8GB	Wired
Testbed	Raspberry Pi 3	ARM Cortex A53 @ 1.2GHz quad-core	1GB	Wired, Wireless
Testbed	Beagle Bone Black	ARM Cortex A8 @ 200MHz dual-core	512MB	Wired, Wireless
Testbed	Intel Edison	Atom @ 500MHz dual-core	1GB	Wireless
Testbed	Galaxy S6 (Android)	ARM Cortex A57 @ 2.1GHz octa-core	3GB	Wireless

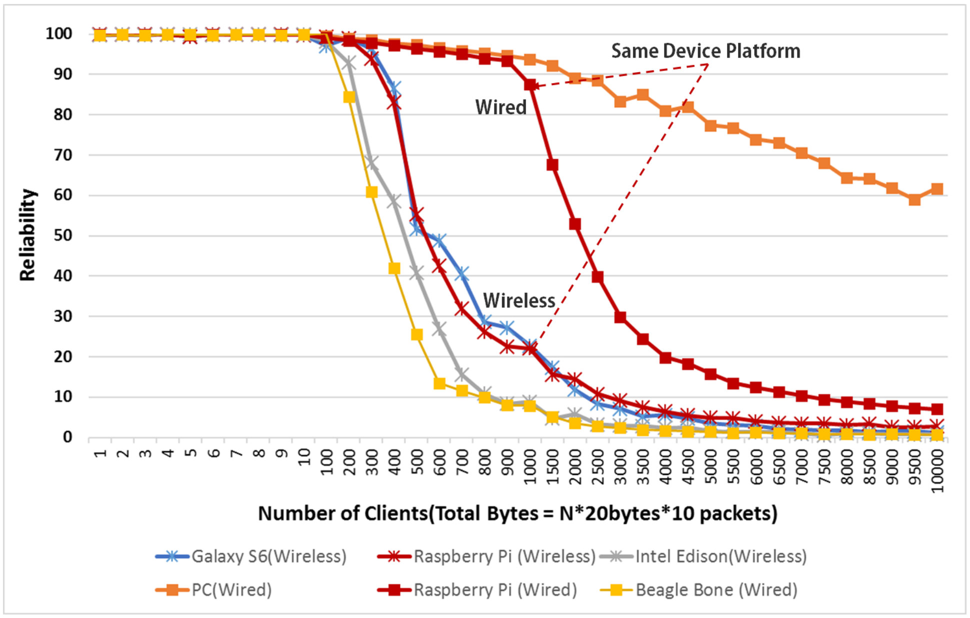


Figure 9. Reliability of Hardware Platforms in Wired and Wireless Networks.

5.2 Reliability of Wired and Wireless Environments

We used the Raspberry Pi 3 and the BeagleBone Black testbeds in the wired environment for the experiments, whereas other testbeds without LAN ports had to be tested in the wireless environment. Because the proposed platforms transfer packets on the CoAP protocol with non-confirmable messages, both wired and wireless transmission reliability is further decreased. Therefore, to conduct accurate tests with reliable network environments, we had to test and confirm the reliable accuracy of the experimental environments.

Wireless reliability tests between server and client in both unicast and multicast connectivity with CoAP were presented in [20]. These researchers gradually increased the wireless interference in the network in increments of 5 MB/s, and measured the reliability of receiving responses to the respective requests, with the number of clients varying between 1 and 30. The results showed that the reliability of Unicast with a confirmable message was approximately 98–100%, and the reliability of the Multicast with non-confirmable messages was approximately 80%. These experiments were tested with up to 30 clients requesting only one packet/1.7–5 s. However, in our experiment, we tested the server with up to 10,000 clients and each client requests 10 packets/s. Therefore, the reliability found in these previous Unicast and Multicast experiments may not be sufficiently conservative.

In this paper, we measured the reliability of wired hardware platforms (PC, Raspberry Pi 3, BeagleBone Black) and wireless hardware platforms (Galaxy S6, Raspberry Pi, Intel Edison), as shown in Figure 9. The figure shows that the cause of reliability decrease is network interference with wireless systems or lack of server capability. Increased network traffic causes increased packet loss, and the server delay caused by insufficient processing capability can affect the response time to clients. To verify the reliability, we measured the performance of each platform within a time limit of 60 s. In addition, we also included the impact on reliability that clients could not receive responses from the server because of lack of server capability.

