KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.9 pp.385~396 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.9.385

Experimental Evaluation and Flexible Performance Improvement of IoT Middleware for Efficient Connectivity

Jeon Soo Bin† · Lee Chung San†† · Han Young Tak††† · Jung In Bum††††

ABSTRACT

Many IoT platforms have been proposed for various IoT devices, from low-end to high-end performance. We previously proposed a new IoT platform called MinT that supports the operation of the sensing devices and network communication. In the proposed platform, the things can flexibly connect to each other and efficiently share their information. Most IoT platforms, including the MinT, support thread pooling to quickly process requests. However, using a thread pool with a fixed thread count can cause network delay and inefficient energy consumption. In this paper, we propose an enhanced method to manage the thread pool efficiently by adjusting the number of threads every cycle to regulate the device's performance. In particular, we aim to improve the performance of the Interaction Thread Pool Group, which is responsible for analyzing, processing, and re-transmitting the received packets. The experiment shows that the improved method increases the average throughput by approximately 25% compared to the existing platforms. Finally, using the proposed method, the MinT can reduce the transmission delay and energy consumption of devices in the IoT environment.

Keywords : Internet of Things, Middleware, Resource Sharing, Multi-Threading, Thread Pool

사물간의 효율적인 연결을 위한 사물인터넷 미들웨어 실험 평가 및 성능 향상 방법

전 수 빈† · 이 충 산†† · 한 영 탁††† · 정 인 범††††

요 약

IoT 환경에서 제한적인 디바이스들의 운영을 위한 많은 IoT 미들웨어들이 연구되었다. 우리는 센싱 디바이스 및 네트워크의 통합적인 운영을 위한 IoT 미들웨어인 MinT를 제안하였다. MinT는 센싱 및 네트워크 디바이스를 통합 관리할 수 있는 센서 추상화 계층, 완성도 높은 시스템 및 통신 계층을 지원한다. 또한 이를 통해 사물들은 이기종 네트워크간의 유연한 연결 및 효율적인 정보 공유를 할 수 있다. MinT의 시스템 계층은 신속한 정보처리를 위해 스레드 풀을 사용한다. 하지만 고정 스레드 크기를 사용하는 방법은 네트워크 지연 및 비효율적인 베터리 소비를 보여준다. 본 논문에서는 가변적으로 스레드 개수를 조정할 수 있는 향상된 스레드 풀 관리 방법을 제안 한다. 특히 수신되는 패킷 분석 및 처리, 재송신을 담당하는 Interaction Thread Pool그룹의 성능 향상을 목표로 한다. 본 논문에서는 실험을 통해 평균 요청 처리율이 평균 25% 증가한 것을 입증하였다. 결국 제안하는 방법을 통해 MinT는 IoT 환경에서 전송 지연 및 디바이스의 에너지 소비를 더 감소시킬 수 있다.

키워드 : 사물인터넷, 미들웨어, 정보공유, 멀티스레드, 스레드풀

1. 서 론

사물인터넷(IoT: Internet of Things)은 센서 기술을 포함하는 다양한 IT 기술을 사람 또는 모든 사물(Things)에 적용하고 네트워크 기술을 통해 상호 협력 관계를 형성하는 사물간의 네트워크를 의미한다. 최근 IoT 환경을 구성하기 위한 많은 IoT 미들웨어가 연구되고 있다. IoT 미들웨어는 제한적인 사물들에게 효율적인 연결성을 보장하고 사물들의

※ 이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2016R1A2B4010018).
※ 이 논문은 2016도 강원대학교 대학회계 학술연구조성비로 연구하였음 (관리번호 520160259).
† 준 회 원 : 강원대학교 컴퓨터정보통신공학과 박사
†† 준 회 원 : 강원대학교 컴퓨터정보통신공학과 박사과정
††† 준 회 원 : 유엔젤 연구원
†††† 종신회원 : 강원대학교 컴퓨터정보통신공학과 교수
Manuscript Received : February 22, 2017
First Revision : May 10, 2017
Second Revision : June 9, 2017
Accepted : July 12, 2017
* Corresponding Author : Jung In Bum(ibjung@kangwon.ac.kr)

자원을 효과적으로 관리할 수 있도록 하였다. 현재 많은 연구자들은 아래와 같이 IoT 미들웨어의 대표적인 문제점을 지적하고 이를 해결하기 위해 노력하고 있다 [1]:

- 다양한 능력을 가진 IoT 하드웨어 플랫폼 관리 문제
- 리소스 수집 및 관리를 위한 수많은 센서 및 네트워크 디바이스의 통합 제어 및 운영
- 다양한 네트워크 환경 및 하드웨어 플랫폼의 증가로 인한 이기종 사물간의 상호 연결성 부족
- 높은 수준의 능력을 요구하는 IoT 디바이스 개발 환경

IoT 기기들은 다른 기기와 정보를 공유하고 사용자 또는 다른 기기에게 서비스를 제공하는 것을 목표로 한다. IoT디바이스의 역할이 기존 WSN의 단순한 수집, 전송 구조가 아닌 수집, 전송, 가공, 서비스 구조로 변경된 것이다. 그러므로 IoT 기기들간의 주기적인 정보 요청이 발생하고 하나의 기기에 정보가 집중될 수 있는 경우도 발생한다. 이런일을 막기 위해 대표적으로 평면 프로토콜(Plat protocol) 또는 계층 프로토콜(Hierarchical protocol)의 다양한 네트워크 프로토콜을 이용해 IoT 환경을 구성해야 한다 [2-4].

하지만 IoT 기기들이 서로의 요청에 대한 처리를 정해진 시간 내에 해내지 못한다면 전체 네트워크의 전송이 지연되고 전체 서비스 지연으로 발전할 수 있다. 이러한 요청에 대한 신속한 정보의 처리를 위해 기존의 제약적인 디바이스에서는 TinyOS[5] 및 Contiki[6]와 같은 이벤트 기반의 미들웨어들이 주로 사용 되었다. 또한 병행 실행 모델의 적용을 통해 통신, 처리 및 센싱 제어에 유연성을 제공하고 있다. 하지만 실행되는 태스크는 순환적 실행 개체 방식을 따르고 있기 때문에 다양한 태스크 및 정보를 동시에 제어할 수 없고 네트워크 손실 및 지연이 증가할 수 있다.

현재 연구되고 있는 많은 IoT 미들웨어는 멀티 스레드를 지원한다. 특히 디바이스제어, 리소스 관리, 통신 부분에 멀티 스레드를 사용하여 효율적인 운용이 가능하게 한다. IoT 미들웨어는 요청에 대한 신속한 처리를 위해 사물들과의 연결 및 처리과정에 많은 자원을 활용하고 있다. 대부분의 미들웨어들은 다른 디바이스와의 통신을 위해 넌블록(Non-block) 소켓을 사용하며 비동기 통신을 선호한다. 비동기 통신에서는 요청에 대한 연속적인 스레드의 생성을 막기 위해 스레드 풀을 사용한다. 특히 스레드 풀의 사이즈를 CPU의 수와 같은 고정 스레드 수를 사용하거나 요청에 따라 새로운 스레드를 생성하는 방법을 사용한다.

고정된 스레드 수를 사용함으로써 CPU의 효율성을 높일 수 있지만 불필요한 자원의 낭비를 발생시킬 수 있다. 예를 들어 디바이스의 처리량 보다 적은 양의 요청이 들어 왔을 때 멀티 스레드는 큰 도움이 될 수 없을 뿐만 아니라 자원 낭비의 원인이 된다. 또한 일반적인 멀티스레드 운영체제에서는 태스크 마다 독립적인 스택을 할당해서 사용하기 때문에 많은 수의 스레드 생성은 메모리 소요량이 늘어나는 문제가 발생한다.

이런 문제를 해결하기 위해 태스크 간의 스택 공유를 통해 메모리 소요량을 줄이는 가상 스레딩 방법에 대한 노력도 이루어지고 있다. 가상 스레드를 이용하여 메모리 소요량을 줄일 수 있지만 가상의 스레드의 스택 공유로 인한 실행 과정의 오버헤드가 비교적 클 수 있다. 그러므로 직접적인 스레드 수의 조절을 통해 메모리 및 자원을 효율적으로 사용하고 기기들로부터 요청된 정보를 신속하게 처리할 수 있는 방법이 필요하다.

우리는 다양한 분야에서 효율적으로 IoT 환경을 구성할 수 있는 IoT 미들웨어인 MinT(Middleware for Interaction of Things)를 제안하였다[7]. MinT는 개발자들이 쉽게 디바이스를 개발 운용할 수 있도록 하고 사물간의 효율적이고 빠른 상호작용을 가능하게 하였다. MinT는 수많은 사물들로부터 요청되는 패킷들을 효율적으로 처리하기 위해 각 단계 별로 멀티 스레드 기반의 스레드 풀을 사용한다. MinT는 쓰레드의 직관적인 관리를 위해 각 스레드의 수를 일반화 하여 사용하고 있다. 각 스레드 풀의 스레드 수는 처리량과 상관 없이 하드웨어 플랫폼의 Core수와 동일한 크기로 설정된다. 특히 정보의 요청을 처리하는 과정에서 고정된 스레드 크기는 불필요한 자원을 사용하는 경우가 생길 수 있기 때문에 에너지 효율성이 감소할 수 있다. 또한 요청되는 패킷 량에 유연하게 대처할 수 없기 때문에 처리 지연이 발생할 수 있다.