As shown in Figure 9, the PC platform, which is in the wired environment, shows high reliability of up to 80%. In this result, we could not know if the cause of the PC platform's reliability is the lack of server capability or a problem with the network reliability. However, we know that the reliability problem in the wired environment of the Raspberry Pi 3 and BeagleBone Black is the lack of server capability because the reliabilities of these devices are lower than that of the PC platform, indicating that the reliability problem in the PC platform is not a network problem as shown in the figure. Therefore, we could measure the throughput of Raspberry Pi 3 and BeagleBone Black platforms in reliable network environments.

However, the wireless Raspberry Pi 3 platform, which has the same specifications as the wired Raspberry Pi 3, shows that the reliability is further decreased compared to the wired platform. From this result, we cannot accurately measure the throughput of the Raspberry Pi 3 and Galaxy S6 in the wireless environment, which is often influenced by network interference. Therefore, we concluded that the wired Raspberry Pi 3 is the most accurate among the tested platforms for the subsequent experimental measurements.

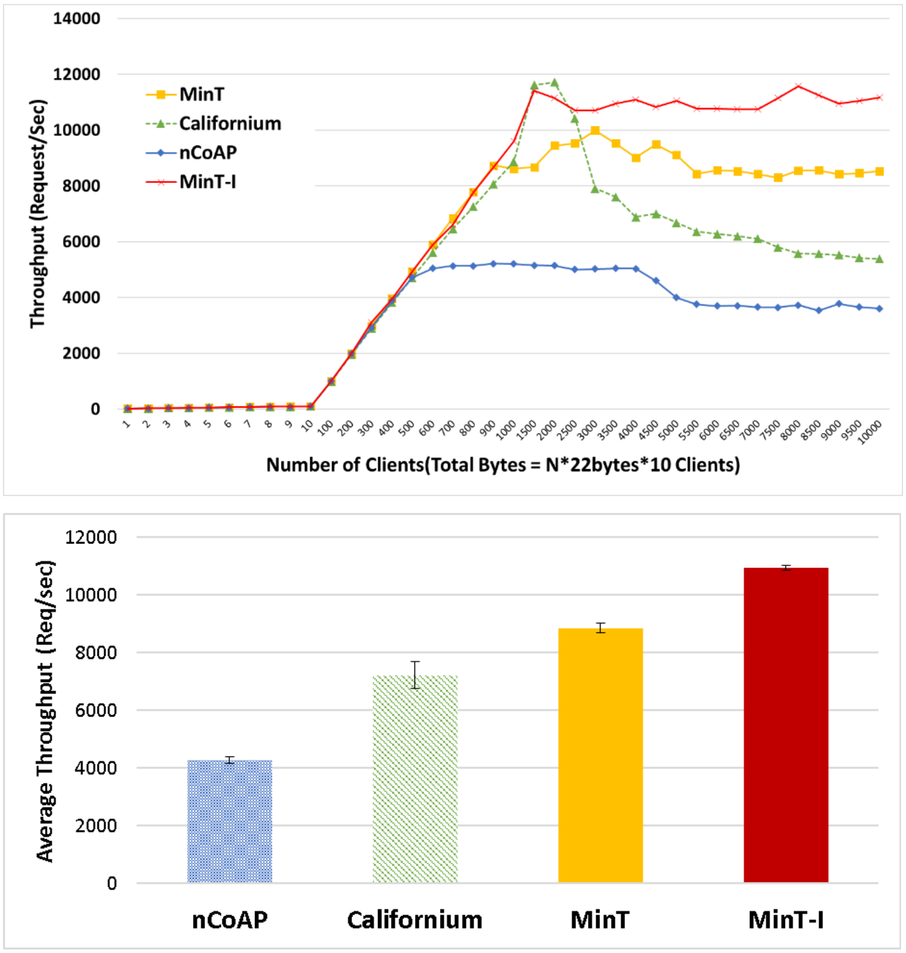


Figure 10. Throughputs with MinT, Californium, nCoAP, and MinT-I (SE).

5.3 Performance in IoT Middlewares

For IoT systems that are small scale, it is fairly easy to ensure that the design ensures low average response time. However, as the scale of IoT systems increases, the average response time tends to increase. Beyond a certain point, this increase starts to have a negative impact upon the operation speed and overall system performance. As the processing delays for IoT services increase, the response time begins to increase in tandem and it will be problem of realibility of large-scale IoT networks. Recently, there are number of platforms in IoT environment to assist efficient information exchange. They help that the hardware platform operates efficiently and assure fast response time between server-client. In this paper, MinT-I that is our proposed platform is compared with the Californium, nCoAP and existing MinT

Figure 10 shows the throughput characteristics achieved with MinT, Californium, nCoAP and MinT-I in terms of total throughput and average throughput. In this experiment, the number of clients varies between 1 and 10,000. This figure represents the number of packets for a 60-s time period at each stage. As shown in the figure, MinT-I, MinT and Californium produce the same throughputs up to 700 clients.