본 논문에서 우리는 기존 MinT에 스레드를 가변적으로 조절할 수 있도록 향상된 스레드 관리 방법(VTA: Variable Thread Adjustment)을 추가한 MinT-I (Improved MinT)를 제안한다. 특히 본 논문은 요청되는 정보의 빠른 처리를 위해 수신되는 패킷 분석 및 처리, 재송신을 담당하는 Interaction Layer의 처리 스레드 풀(HTP: Handler Thread Pool)의 성능 향상을 목표로 한다. HTP에 적용된 VTA는 수신되는 정보의 양에 따라 스레드의 수와 처리량을 조절하기 때문에 스레드 사용으로 인한 자원 및 메모리 관리를 최적화 할 수 있다. 또한 수신되는 정보의 양에 따라 최적의 처리량을 유지해 줄 수 있기 때문에 요청에 대한 신속한 처리가 가능하고 네트워크 환경의 전송 지연을 감소시킬 수 있다.

본 논문은 HTP 중심의 실험 및 분석을 통해 본 논문에서 제안한 방법의 효율성을 입증한다. 또한 기존IoT 미들웨어와의 비교를 통해 본 논문에서 제안한 방법의 요청 처리 효율성에 대한 평가를 실시했다. 평가는 기존 미들웨어와의 처리량 비교, 패킷 블록사이즈 별 처리량 비교, 하드웨어 플랫폼 별 처리량 비교를 진행 하였다. 제안하는 IoT 미들웨어의 성능평가를 위해 총 10대의 클라이언트 플랫폼과 3종류의 테스트 베드를 사용한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 오픈소스 IoT 미들웨어의 스레드 관리 방법에 대해 알아본다. 3장에서는 MinT 미들웨어에 대해 소개하고 문제점을 분석한다. 4장에서는 향상된 Thread 관리 최적화 방법을 알아본다. 5장에서는 제안하는 방법의 성능평가를 기술하고 6장에서는 결론 및 향후 계획을 제시한다.

2. 관련 연구

IoT 디바이스 구성 및 연결성을 위해 위 오픈소스 미들웨어들은 많은 도움을 주고 있다. 특히 다양한 종류의 센싱 디바이스 제조사들과의 협력을 통해 센싱 디바이스의 통합 관리가 가능하다. CoAP (Constrained Application Protocol) [8], MQTT (Message Queue Telemetry Transport) [9] 등과 같은 어플리케이션 프로토콜을 통해 이기종 디바이스 간의 연결성 또한 보장되고 있다. IoT 디바이스는 디바이스간의 유연한 연결을 통해 자신들의 정보를 공유하는 것을 목표로 한다. IoT은 제한적인 성능을 가지는 저사양 하드웨어 플랫폼부터 멀티코어 프로세서를 탑재한 고 사양 하드웨어 플랫폼들을 효율적으로 연결하고 운영할 수 있는 방법이 필요하다. 특히 IoT 기기들이 서로의 요청에 대한 처리를 정해진 시간 내에 해내지 못한다면 전체 네트워크의 전송이 지연되고 전체 서비스 지연으로 발전할 수 있다.

요청에 대한 신속한 정보처리를 위해 기존의 제약적인 디바이스에서는 이벤트 기반의 미들웨어가 주로 사용되었다. 대표적인 미들웨어로는 TinyOS [5]와 Contiki [6], OpenIoT [10]가 있다. 두 미들웨어는 연결된 센싱 디바이스와 통신의 병렬적인 제어를 위해 이벤트 기반의 병행 실행 모델을 지원한다. 하지만 실행되는 태스크는 순환적 실행개체 방식을 따르고 있기 때문에 다양한 태스크 및 정보를 동시에 제어해야 하는 환경에 적합하지 않다.

TOSThread는 TinyOS에 일반적인 멀테스레드 모델을 추가한 방법이다. 하지만 TOSThread을 통해 멀티스레드를 생성할 때 스레드별로 스택을 할당해야 하는 문제가 발생하기 때문에 소형 디바이스의 메모리 관리에 문제가 발생한다. UnstackedC는 메모리 소요량을 줄이기 위해 스레드별로 스택이 필요하지 않도록 변형한 TinyOS 멀테스레드 모델이다. 하지만 하나의 스택을 공유하기 위해서는 실행중인 스레드가 다른 스레드에 의해 선점되어서는 안된다. 또한 스레드가 실행 중에 대기해야 하는 경우에는 모든 지역변수들을 별도의 자료구조로 관리하는 번거로움이 발생하기 때문에 실행 과정에 오버헤드가 비교적 클 수 있다[11, 12].

ProtoThread는 Contiki에 추가하여 사용할 수 있는 멀티스레드 모델이다. ProtoThread는 메모리가 제한된 시스템에서 특정 이벤트에 의해 구동되는 태스크를 가상적으로 여러 개의 스레드로 구성할 수 있도록 하였다. 이 방법은 스레드별로 별도의 스택을 할당할 필요가 없기 때문에 메모리 용량을 절약할 수 있다. 하지만 기본적으로 비지웨이팅 방식을 사용하고 있기 때문에 CPU의 동작을 일시적으로 제어할 수 없다. 또한 기본적으로 싱글 스레드 모델을 가상화 한 방법이기 때문에 멀티 프로세서 환경에 적용할 수 없다[13].

최근 하드웨어 기술의 발전으로 인해 많은 고성능의 소형 하드웨어 미들웨어가 개발되고 있다. 이에 따라 현재 연구되고 있는 많은 IoT 미들웨어는 고성능 하드웨어에서도 사용할 수 있도록 멀티스레드 모델을 지원한다. 특히 대용량의 IoT 네트워크 망에서는 하나의 사물에 정보를 요청하는 횟수가 많아지고 각 사물들은 이런 동시 연결성에 신속하게 처리할 수 있는 능력이 필요할 수 있다.

IoTivity는 리눅스 재단(Linux Foundation)이 주관하는 IoT 소프트웨어 프레임워크이다. IoTivity의 구조는 근접/원격 장치 및 자원을 발견하는 Discovery, 메시지와 스트리밍 모델 기반 정보를 교환 및 제어하는 Data Transmission, 장치의 구성과 권한 등을 관리하는 Device Management, 데이터의 수집, 저장, 분석을 담당하는 Data Management로 이루어져 있다. 특히 IoTivity는 메시지 처리를 위해 고정 스레드풀링을 사용하며 요청되는 메시지를 동시에 처리할 수 있다. 또한 IoTivity는 저성능 및 고성능 디바이스간의 효율적인 연결을 위해 기본적으로 CoAP 어플리케이션 프로토콜을 사용한다[14].

Californium은 JAVA로 개발된 CoAP 기반의 오픈 소스 연결 미들웨어다. Californium은 여러 네트워크 프로토콜과의 연결을 위한 Network Stage, 요청 메시지를 처리하기 위한 CoAP 기반의 Protocol Stage, 사용자들에게 서비스를 제공하기 위한 Business logic stage로 구분된다. 특히 Californium은 Protocol Stage를 단순화 하고 멀티코어 스레드 기반의 구조화를 통해 집중된 요청에 대한 신속한 처리가 가능하도록 하였다[15].

nCoAP (Netty CoAP)는 netty 프레임워크를 사용한 CoAP 프로토콜을 기반으로 하는 IoT 오픈소스 연결 미들웨어다. 기본적인 통신 프레임워크는 netty를 기반으로 하고 있기 때문에 넌블럭 형태의 비동기 통신을 사용한다[16].

AndroidThings는 구글이 개발한 IoT 미들웨어로 안드로이드의 커널 등 하위 시스템을 기반으로 한다. AndroidThings는 저전력, 저사양의 제한된 IoT 기기들의 크로스 플랫폼을 목표로 하고 있으며 BLE, WiFi 등의 M2M (Machine-to-Machine) 통신을 기본으로 클라우드 시스템을 지원한다. 또한 Weave 프로토콜을 통하여 AndroidThings장치들은 이기종의 IoT 미들웨어들과 상호작용할 수 있다. AndroidThings 프로젝트의 특징은 안드로이드 기반이지만 기존 안드로이드보다 하드웨어 요구사항을 대폭 낮추어 32~64MB의 메모리를 가진 가전 기기에서도 안정적으로 동작한다[17].

IoT는 수많은 사물들이 상호 연결되어 리소스를 공유한다. 위 미들웨어들은 확장성 있는 IoT 클라우드 서비스를 위한 효율적인 시스템 구조를 제공한다. 특히 외부와 연결되어 요청되는 정보를 처리하는 연결 처리 부분은 멀티스레드 기반의 스레드 풀을 사용한다. 위 미들웨어들은 기본적으로 최대 Thread의 수를 하드웨어 플랫폼의 CPU 코어 개수로 고정해서 제어 하고 있다. 저전력, 저 성능으로 동작하는 IoT 미들웨어에서 스레드 수의 증가는 성능 저하의 원인이 될 수 있으며 불필요한 작업량으로 인한 전력 소비의 원인이 될 수 있다. 그러므로 기본적인 하드웨어 플랫폼의 능력에 맞는 성능을 유지하기 위해 CPU 코어의 수로 고정한다.

마지막으로 Tizen은 센서 관리자를 통해 다른 기기의 요청을 받는다. 센서관리자는 각 요청 이벤트 마다 워커 스레드를 생성한다. 그러므로 요청이 많이 발생함에 따라 스레

드의 수가 계속해서 증가하게 된다. 지속적인 스레드 증가는 CPU 자원 및 메모리 오버헤드를 발생 시키며 프로세스를 정지시킬 위험이 있다[18].

요청되는 다수의 정보를 빠르게 처리하기 위해 대부분의 IoT 미들웨어는 멀티스레드 기반의 정보 처리를 사용한다. 하지만 CPU Core 개수에 따른 스레드 풀의 스레드 개수의 고정은 정보의 요청을 처리하는 과정에서 불필요한 스레드 사용으로 인한CPU 자원 및 메모리 낭비를 발생시킬 수 있다. 그러므로 본 논문에서는 스레드를 가변적으로 조절할 수 있는 향상된 스레드 관리 방법을 제안한다. 제안하는 방법은 스레드를 물리적으로 조절함으로써 가상화로 인한 구조의 복잡성 및 오버헤드를 감소할 수 있고 고정된 스레드에 비해 메모리 사용 및 성능을 높일 수 있다.

3. MinT 미들웨어의 구조 및 문제점 연구

MinT는 분산 IoT 환경에서 동작될 수 있는 IoT 디바이스를 제작하기 위한 미들웨어다. MinT는 IoT를 위한 미들웨어의 요구사항을 만족하기 위해 다양한 센서 및 네트워크 디바이스 개발을 위한 HAL 지원, 사물간의 능동적인 상호작용을 위한 연결 관리, 에너지 효율적인 센서 디바이스 및 데이터 관리, 쉽고 편리한 IoT 디바이스 개발을 위한 상위 인터페이스 제공 등을 목표로 하고 있다[7].

본 논문에서는 MinT의 스레드 풀 관리 방법의 단점을 지적하고 새로운 스레드 풀 관리 기법을 제안한다. 그러므로 본 단원에서는 MinT에서 스레드 풀을 관리하는 시스템 계층 구조에 대해 살펴보고 문제점에 대한 분석을 실시한다.

3.1 MnT 시스템 계층(System Layer) 구조

Fig. 1은 MinT의 시스템 레이어 전체 구조도를 보여준다. System Layer는 MinT를 효율적으로 운영하기 위한 레이어로 Device Management, Service Management, System Scheduler, Resource Management 모듈을 가진다.

Device Management는 finder와 linker 모듈을 이용하여 MinT에 연결된 센싱 및 네트워크 디바이스 드라이버를 찾아내고 연결 한다. Handler 모듈은 연결된 디바이스의 데이터와 동작을 관리한다. 또한 Resource Management의 Resource Handler와 연결을 통해 데이터를 저장하고 외부로부터 요청한 동작을 수행시킨다.

Service Management는 MinT의 서비스를 위한 Application을 개발할 때 사용된다. Application 개발자는 제공되는 API를 이용하여 MinT의 센서 제어, 리소스 요청, 다른 사물의 리소스 요청, 리소스 공개 등의 다양한 Application 개발이 가능하다. MinT에는 여러 개의 서비스가 생성될 수 있고 모든 서비스의 동작은 System Scheduler에 의해 관리된다.

Resource Management는 MinT의 모든 리소스를 저장하는 모듈이다. 리소스를 저장하기 위한 저장공간은 Local과 Network 저장소로 구분된다. Local 저장소는 MinT에 연결된 디바이스로부터 수신된 리소스를 저장한다. Network 저장소는 Interaction layer를 통해 외부 사물로부터 수신된 리소스를 저장한다. 또한 효율적인 재 요청 관리를 위해 네트워크 캐시 저장소로도 사용한다. Resource Handler는 Device Handler와 직접적으로 연결되는 모듈이다. 이 모듈은 센싱 디바이스로부터 수신된 리소스를 저장하거나 센싱 디바이스에 직접 명령을 주기 위해 사용된다. Resource Manager 모듈은 Device Management로부터 연결된 리소스 및 외부 사물로부터 연결된 네트워크 리소스를 관리한다.

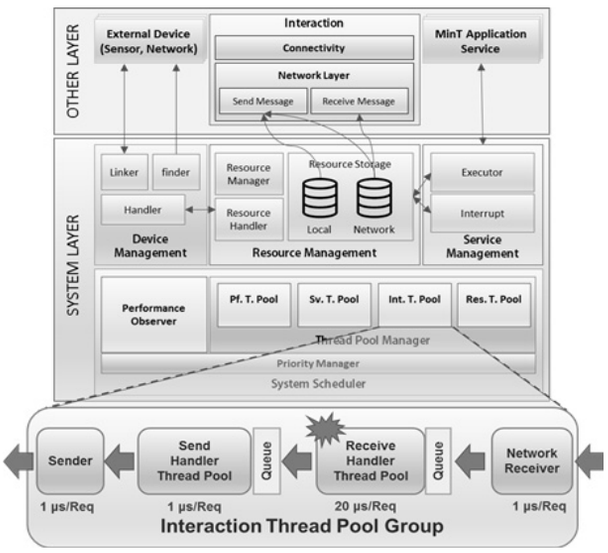


Fig. 1. System Layer and Interaction Thread Pool Architecture

System Scheduler는 System Layer, Interaction Layer, Application layer 에서 생성되는 스레드를 효율적으로 관리하기 위한 통합 스레드 관리 도구이다. Fig. 1과 같이 System Scheduler는 Thread Pool Manager (TM), Priority Manager (PM), Performance Observer (PO)로 구성된다. Thread Pool Manager는 총 4개의 스레드 풀 그룹을 관리한다. 그룹의 종류는 다음과 같다.

1) Application Service: System Layer의 Service Management 모듈에서 사용되고 MinT 어플리케이션을 운영하기 위한 서비스들의 그룹이다. Application마다 생성되는 스레드의 수가 다르고 모든 서비스가 동시에 동작될 수 있기 때문에 최대 스레드 풀 크기는 적응적으로 서비스의 수만큼 증가/감소한다. 또한 대기 큐가 없는 것이 특징이다.

2) Resource Management: 연결된 외부 센싱 디바이스를 통해 수집된 리소스 핸들러 관리 그룹이다. 최대 Thread Pool Size는 CPU 코어 수만큼 설정된다. 대기 큐 크기는 I/O양이 많지 않기 때문에 100으로 설정한다.

3) Interaction: Interaction Layer에서 사용되며 Protocol Adapter, Send 및 Receive Handler의 스레드 풀 관리 그룹이다. 최대 스레드 풀 크기는 CPU 코어 수만큼 설정된다. 이 그룹은 MinT내에서 가장 많은 작업을 처리하는 그룹이다. 그러므로 그룹의 작업 양, 하드웨어 플랫폼의 성능에 따라 최대 스레드 풀 크기 및 우선순위를 변경 시켜야 할 수도 있다. 하지만 MinT에서는 이를 위한 방법 및 실험을 하

지 않기 때문에 최대 스레드 풀 크기는 CPU 코어 수만큼 설정한다. 대기 큐 크기 설치되는 하드웨어 플랫폼의 메모리 크기에 따라1kb~10MB의 크기로 설정한다.

4) Performance Profile: System Layer의 Performance Observer에서 사용되며 각 스레드 풀 그룹들의 성능 측정 및 관리한다. 이 그룹의 스레드는 각 스레드 풀 별로 1개씩 생성되어 성능을 측정한다.

3.2 특정 핸들러의 작업 처리 지연 문제

MinT는 Interaction Layer를 통해 외부로부터 요청되는 메시지를 수신한다. 수신된 메시지는 Interaction Layer의 Network Layer에서 메시지 매칭, 교환, 핸들링을 위해 단계별로 동작된다. Interaction Layer의 단계별 메시지 처리 과정은 Fig. 1과 같이 총 2개의 모듈 및 2개의 스레드 풀을 통해 진행된다. 각 모듈 및 스레드 풀은 Interaction thread pool 그룹에서 관리된다. Protocol Adapter의 Network Receiver를 통해 수신된 메시지는 Network Layer의 HTP Queue에 추가된다. HTP는 Queue에 추가된 메시지를 꺼낸 후 패킷 분석 및 목적지 판별 과정을 통해 메시지의 요청의 요구사항을 분석한다. MinT는 요구사항 분석에 따라 센싱 디바이스를 제어하거나 저장되어 있는 리소스를 송신하기 위한 작업을 시작한다. 요구사항에 대한 응답 준비를 마친 메시지는 다시 Send Handler Queue로 전달된다. Send Handler Queue는 전달된 메시지를 CoAP등과 같은 어플리케이션 프로토콜에 맞는 패킷으로 변환한 후 Hardware Platform Adapter를 통해 각 목적지로 전송된다.