Beyond that point, Californium shows a significant increase in throughput in the interval between 800 and 1,500 clients. Beyond the 1,500-client mark, the throughput of Californium begins to decrease sharply. This is owing to the fact that in Californium, when the number of requests reaches the point where the processing speed of the server becomes overmatched, the working queue volume increase significantly. The design of MinT simplifies the processing procedures for client requests and optimizes the memory usage. As a result, even if the number of clients simultaneously aceesing the IoT network increases, MinT shows better performance compared to Californium.

The design of MinT-I improved the performance of system handler that MinT processes the requests efficiently. As shown in the figure, MinT-I provides a performance increase compared to MinT, Californium and nCoAP since 800 clients. In addition, it keeps the processing performance after 800 clients. It does not process the requests with the maximum performance of the platform. It adjusts the usage of the resources according to amount of received requests and enhances the efficieny of resource usage.

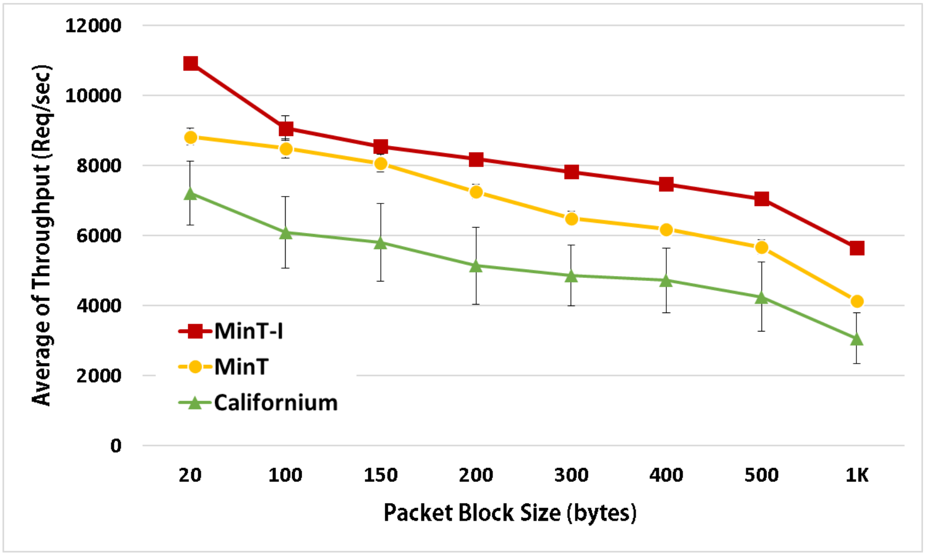


Figure 11. Average of Throughput by Packet Block Size of MinT and Californium (SE).

5.4 Performance to Packet Block Sizes in IoT Midlewares

Figure 11 shows the variation of average throughput according to the packet block sizes in MinT-I, MinT and Californium. In our experiments, the block size are increased from 20 bytes to 1KB (including the header). In this experiment, the number of clients varies between 1000 and 10,000. As the block size increase, these figures show that the average throughput decreases for MinT-I, MinT and Californium. As shown in figure, MinT-I provides a performance increase of 5% compared to MinT and 30% compared to Californium across all the block sizes. In addition, if each thing exchanges the packet included many information at one swoop such as Discovery, the packet size might be 100 ~ 200 bytes. In this range, MinT-I demonstrates an average performance improvement of 5% compared to MinT and 30% compared to Californium. The performance of nCoAP is not a part of the performance comparisons owing to the fact that it showed the lowest performance of the three platforms.