Interaction Layer에서 동작되는 모든 스레드 풀들은 System Scheduler의 Thread Pool Manager를 통해 제어된다. Thread Pool Manager는 MinT에서 가장 많은 작업량을 수행하는 Interaction Thread Pool 그룹에게 가장 많은 Thread를 할당하고 있다. MinT는 기본적으로 최대 Thread의 수를 하드웨어 플랫폼의 CPU 코어 개수로 고정해서 제어 하고 있다.

서버 환경에서 수많은 요청에 대한 처리를 위한 멀티 스레드 사용은 기본적인 방법이 되고 있다. 그러므로 기존 MinT 및 미들웨어에서는 IoT 환경을 위한 효율적인 정보처리를 위해 대부분 스레드 풀의 크기를 CPU Core의 크기로 고정하여 사용하고 있다. 이 방법은 CPU의 성능에 따라 스레드 풀의 개수를 고정해 하드웨어의 성능 저하 및 전력 소비를 줄일 수 있다고 판단 했다. 하지만 요청에 대한 빠른 처리를 위해 스레드의 수를 늘려야 하는 상황에서 CPU Core의 수가 1개뿐인 디바이스는 1개의 스래드를 생성할 수 있기 때문에 성능 저하 문제를 발생시킨다. 또한 고성능의 디바이스에서 4개 이상의 멀티 스레드 환경을 구성할 수 있지만 이 방법 또한 적은 양의 요청이 빈번한 상황에서는 자원 낭비 및 높은 전력 소비의 원인이 된다.

위 문제에 대한 분석을 위해 본 논문에서는 다양한 스레드 환경에서의 메시지 처리량 비교를 실시 하였다. 본 실험을 위해 Raspberry Pi에 고정 스레드를 사용하는 MinT를 적용하여 메시지를 처리할 수 있는 간단한 서버-클라이언트 환경을 구성하였다. Fig. 2부터 Fig. 4는 단일 Core 환경 및 멀티 Core 환경에서 Interaction Layer의 HTP 메시지 처리량을 비교한 그래프이다.

Fig. 2는 Thread Pool 크기를 1개로 제한한 후 클라이언트의 요청에 대한 처리 속도를 보여 준다. Fig. 2의 경우 평균적으로 12000 Request/Sec의 클라이언트 요청(NR: Network Receiver)양을 보이고 있고 처리량(HTP)은 평균 6000 Request/Sec를 보이고 있다. 그러므로 클라이언트 요청량에 1/2의 처리량을 보이고 있는 단일 스레드는 요청에 대한 처리 지연이 발생하며 메시지 큐의 초과로 인한 메시지 손실이 발생한다.

Fig. 3은 Thread Pool 크기를 2개로 제한한 후 클라이언트 요청에 대한 처리 속도를 보여준다. 하지만 이 그래프도 클라이언트의 평균 요청 양인 10000 Request/Sec에 비해 처리량이 평균 6000 Request/Sec에 불과하기 때문에 Fig. 4와

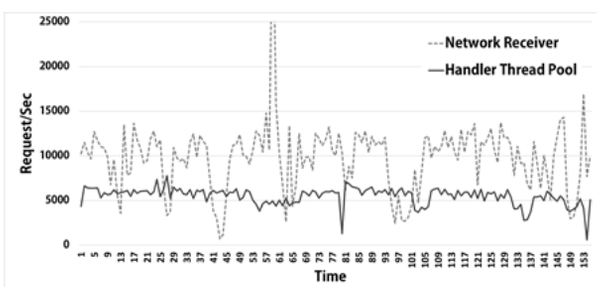


Fig. 2. Throughput of Network Receiver and HTP on Single Thread

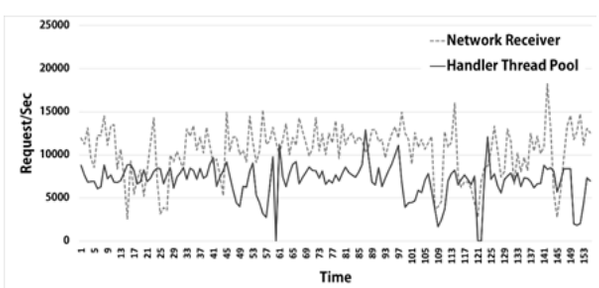


Fig. 3. Throughput of Network Receiver and HTP on 2 Threads

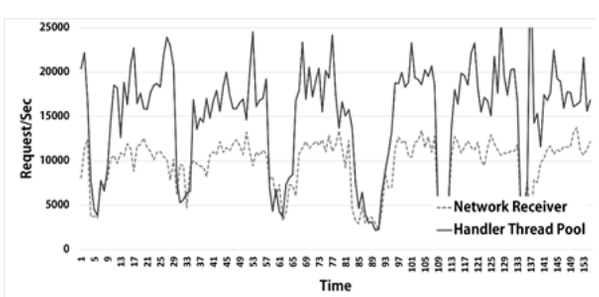


Fig. 4. Throughput of Network Receiver and HTP on 4 Threads

마찬가지로 처리 지연 및 메시지 손실이 발생할 수 있다.

Fig. 4는 총 4개의 Thread로 동작된다. Fig. 2와는 반대로 위 상황은 요청 량(약 10000 Request/Sec)에 비해 처리 량(약 16000 Request/Sec)이 더 많은 것을 볼 수 있다. 그러므로 Fig. 4와 같이 4개의 Thread는 많은 요청을 처리할 수 있다. 하지만 그림과 같이 처리할 수 있는 양보다 요청 양이 적을 때에도 고정된 스레드 수로 인해 많은 CPU 자원을 소유하게 되므로 불필요한 자원을 소모할 수 있다.

기존 스레드 풀 관리는 미들웨어 시스템 운영에 효율적인 도움을 줄 것으로 예상했고 실험을 통해 기존 시스템들보다 더 좋은 성능을 보여주었다. 하지만 위에서 언급한 내용과 같이 CPU의 Core 크기를 활용한 스레드 풀 크기의 관리는 성능 저하 및 자원 낭비에 몇 가지 문제점을 보여주고 있다. 그러므로 본 논문에서는 위 문제를 해결하고 빠른 정보 처리를 위한 스레드 풀 관리의 최적화 방법을 제안한다.

4. 향상된 스레드 풀 관리 최적화 방법

본문 3.3와 같이 요청된 많은 양의 데이터를 처리하기 위해 MinT의 Thread Pool Manager는 멀티 스레드 작업을 수행한다. 하지만 기존 스레드 관리는 CPU의 코어 수를 이용한 고정 개수의 스레드를 사용하기 때문에 시스템의 성능 저하 및 자원 낭비에 문제점을 발생시킨다. 본 논문에서는 위의 문제를 해결하기 위해 스레드 풀의 스레드 개수를 가변적으로 조정할 수 있는 향상된 스레드 풀 관리 최적화 방법을 제안한다.

MinT는 System Layer의 Performance Observer를 통해 Network 및 System 레이어에서 동작하는 모듈들의 성능을 분석한다. 성능 분석은 매초마다 수집되고 과거 10초 이내의 자료를 실시간으로 저장하고 있다. 본 논문에서는 Fig. 1과 같이 System Layer에서 정보의 요청을 담당하는 Interaction Thread Pool 그룹의 HTP성능 향상을 목표로 한다. 그러므로 Interaction Thread Pool 그룹의 HTP 정보의 처리량(Request/Sec)과 Queue 정보를 사용한 가변적 스레드 풀 관리 방법을 제안한다.

Interaction Thread Pool그룹의 향상된 스레드 관리 방법은 다음과 같다. 첫 번째로 요청되는 정보의 양을 이용하여 하드웨어 플랫폼의 성능을 분석 하고 처리에 필요한 스레드 수를 예측한다. 두 번째로 요청에 대한 신속한 처리를 보장하고 시스템의 효율적인 메모리 관리를 위해 Queue의 크기와 Retransmission Timout (RTO)에 대한 가중치를 설정한다. 설정된 가중치는 예비 스레드 수에 적용되어 다음 스레드 수 계산에 이용된다. 세 번째로 지속적으로 스레드 풀의 성능을 측정하고 최대 성능을 유지할 수 있는 최대 스레드 수를 계산한다. 최종적으로 다음 시간 t에 적용될 스레드 수는 최대 스레드 수 범위 안에서 계산된 예비 스레드 수를 이용하여 계산한다.