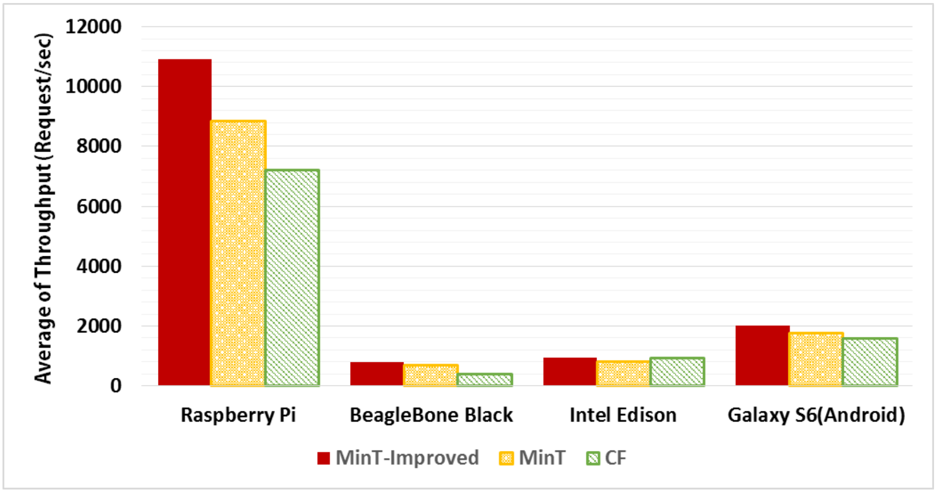


Figure 12. Average of Throughput by Hardware Platforms with MinT, Proposed MinT or Californium (SE)

5.5 Performance on Various Hardware platforms

Figure 12 shows the average throughput performance on various hardware platforms for MinT-I, MinT and Californium. In the experiments, the Raspberry Pi 3, Intel Edison, BeagleBone Black, and Galaxy S6 Android are used. Both the Raspberry Pi 3 and Galaxy S6 have high performance quad core CPUs. On the other hand, the Intel Edison and BeagleBone Black are equipped with low performance dual core CPUs. When compared to Californium, MinT shows about 19% better performance on the Raspberry Pi 3, about 8% better on Galaxy S6, and about 45% better on BeagleBone Black. In addition, MinT-I shows about 19% better on the Raspberry Pi 3, about 12% better on Galaxy S6, and about 13% better on Beagle Bone Black compared to MinT. The proposed MinT-I shows better performance than MinT and Californium. Therefore, MinT-I can be efficient approach to achieve low transmission delays and enhanced battery life an IoT environment.

As shown in figure, the results show that MinT-I, MinT, and Californium demonstrates the high average throughput in the Raspberry Pi 3 based on Linux. However, in the case of Galaxy S6, they represent an average throughput decrease of 75% compared to Raspberry Pi 3. In section 5.2, we discussed the reliability of wireless and wired network environments. In addition, owing to we proved that the platform on the wireless envrionments was influenced by network interference very much, we expect the throughput of Galaxy S6 on wireless to decrease.

6. Conclusions

Recently, many researchers have been studying middleware for using in variable IoT environments. Toward this objective, the MinT was proposed to improve the operational efficiency of IoT systems. The MinT enables developers to operate IoT devices easily and promotes efficient interactions among the associated devices. In particular, the MinT uses a thread pool based on multi-thread to process the received packets from the other IoT devices quickly. The previous of the MinT assigned the number of threads in each thread pool to be equal to the number of CPU cores. However, because this could not adapt to changing throughput based on the received packets, the middleware incurred performance degradation, excess energy consumption and waste of resources and memories.

In this paper, we proposed a MinT-I that extends new method called VTA for optimizing thread pool management to solve the above problems. The MinT-I analyzes the received requests and the throughput of each thread pool using the Performance Observer in the System Layer. Each thread pool in the System Layer decides the number of threads based on amount of the information, which is analyzed in the Performance Observer. In particular, we aimed to increase the performance of the ITPG that processes the received packets. Therefore, we tested the performance of ITPG and evaluated the efficiency and availability of the MinT-I.

To test the availability of the MinT-I, we compared the throughput of the NR and HTP while operating the proposed method in the ITPG. The results showed that the number of threads and processing speed of the HTP were changed by the amount of received requests. After evaluating the availability of the proposed method, we compared it with the previous MinT, Californium, and nCoAP.

The evaluations were performed to assess the throughputs for various packet block sizes and on several leading hardware platforms. In our experimental results, the MinT-I showed improved performance by 25%, 35% and 62%, compared to existing MinT, Californium and nCoAP middlewares, respectively. In addition, the MinT-I shows better performance than MinT and Californium in the performance of hardware platforms. Because the MinT-I responded more quickly than the existing MinT middleware, MinT-I based IoT implementation is considered to provide significant advantages in terms of reduced latency and energy consumption in IoT environment.