4.1 정보의 처리량을 이용한 스레드 수 예측

Network Layer의 Network receiver와 Interaction Layer의 HTP와의 처리량 비교를 통해 요청되는 처리에 필요한 HTP의 예비 스레드의 수를 계산한다. 예비 스레드 수는 현재 시간의 Network receiver의 처리량 R_t(Request, Request/Sec)에 만족하기 위해 높여야 하는 HTP의 처리량 H_t(Handler, Request/Sec)을 목표로 한다. 그러므로 R_t와 H_t의 차이값과 최근 n초까지의 HTP의 처리량(HT_t: Handler Trend)을 이용하면 예비 스레드 수를 계산할 수 있다. HT_t에 대한 계산식은 Equation (1)과 같다. 또한 R_t와 H_t의 차이 값을 이용한 HTP의 예비 스레드 수 $PD_{N,t}$의 계산식은 Equation (2)와 같다. n값은 현재 시간으로부터 5초전까지의 데이터 평균값을 사용한다. 5초(1 sec * 5 Interval) 데이터는 가장 최근에 측정되는 HTP의 양과 근접한 데이터를 이용하고 데이터의 변동폭을 줄이기 위해 결정하였다.

$$HT_t = \frac{1}{n}\sum_{i=0}^{n}\frac{H_{t-i}}{N_{t-i}} \tag{1}$$

$$PD_{N,t} = \frac{R_t - H_t}{HT_t} \tag{2}$$

4.2 Queue와 Retransmission Timeout 정보를 이용한 스레드 수 예측

본 논문에서는 요청에 대한 신속한 처리를 보장하기 위해 HTP의 Queue 크기와 RTO를 이용하여 첫 번째 예비 스레드 수에 추가적인 값을 부여한다. 기본적으로 HTP의 Queue 크기는 하드웨어 플랫폼의 1~2%의 양으로 설정된다. Raspberry Pi의 경우 최대 10MB정도의 공간이 동적으로 설정된다. IoT 환경에서 주로 사용되는 대표적인 프로토콜인 CoAP를 통해 수신되는 패킷은 최소 4byte에서 최대 20byte까지의 양을 가진다. 그러므로 Raspberry Pi의 경우 수용할 수 있는 HTP Queue의 최대 사이즈는 최대 약 500000이 될 수 있다. 대용량의 IoT 디바이스는 많은 양의 요청을 수용할 수 있기 때문에 정보의 손실률을 줄일 수 있다.

패킷의 수용 능력과는 별개로 낮은 정보의 처리 능력은 요청에 대한 대기시간의 증가로 이어질 수 있다. 예를 들어 신뢰기반의 CoAP 프로토콜은 손실된 데이터의 재전송을 위해 기본적으로 2~3초의 대기시간을 가진다. 대기시간이 초과되면 CoAP는 해당 패킷을 최대 4번 재전송하고 최대 전송 시간(MAX_RETRANSMIT)은 93s의 재전송 대기시간을 가질 수 있다[14]. 대기시간의 증가는 네트워크 전체의 응답 시간 증가에 영향을 줄 수 있기 때문에 빠른 응답 시간을 위해서는 패킷의 재 전송률을 줄여 정보에 대한 신속하고 효율적인 처리가 이루어질 수 있도록 해야 한다. 또한 대용량 RAM을 사용하고 있는 Raspberry Pi와 같은 경우 많은 양의 정보를 수용할 수 있다. 하지만 대부분의 IoT 디바이스는 적은 용량의 RAM 공간을 사용하기 때문에 신속한 처

리를 통해 Queue의 적재율을 줄여야 한다.

본 논문에서는 요청에 대한 신속한 처리를 보장하기 위해 Queue의 크기와 RTO를 이용하여 첫 번째 예비 스레드 수에 추가적인 값을 부여한다. Equation (3)은 Queue의 크기와 RTO를 이용하여 Equation (2)에 가중치를 주기 위한 가중치를 계산한다. Queue의 크기와 대기시간을 고려하기 위해서는 우선 다음 시간(1초)에 적재되는 큐의 크기를 예측해야 한다. 첫 번째로 현재 큐에 대기하고 있는 패킷의 양(QS_t)을 측정한다. 두 번째로 과거의 데이터를 이용하여 예상되는 패킷의 양(e_t: Expected Demand)을 계산한다. e_t는 현재 시간으로부터 5초 전까지 데이터의 평균값을 사용한다. 5초(5 Interval)데이터 또한 HT_t와 마찬가지로 현재 큐에 적재되는 양과 근접한 데이터를 이용하고 데이터의 변동폭을 줄이기 위해 결정하였다. 마지막으로 큐에 쌓인 패킷 및 예상되는 추가 패킷들을 신속하게 처리하기 위해 예상되는 큐의 패킷 가중치($Q_{N,t}$)와 HT_t와 RTO의 최소 대기 시간($W_{\min,RTO}$: Minimum Waiting Time of RTO)과의 비율을 통해 추가적인 처리에 필요한 가중치를 구한다. RTO는 네트워크에 사용되는 프로토콜에 따라 계산된다. 예를 들어 CoAP의 경우 Minimum RTO는 2초이며 exponential back-off 방식을 사용한다. 최종적으로 다음 시간에 동작될 HTP의 예비 스레드 수 $P_{N,t+1}$은 4.1과 4.2에서 각각 계산된 예상 스레드 수 $P_{N,t}$와 $Q_{N,t}$중 최대값을 계산하여 이전 시간의 스레드 수 $C_{N,t}$와의 합으로 계산되며 Equation (4)와 같다.

$$Q_{N,t} = \frac{QS_t(req) + e_t(req)}{HT_t(req/s) * W_{\min,RTO}(s)} \tag{3}$$

$$P_{N,t+1} = MAX[P_{N,t}, Q_{N,t}] + C_{N,t} \tag{4}$$

4.3 HTP의 최종 스레드 수 계산

위 Equation (4)를 통해 정보의 처리속도 및 큐의 정보를 고려하여 HTP의 스레드 수를 조정하고 정보를 신속하게 처리할 수 있다. 다중 스레드를 이용한 병렬 처리 방법은 작업을 신속하고 효율적으로 처리할 수 있다. 하지만 스레드의 수 증가는 관리가 쉽지 않고 많은 메모리 사용의 증가 문제가 발생할 수 있다. 특히 스레드 수에 비례하여 시스템의 성능이 좋아지는 것이 아니라 일정 부분을 지나면서 성능이 점점 감소하고 오히려 단일 스레드 방법보다 성능이 감소하는 현상이 나타난다. Fig. 5는 스레드 수에 따른 정보 처리량을 비교 실험한 그래프 이다. Fig. 2-4와 같은 환경에서 측정하였고 서버의 스레드 수를 1개에서 200개까지 늘려 실험한 결과이다. 그림에서 보는 것과 같이 스레드 수가 1~2개일 때에는 평균 요청량(Network Receiver)이 10000~14000 Request/Sec이고 이에 대한 처리량(HTP)은 6000~8000

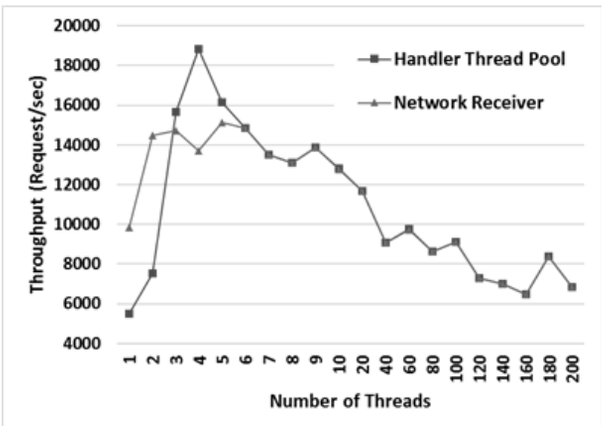


Fig. 5. Throughput of Network Receiver and HTP by Number of Thread

Request/Sec에 불과하다. 또한 스레드의 수가 3~4개까지 증가하면 요청에 대한 처리량이 16000~18000 Request/Sec까지 증가하고 요청에 대한 모든 메시지를 처리할 수 있다. 하지만 그 이후 스레드의 수가 증가함에 따라 처리량이 감소하고 스레드의 수가 7개를 넘어갈 때부터 처리량이 급격히 줄어드는 것을 확인할 수 있다. 특히 스레드의 수가 증가함에 따라 CPU의 많은 자원을 HTP가 점유하기 때문에 Network Receiver의 자원 점유량이 감소하면서 평균 요청량이 감소하는 것을 볼 수 있다.

우리는 Fig. 5를 통해 요청량에 따라 스레드의 수를 제한 없이 증가시키면 처리량이 감소하는 것을 확인하였다. 그러므로 본 논문에서는 지속적으로 스레드 풀의 성능을 측정하고 최대 성능을 유지할 수 있는 최대 스레드 수($NT_{\max,t+1}$: Maximum value of Number of Threads at next t)를 찾아낸다. Equation (5)는 HT (Handler Trend)의 변화 량을 지속적으로 측정하고 HTP의 최대 스레드 수를 찾는 식이다. 시간 t에서 동작하고 있는 HTP의 HT가 증가했다면 최대 스레드 수를 증가시키고 다음 시간 $t+1$의 스레드 결정에 적용한다. 하지만 HT가 더 이상 증가하지 않으면 최대 스레드 수는 더 이상 증가하지 않게 된다. Equation (5)에서 $HT_{d,t}$는 HT_t와 HT_{t-1}의 차이 값($HT_{d,t} = HT_t - HT_{t-1}$)이고 $HT_{\max,t}$는 HT의 최대값을 말한다.

최종적으로 Equation (6)과 같이 HTP의 다음 시간 t의 스레드 수(N_{t+1}: Number of Threads at t+1)를 구할 수 있다.

$$NT_{\max,t+1} = \begin{cases} NT_{\max,t} + 1, & HT_{d,t} < 0 \text{ and } HT_t > HT_{\max,t} \\ NT_{\max,t}, & otherwise \end{cases} \tag{5}$$

$$N_{t+1} = \begin{cases} 1, & P_{N,t+1} < 1 \\ NT_{\max,t} + 1, & P_{N,t+1} > NT_{\max,t+1} \\ P_{N,t+1}, & otherwise \end{cases} \tag{6}$$

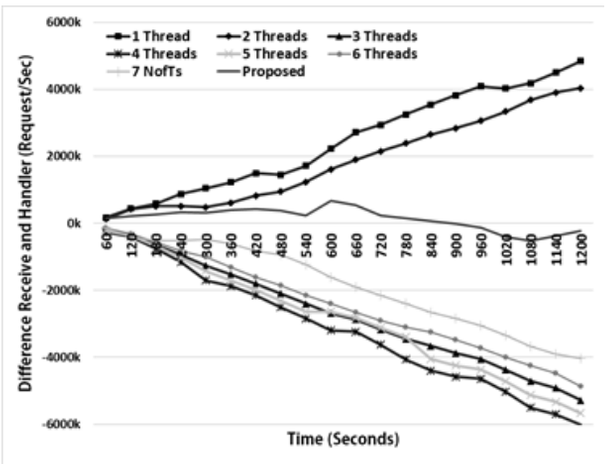


Fig. 6. Cumulative Differences in Throughput between NR and HTP

4.4 스레드 풀 관리 최적화 방법 검증

본문 3.3에서 설명한 것과 같이 스레드 풀은 스레드의 수에 따라 처리하는 정보의 양이 다르기 때문에 고정 스레드 수를 사용하면 처리해야 할 정보의 양보다 처리속도가 낮거나 너무 높을 경우가 생길 수 있다. 본 논문에서는 유입되는 정보의 양에 따라 스레드의 수를 조절하는 스레드 풀 관리 방법을 제안한다. 제안하는 스레드 풀 관리 방법은 패킷의 수신 양과 최대한 비슷한 속도로 패킷을 처리할 수 있어야 한다. 그러므로 간단한 실험을 통해 본 논문에서 제안하는 알고리즘을 이용한 패킷 처리 결과의 유효성을 판단한다.

실험을 위해 우리는 20분동안 지속적으로 Raspberry Pi 기반의 MinT-Server에 초당 40~50K 개의 패킷을 전송 하였다. Fig. 6은 스레드 수 별 Network receiver (NR)와 HTP의 누적된 처리량의 차이 값을 비교한 그래프이다. 우리는 이 그래프에서 값이 양수로 갈수록 NR의 처리량이 높고 값이 음수로 갈수록 Handler의 처리량이 높다는 것을 확인할 수 있다. 스레드의 수가 1~2개인 경우 처리 속도가 NR의 처리량 보다 낮기 때문에 양수 값으로 증가하는 것을 볼 수 있다. 하지만 스레드의 수가 3~7개인 경우 처리속도가 NR의 처리량 보다 높기 때문에 차이 값을 음수 값으로 증가하는 것을 볼 수 있다. 위 그림과 같이 유입되는 패킷의 양과 처리하는 속도의 차이가 나게 되면 처리 속도나 자원관리 효율성 측면에서 문제가 발생할 수 있고 결국 응답 지연 및 신뢰도 감소로 이어질 수 있다.

Fig. 7은 본 논문에서 제안한 스레드 풀 관리 방법을 이용한 시간별 처리량 및 스레드 수를 비교한 그래프이다. 그림에서 보는 것과 같이 패킷을 수신하는 NR의 처리속도에 따라 HTP의 스레드 수를 변화시키기 때문에 HTP의 처리속도가 NR의 처리속도와 유사하게 유지되는 것을 확인할 수 있다. 또한 Fig. 7에서도 제안하는 방법은 차이 값이 계속해서 0에 가깝게 유지되는 것을 볼 수 있다. 그러므로 NR과 HTP의 처리속도가 계속 비슷하게 유지되고 있다는 것을 확인할 수 있었다.

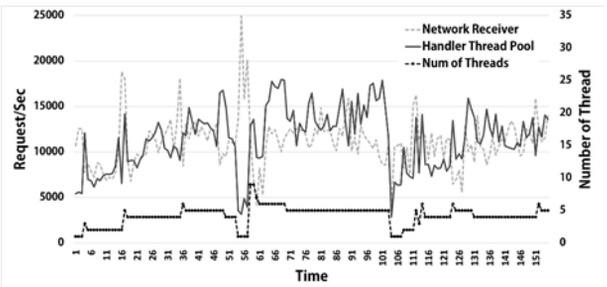


Fig. 7. Throughputs and Number of Threads Using the Proposed Method to Improve Thread Pool Performance

위 Fig. 7을 통해 제안하는 방법은 수신되는 요청량에 따라 스레드 수를 조절하고 처리 속도를 제어할 수 있다는 것을 확인하였다. 특히 기존 미들웨어들은 하드웨어의 CPU 코어 수 에 따라 스레드 풀이 생성되기 때문에 Fig. 3과 같이 저 사양의 하드웨어 플랫폼에서 요청량에 대한 빠른 정보 처리를 할 수 없다. 하지만 제안하는 방법은 VTA를 통해 요청량에 따라 스레드 수를 조절하고 처리속도 높일 수 있기 때문에 기존 방법에 비해 빠른 속도로 정보의 요청에 대응할 수 있다. 또한 Fig. 3과 같이 고 사양의 하드웨어 플랫폼에서 적은 요청량에서 스레드 수를 줄이고 처리 속도를 감소시킬 수 있기 때문에 기존 고정 스레드 방식에 비해 불필요한 자원 및 메모리 소비를 최소화 하여 자원 효율성을 높일 수 있다.

5. 성능 측정

IoT는 다양한 사물들이 네트워크 망을 구성하고 각자의 목적에 맞게 서로 상호작용 및 정보공유를 하기 때문에 서버로 연결이 집중되는 기존 서버-클라이언트 환경 및 사물간 직접 연결이 가능한 M2M 환경을 포함한다. 특히 M2M 환경을 포함하는 IoT 환경에서는 정보가 분산되어 있는 네트워크 환경에서 특정 정보를 가지고 있는 IoT 디바이스에 연결이 집중되는 현상이 발생할 수 있다. 이런 상황에서 신속하지 못한 정보의 처리는 정보를 요청한 디바이스에 대한 응답 지연을 발생시킬 수 있고 전체 IoT 네트워크 환경의 서비스 지연으로 확장될 수 있다. 그러므로 디바이스를 운영하는 미들웨어에서 요청되는 메시지의 효율적이고 신속한 처리가 필요하다.

본 논문에서 우리는 기존 MinT에 스레드를 가변적으로 조절할 수 있도록 향상된 스레드 관리 방법인 VTA을 추가하였다. 특히 본 논문은 요청되는 정보의 빠른 처리를 위해 수신되는 패킷 분석 및 처리, 재송신을 담당하는 HTP의 성능 향상을 목표로 한다. 우리는 본문 3에서 MinT 시스템을 제안 하고 스레드 관리 측면의 성능 저하 문제를 살펴 보았다. 또한 본문 4를 통해 향상된 스레드 관리 방법에 대해 알아보았고 실험을 통해 효율성을 확인하였다. 실험 결과 본 논문에서 제안하는 방법은 기존 방법에 비해 더 효율적

인 정보 처리 능력을 보여줄 것으로 예상된다. 그러므로 IoT 환경 구성에 사용되고 있는 기존 IoT 미들웨어와의 성능 비교를 통해 제안하는 방법에 대한 정보 처리 능력에 대한 효율성을 확인할 필요가 있다.

5.1 실험 환경

실험의 평가를 위해 우리는 CoAP프로토콜을 사용하였다. CoAP는 다양한 IoT 디바이스에서 효율적으로 정보를 전달할 수 있는 어플리케이션 프로토콜이다. 현재 대부분의 IoT 미들웨어는 CoAP를 지원한다. 모든 미들웨어는 간단한 GET 요청(/request)에 대한 응답 데이터("hello world")를 송신한다. 모든 요청 및 응답 데이터는 CoAP헤더를 포함하고 미들웨어의 성능을 평가하기 위해 메시지는 Non-confirmable(NoNs) 형태로 보내진다. 또한 미들웨어의 정확한 성능 측정을 위해 Ethernet 기반의 인터넷 프로토콜(Internet protocol)을 사용한다.