Recently, IoT middlewares have been developed that connect with many sensing devices, aggregate peripheral information, and share their resources with each other. The MinT starts to aggregate peripheral information when clients request information from the MinT. However, as the number of requests to the MinT increases, the energy consumption can also be greatly increased because the aggregation cycle of the MinT also increases. In particular, periodic aggregation from sensing devices, which do not update information frequently, can cause transmission delays and excess energy consumption. In future work, we will propose an efficient resource management method to control the aggregation cycle according to accuracy, significance, and frequency of information, with the final result of reducing energy consumption.

Reference

[1] Da Xu, L., He, W. and Li, S., "Internet of things in industries: A survey," IEEE Transactions on Industrial Informatics, Vol 10, No 4, pp.2233-2243, 2014.

[2] Atzori, L., Iera, A. and Morabito, G., "The internet of things: A survey," Computer networks, Vol 54, No 15, pp.2787-2805, 2010.

[3] Whitmore, A., Agarwal, A. and Da Xu, L., "The Internet of Things?A survey of topics and trends," Information Systems Frontiers, Vol 17, No 2, pp.261-274, 2015.

[4] C. S. Lee, S. B. Jeon, Y. T. Han, and I. B. Jung, “Integrated Platform for Device Development in Internet of Things,” Proc. of the KIISE Korea Computer Congress 2015, pp.471-473, 2015.

[5] S. B. Jeon. and I. B. Jung., “MinT: Middleware for cooperative interaction of things,” Sensors, Vol 17, No 6, pp.1452, 2017.

[6] Shelby, Z., Hartke, K. and Bormann, C., (2014, June). The Constrained Application Protocol(CoAP) [Online]. Available: https://www.rfc-editor.org/info/rfc7252 (downloaded 2016. Dec. 09)

[7] Hunkeler, U., Truong, H. L., and Stanford-Clark, A., “MQTT-S?A publish/subscribe protocol for Wireless Sensor Networks,” In Communication systems software and middleware and workshops, pp. 791-798, 2008.

[8] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E. and Culler, D., “Tinyos: An operating system for sensor networks”, pp. 115-148, Heidelberg, Berlin, 2005.

[9] Dunkels, A., Gronvall, B., and Voigt, T., “Contiki-a lightweight and flexible operating system for tiny networked sensors,” In Local Computer Networks, 29th Annual IEEE International Conference on IEEE, pp.455-562, 2004.

[10] Soldatos, J., Kefalakis, N., Hauswirth, M., Serrano, M., Calbimonte, J. P., Riahi, M., and Skorin-Kapov, L., “Openiot: Open source internet-of-things in the cloud,” In Interoperability and open-source solutions for the internet of things, pp.13-25, 2015.

[11] Klues, K., Liang, C. J. M., Paek, J., Musaloiu-Elefteri, R., Levis, P., Terzis, A., and Govindan, R., “TOSThreads: thread-safe and non-invasive preemption in TinyOS,” In SenSys, Vol 9, pp.127-140, Nov., 2009

[12] McCartney, W. P., and Sridhar, N., “Stackless preemptive multi-threading for TinyOS. In Distributed Computing in Sensor Systems and Workshops,” 2011 International Conference on IEEE, pp.1-8, June, 2011.

[13] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M., “Protothreads: simplifying event-driven programming of memory-constrained embedded systems,” In Proceedings of the 4th international conference on Embedded networked sensor systems, pp.29-42, Oct, 2006.

[14] IoTivity [Online]. Available: http://www.iotivity.org (accessed on 2016, Dec. 09)

[15] Kovatsch, M., Lanter, M. and Shelby, Z., "Californium: Scalable cloud services for the internet of things with coap," Internet of Things (IOT), 2014 International Conference on the. IEEE, pp. 1-6, 2014.

[16] nCoAP [Online]. Available: https://github.com/okleine/nCoAP (accessed on 2016. Dec. 09)

[17] Brillo [Online]. Available: http://developers.google.com/brillo (accessed on 2016, Dec. 09)

[18] Tizen [Online]. Available: https://www.tizen.org/ (accessed on 2016, Dec. 09)

[19] Schmidt, B. K., Lam, M. S., and Northcutt, J. D., “The interactive performance of SLIM: a stateless, thin-client architecture,” In ACM SIGOPS Operating Systems Review, Vol. 33, No. 5, pp.32-47, Dec, 1999.

[20] Ishaq, I., Hoebeke, J., Moerman, I. and Demeester, P., “Experimental Evaluation of Unicast and Multicast CoAP Group Communication,” Sensors, Vol 16, No 7, p.1137-1165, July. 2016.