Fig. 8과 같이 본 논문에서는 실험을 위해 총 10대의 테스트베드를 사용하여 서버에 접속할 클라이언트를 생성한다. 클라이언트는 Raspberry Pi 3로 구성된 테스트 서버에 연결되어 테스트를 진행한다. 다양한 미들웨어에서의 적응성과 확장성을 테스트 하기 위해서 클라이언트는 Raspberry Pi 3, Beagle Bone Black, Intel Edison 하드웨어 플랫폼으로 구성된 테스트 서버에 연결되어 각각 테스트를 진행한다. 두 플랫폼 간의 연결을 위해 Gigabit Ethernet 라우터를 사용하였고 이 플랫폼은 각 하드웨어 플랫폼과 유선 및 무선으로 연결된다. 우리는 10대의 테스트 클라이언트 베드를 사용하여 서버에 접속하는 클라이언트의 수를 1개에서 10,000개까지 증가 시킨다. 각 클라이언트는 초당 10개의 패킷을 60초 동안 테스트 시스템으로 전송한다. 실험은 클라이언트 수에 따라 3번의 측정을 실시하고 클라이언트 수가 변경되면 60초 동안의 쿨 타임 시간을 준다.

5.2 IoT 미들웨어들의 성능 비교

수많은 사물들의 요청들은 효율적이고 빠르게 처리되어야 한다. 만약 처리량이 늦어 응답 시간이 증가한다면 다른 사물들의 처리속도에도 영향을 줄 수 있고 극단적으로 IoT 망 전체의 지연이 발생할 수 있다. 이런 처리 지연은 사물들의 작업시간을 늘리고 더 많은 전력을 소모할 수 있다. 이런 문제점을 피하기 위해서는 각 사물은 효율적이고 빠른 응답 속도를 필요로 한다.

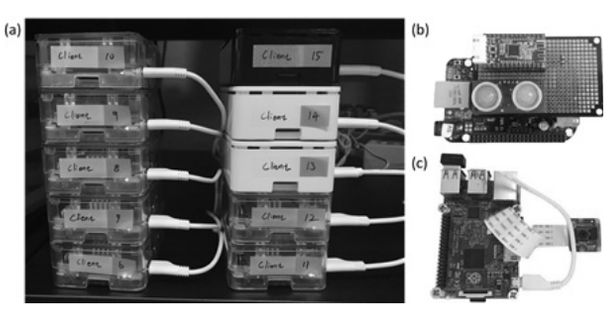


Fig. 8. IoT Experiment Environment Implementation Using MinT

Fig. 9와 10은 기존의 IoT 미들웨어인 Californium, nCoAP, MinT와 개선된 MinT-I의 성능을 비교한 그래프이다. Californium과 nCoAP는 IoT 디바이스의 메시지 처리 및 자원의 관리를 중심으로 하는 미들웨어 이다. 이 두개의 미들웨어는 MinT와 마찬가지로 신속한 정보의 처리를 위해 멀티스레드 기반의 정보 처리 방법을 사용하고 있다. 특히 Californium과 nCoAP는 센서 디바이스 및 하드웨어 미들웨어 관리를 제외한 메시지 처리 성능 향상을 목표로 하고 있다. 그러므로 기존 MinT 미들웨어 뿐만 아니라 Californium 및 nCoAP와의 성능 비교를 통해 본 논문에서 제안하는 MinT-I의 성능 분석을 실시한다.

Fig. 9는 기존의 IoT 미들웨어인 Californium 및 nCoAP, MinT 그리고 본 논문에서 제안한 MinT-I의 성능을 비교한 그래프이다. 또한 Fig. 10은 평균 요청 처리량을 비교한 그래프이다. Fig. 9는 구성된 테스트베드 중 Rapberry Pi 3에서 각 미들웨어 별로 패킷을 전송하는 클라이언트의 수를 1개부터 10,000개까지 증가시켜 가며 성능측정을 하였다. 이 그래프는 각 단계별 60초동안의 초당 패킷수를 보여준다. Fig. 10에서 보는 것과 같이 MinT와 MinT-I, Californium은 700개의 동시 요청까지 비슷한 처리량을 보이고 있다.

Californium은 Fig. 9의 800~1,500클라이언트 수 사이에서 비교적 높은 처리량을 보여주고 있지만 그 이후 급격히 처리량이 감소하는 것을 볼 수 있다. Californium은 요청 처리 속도에 비해 요청 수가 많아지면 작업 큐의 양이 증가하고

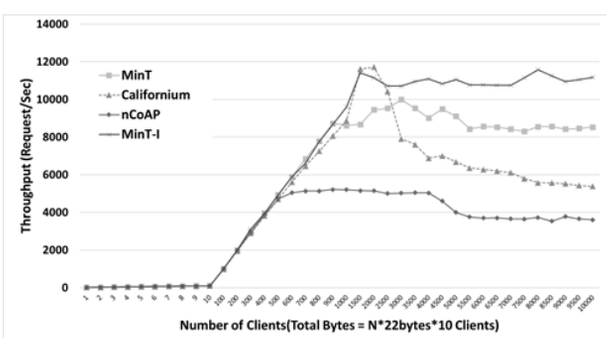


Fig. 9. Throughput of MinT, Californium, nCoAP and Improved MinT

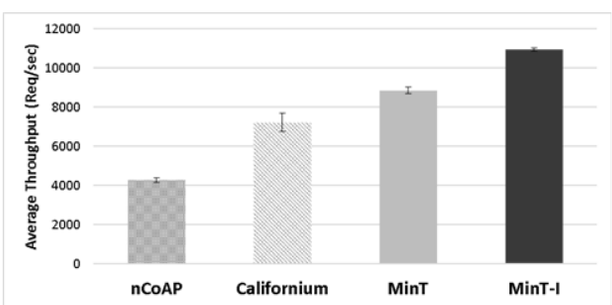


Fig. 10. Average Throughput (SE) of MinT, Californium, nCoAP and MinT-I

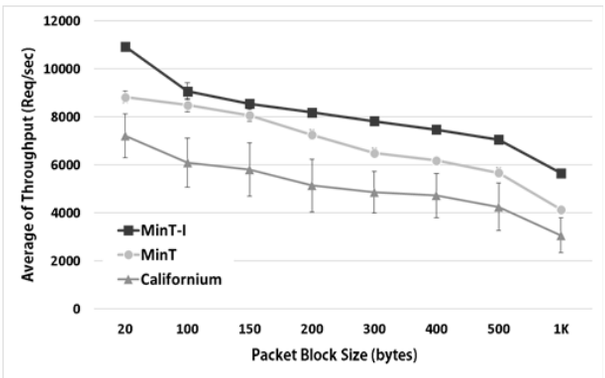


Fig. 11. Average of Throughput by Packet Block Size of MinT-I, MinT and Californium (SE)

처리속도가 감소한다. MinT의 경우 요청 처리 과정을 단순화하고 처리과정의 메모리 사용을 최적화 하였다. 그러므로 동시에 접속하는 클라이언트 수가 증가하여도 Californium에 비해 비교적 높은 처리량을 보여주고 있다.

MinT-I는 기존 MinT의 문제점을 개선한 향상된 정보처리 능력을 가지고 있다. Fig. 10을 보면 800 클라이언트 수부터 MinT와 Californium, nCoAP보다 처리속도가 더 높은 것을 확인할 수 있다. 또한 클라이언트 수가 증가해도 처리 속도가 계속해서 유지되는 것을 볼 수 있다. MinT-I가 하드웨어 플랫폼의 최대 성능을 이용하여 정보를 처리하는 것은 아니다. MinT-I는 수신되는 정보의 양에 따라 처리량을 비슷하게 유지하면서 기존 다른 자원의 효율성을 높여줄 수 있다. MinT-I는 하드웨어 플랫폼의 최대 성능이 아닌 최적의 성능을 유지할 수 있다. 이를 통해 Fig. 11과 같이 하드웨어 플랫폼의 에너지 효율 및 정보 처리 능력을 높일 수 있었다.

5.3 패킷 블록 크기 별 성능 비교

Fig. 11은 MinT-I와 MinT, Californium의 패킷 블록 크기에 따른 미들웨어 별 평균 처리량을 보여준다. 블록사이즈는 헤더를 포함한 기본 크기인 20byte부터 1KB까지 크기를 증가시킨다. 평균값을 구하기 위해 동시에 전송하는 클라이언트 수의 1,000부터 10,000까지의 값을 사용하였다. 실험 결과 블록 사이즈가 증가할수록 MinT-I와 MinT, Californium 모두 초당 패킷 처리량이 감소하는 것을 볼 수 있다. 이는 패킷 크기가 증가할수록 전송 속도가 감소하고 지연시간이 증가하면서 발생하는 결과로 볼 수 있다. Fig. 12에서 보는 것과 같이 모든 패킷 사이즈에서 MinT-I가 MinT에 비해 평균 5%, Californium에 비해 평균 30%의 높은 성능을 보여주고 있다. 또한 Discovery 조건 또는 대량의 정보 요청에 따른 응답에서는 평균 100~200byte의 패킷 사이즈를 사용한다. 이 부분에서도 MinT-I는 MinT 에 비해 평균 3%, Californium에 비해 평균 30%의 높은 성능을 보여준다. 세 가지의 미들웨어에 비해 성능이 낮은 nCoAP는 따로 성능 비교를 실시하지 않았다.

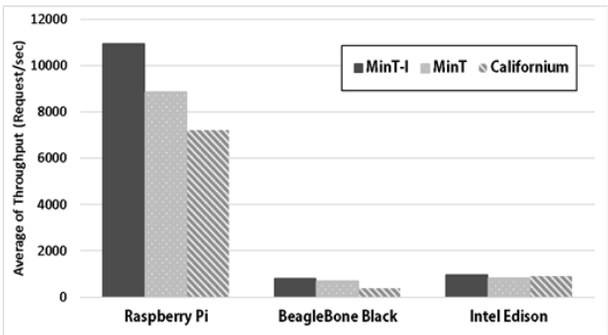


Fig. 12. Average of Throughput by Hardware Platforms with MinT-I, MinT or Californium (SE)

5.4 하드웨어 플랫폼 별 성능 비교

Fig. 12는 MinT-I와 MinT, Californium의 하드웨어 플랫폼 별 평균 처리량을 비교한 그래프이다. 실험에는 Raspberry Pi 3, Intel Edison, BeagleBone Black을 사용하였다. Raspberry Pi 3는 고성능의 쿼드 코어 CPU를 탑재하고 있고 Beagle Bone Black과 Intel Edison은 저성능의 듀얼코어 CPU를 탑재하고 있다. 실험 결과 MinT-I는 기존 MinT에 비해 Raspberry Pi 3는 약 19%, Galaxy S6은 약 12%, Beagle Bone Black은 약 13%, Intel Edison은 약 13%의 성능 향상을 보이고 있다.

MinT-I는 하드웨어의 성능에 따라 메시지를 처리하는 스레드의 개수를 조절하고 이를 통해 성능 및 자원 효율성 향상을 목표로 하고 있다. 그러므로 MinT 각 하드웨어 성능에 따라 메시지 처리량을 조절하고 최대의 성능을 이끌어 낼 수 있기 때문에 그림과 같이 본 논문에서 제안하는 MinT-I는 기존 MinT와 Californium에 비해 높은 처리량을 보여주고 있다. 이를 통해MinT-I는 IoT 환경에서의 전송 지연을 줄이고 에너지 효율적인 IoT 미들웨어 플랫폼이 될 수 있음을 증명할 수 있다.

6. 결 론

우리는 사물 인터넷 환경에서 효율적인 사물 인터넷 디바이스 운영을 위해 MinT를 제안하고 그 효율성을 입증하였다. MinT는 개발자들이 쉽게 디바이스를 개발 운용할 수 있도록 하고 사물간의 효율적이고 빠른 상호작용을 가능하게 하였다. 특히 MinT는 사물들로부터 요청되는 패킷들을 효율적으로 처리하기 위해 멀티 스레드 기반의 스레드 풀을 사용한다. MinT는 패킷의 처리량과 상관없이 스레드 풀의 성능을 위한 스레드의 수를 CPU Core 수와 동일한 크기로 일반화 하여 사용하고 있다. 고정된 스레드 크기는 수신되는 요청 량에 유연하게 대처할 수 없기 때문에 요청에 대한 처리 지연 및 시스템 자원의 비효율성 문제를 발생시킬 수 있다.

본 논문에서는 위 문제를 해결하기 위해 가변적으로 스레드 개수를 조정할 수 있는 향상된 스레드 풀 관리 방법을 제안하였다. 제안하는 방법은 수신되는 패킷 량 및 시스템의 처리량을 분석한다. System Layer의 각 스레드 풀은 분석된 데이터를 이용하여 스레드의 수를 판단할 수 있다. 특히 본

논문은 수신되는 패킷 분석 및 처리, 재송신을 담당하는 Interaction Thread Pool 그룹의 성능 향상을 목표로 한다. 그러므로 Interaction Thread Pool 그룹 중심의 실험을 통해 본 논문에서 제안한 방법의 효율성에 대한 평가를 실시하였다.

우선 제안하는 스레드 풀 관리 방법의 유효성을 판단하기 위해 Interaction Layer의 Network Receiver 및 HTP의 처리량을 비교하였다. 실험 결과 제안하는 방법은 수신되는 처리량에 따라 HTP의 처리 속도가 변화된 것을 확인할 수 있었다. 제안하는 방법의 유효성 평가 후 제안하는 방법을 이용한 MinT-I의 성능 평가를 실시하였다. 성능 평가에는 MinT를 포함한 기존 미들웨어와의 처리량 비교, 패킷 블록사이즈 별 처리량 비교, 하드웨어 플랫폼 별 처리량 비교를 진행하였다. 실험 결과 MinT-I는 MinT에 비해 25%, Californium에 비해 35%의 성능 향상을 보여주고 있다. 특히 고정 스레드 수를 사용하던 기존 MinT에 비해 요청에 빠르게 응답할 수 있는 MinT-I 미들웨어는 IoT 전체 통신 지연시간 및 디바이스의 에너지 소비를 더 감소시킬 수 있다.

현재 IoT 디바이스는 많은 센싱 디바이스를 연결하고 주변 정보를 수집하고 리소스를 공유한다. MinT는 리소스 요청이 오면 센싱 디바이스를 동작시켜 주변 정보를 수집한다. 하지만 리소스 요청 양이 많아지면 MinT의 정보 수집 주기도 많아지기 때문에 위와 같은 방법을 사용한다면 에너지 소비가 증가할 수 있다. 특히 정보의 변화가 크지 않은 센싱 디바이스의 주기적인 센싱은 전송 및 에너지 낭비를 불러올 수 있다. 추후 연구에서는 효율적인 센싱 디바이스의 리소스 관리를 통해 에너지 소비를 감소시킬 수 있는 방법을 연구할 예정이다.

References

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

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

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

[6] A. Dunkels, B. Gronvall, and T. Voigt, “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.

[7] S. B. Jeon and I. B. Jung, “MinT: Middleware for Cooperative Interaction of Things,” *Sensors*, Vol.17, No.6, p.1452, 2017.

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

[9] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks,” In *Communication Systems Software and Middleware and Workshops*, pp.791-798, 2008.

[10] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J. P. Calbimonte, M. Riahi, and L. Skorin-Kapov, “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] K. Klues, C. J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan, “TOSThreads: thread-safe and non-invasive preemption in TinyOS,” In *SenSys*, Vol.9, pp.127-140, Nov., 2009.

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

[13] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “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 [Internet], http://www.iotivity.org.

[15] M. Kovatsch, M. Lanter, and Z. Shelby, “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 [Internet], https://github.com/okleine/nCoAP.

[17] Brillo [Internet], http://developers.google.com/brillo.

[18] Tizen [Internet], https://www.tizen.org/.

전 수 빈

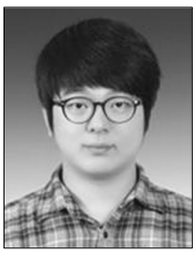
e-mail : sbjeon@kangwon.ac.kr

2010년 강원대학교 컴퓨터정보통신공학과 (학사)

2012년 강원대학교 컴퓨터정보통신공학과 (석사)

현 재 강원대학교 컴퓨터정보통신공학과 박사

관심분야 : Intelligent Transportation System, Sensor Network, Internet of Things



이 충 산

e-mail : zealee@kangwon.ac.kr
2014년 강원대학교 컴퓨터정보통신공학과 (학사)
2016년 강원대학교 컴퓨터정보통신공학과 (석사)
2016년~현 재 강원대학교 컴퓨터정보통신공학과 박사과정
관심분야 : Sensor Network, Internet of Things

한 영 탁

e-mail : gksdudxkr@kangwon.ac.kr
2015년 강원대학교 컴퓨터정보통신공학과 (학사)
2017년 강원대학교 컴퓨터정보통신공학과 (석사)
2017년~현 재 유엔젤 연구원
관심분야 : Intelligent Transportation System, Operating System, Sensor Network, Internet of Things

정 인 범

e-mail : ibjung@kangwon.ac.kr
1985년 고려대학교 전자공학과(학사)
1985년~1995년 삼성전자 선임연구원
1992년~1994년 한국과학기술원 정보통신공학과(석사)
1995년~2000년 한국과학기술원 전산학과(Ph.D.)
2001년~현 재 강원대학교 컴퓨터정보통신공학과 교수
관심분야 : Operating System, Software Engineering, Multimedia System, Sensor Network, Intelligent Transportation System, Internet of Things